Advanced editing ~
|cmdline.txt| Command-line editing
|options.txt| description of all options
|pattern.txt| regexp patterns and search commands
|map.txt| key mapping and abbreviations
|tagsrch.txt| tags and special searches
|quickfix.txt| commands for a quick edit-compile-fix cycle
|windows.txt| commands for using multiple windows and buffers
|tabpage.txt| commands for using multiple tab pages
|syntax.txt| syntax highlighting
|spell.txt| spell checking
|diff.txt| working with two to four versions of the same file
|autocmd.txt| automatically executing commands on an event
|filetype.txt| settings done specifically for a type of file
|eval.txt| expression evaluation, conditional commands
|channel.txt| Jobs, Channels, inter-process communication
|fold.txt| hide (fold) ranges of lines

==
cmdline.txt   For Vim version 8.0. Last change: 2017 Sep 17

                  VIM REFERENCE MANUAL   by Bram Moolenaar

                                Cmdline-mode *Command-line-mode*
Command-line mode   *Cmdline* *Command-line* *mode-cmdline* *:*

Command-line mode is used to enter Ex commands (":"), search patterns
("/" and "?"), and filter commands ("!").

Basic command line editing is explained in chapter 20 of the user manual
|usr_20.txt|.

1. Command-line editing |cmdline-editing|
2. Command-line completion |cmdline-completion|
3. Ex command-lines |cmdline-lines|
4. Ex command-line ranges |cmdline-ranges|
5. Ex command-line flags |ex-flags|
6. Ex special characters |cmdline-special|
7. Command-line window |cmdline-window|

==
1. Command-line editing   *cmdline-editing*

Normally characters are inserted in front of the cursor position. You can
move around in the command-line with the left and right cursor keys. With the
<Insert> key, you can toggle between inserting and overstriking characters.
{Vi: can only alter the last character in the line}

Note that if your keyboard does not have working cursor keys or any of the
other special keys, you can use ":cnoremap" to define another key for them.
For example, to define tcsh style editing keys: *tcsh-style* >
        :cnoremap <C-A> <Home>
        :cnoremap <C-F> <Right>
        :cnoremap <C-B> <Left>
        :cnoremap <Esc>b <S-Left>
        :cnoremap <Esc>f <S-Right>
(<> notation |<>|; type all this literally)

                                                        cmdline-too-long
When the command line is getting longer than what fits on the screen, only the

```
part that fits will be shown.  The cursor can only move in this visible part,
thus you cannot edit beyond that.

                                                    *cmdline-history* *history*
The command-lines that you enter are remembered in a history table.  You can
recall them with the up and down cursor keys.  There are actually five
history tables:
- one for ':' commands
- one for search strings
- one for expressions
- one for input lines, typed for the |input()| function.
- one for debug mode commands
These are completely separate.  Each history can only be accessed when
entering the same type of line.
Use the 'history' option to set the number of lines that are remembered
(default: 50).
Notes:
- When you enter a command-line that is exactly the same as an older one, the
  old one is removed (to avoid repeated commands moving older commands out of
  the history).
- Only commands that are typed are remembered.  Ones that completely come from
  mappings are not put in the history.
- All searches are put in the search history, including the ones that come
  from commands like "*" and "#".  But for a mapping, only the last search is
  remembered (to avoid that long mappings trash the history).
{Vi: no history}
{not available when compiled without the |+cmdline_hist| feature}

There is an automatic completion of names on the command-line; see
|cmdline-completion|.

                                                            *c_CTRL-V*
CTRL-V          Insert next non-digit literally.  Up to three digits form the
                decimal value of a single byte.  The non-digit and the three
                digits are not considered for mapping.  This works the same
                way as in Insert mode (see above, |i_CTRL-V|).
                Note: Under Windows CTRL-V is often mapped to paste text.
                Use CTRL-Q instead then.
                                                            *c_CTRL-Q*
CTRL-Q          Same as CTRL-V.  But with some terminals it is used for
                control flow, it doesn't work then.

                                                            *c_<Left>* *c_Left*
<Left>          cursor left
                                                            *c_<Right>* *c_Right*
<Right>         cursor right
                                                            *c_<S-Left>*
<S-Left> or <C-Left>                                        *c_<C-Left>*
                cursor one WORD left
                                                            *c_<S-Right>*
<S-Right> or <C-Right>                                      *c_<C-Right>*
                cursor one WORD right
CTRL-B or <Home>                                    *c_CTRL-B* *c_<Home>* *c_Home*
                cursor to beginning of command-line
CTRL-E or <End>                                     *c_CTRL-E* *c_<End>* *c_End*
                cursor to end of command-line

                                                            *c_<LeftMouse>*
<LeftMouse>     Move the cursor to the position of the mouse click.

                                                            *c_<MiddleMouse>*
<MiddleMouse>   Paste the contents of the clipboard (for X11 the primary
```

```
                selection).  This is similar to using CTRL-R *, but no CR
                characters are inserted between lines.

CTRL-H                                              *c_<BS>* *c_CTRL-H* *c_BS*
<BS>            Delete the character in front of the cursor (see |:fixdel| if
                your <BS> key does not do what you want).
                                                    *c_<Del>* *c_Del*
<Del>           Delete the character under the cursor (at end of line:
                character before the cursor) (see |:fixdel| if your <Del>
                key does not do what you want).
                                                    *c_CTRL-W*
CTRL-W          Delete the |word| before the cursor.  This depends on the
                'iskeyword' option.
                                                    *c_CTRL-U*
CTRL-U          Remove all characters between the cursor position and
                the beginning of the line.  Previous versions of vim
                deleted all characters on the line.  If that is the
                preferred behavior, add the following to your .vimrc: >
                        :cnoremap <C-U> <C-E><C-U>
<
                                                    *c_<Insert>* *c_Insert*
<Insert>        Toggle between insert and overstrike.  {not in Vi}

{char1} <BS> {char2}    or                          *c_digraph*
CTRL-K {char1} {char2}                              *c_CTRL-K*
                enter digraph (see |digraphs|).  When {char1} is a special
                key, the code for that key is inserted in <> form.  {not in Vi}

CTRL-R {0-9a-z"%#:-=.}                              *c_CTRL-R* *c_<C-R>*
                Insert the contents of a numbered or named register.  Between
                typing CTRL-R and the second character '"' will be displayed
                to indicate that you are expected to enter the name of a
                register.
                The text is inserted as if you typed it, but mappings and
                abbreviations are not used.  Command-line completion through
                'wildchar' is not triggered though.  And characters that end
                the command line are inserted literally (<Esc>, <CR>, <NL>,
                <C-C>).  A <BS> or CTRL-W could still end the command line
                though, and remaining characters will then be interpreted in
                another mode, which might not be what you intended.
                Special registers:
                        '"'     the unnamed register, containing the text of
                                the last delete or yank
                        '%'     the current file name
                        '#'     the alternate file name
                        '*'     the clipboard contents (X11: primary selection)
                        '+'     the clipboard contents
                        '/'     the last search pattern
                        ':'     the last command-line
                        '-'     the last small (less than a line) delete
                        '.'     the last inserted text
                                                    *c_CTRL-R_=*
                        '='     the expression register: you are prompted to
                                enter an expression (see |expression|)
                                (doesn't work at the expression prompt; some
                                things such as changing the buffer or current
                                window are not allowed to avoid side effects)
                                When the result is a |List| the items are used
                                as lines.  They can have line breaks inside
                                too.
                                When the result is a Float it's automatically
                                converted to a String.
```

```
                See |registers| about registers.  {not in Vi}
                Implementation detail: When using the |expression| register
                and invoking setcmdpos(), this sets the position before
                inserting the resulting string.  Use CTRL-R CTRL-R to set the
                position afterwards.

CTRL-R CTRL-F                           *c_CTRL-R_CTRL-F* *c_<C-R>_<C-F>*
CTRL-R CTRL-P                           *c_CTRL-R_CTRL-P* *c_<C-R>_<C-P>*
CTRL-R CTRL-W                           *c_CTRL-R_CTRL-W* *c_<C-R>_<C-W>*
CTRL-R CTRL-A                           *c_CTRL-R_CTRL-A* *c_<C-R>_<C-A>*
                Insert the object under the cursor:
                        CTRL-F  the Filename under the cursor
                        CTRL-P  the Filename under the cursor, expanded with
                                'path' as in |gf|
                        CTRL-W  the Word under the cursor
                        CTRL-A  the WORD under the cursor; see |WORD|

                When 'incsearch' is set the cursor position at the end of the
                currently displayed match is used.  With CTRL-W the part of
                the word that was already typed is not inserted again.

                {not in Vi}
                CTRL-F and CTRL-P: {only when |+file_in_path| feature is
                included}

                                        *c_CTRL-R_CTRL-R* *c_<C-R>_<C-R>*
                                        *c_CTRL-R_CTRL-O* *c_<C-R>_<C-O>*
CTRL-R CTRL-R {0-9a-z"%#:-=. CTRL-F CTRL-P CTRL-W CTRL-A}
CTRL-R CTRL-O {0-9a-z"%#:-=. CTRL-F CTRL-P CTRL-W CTRL-A}
                Insert register or object under the cursor.  Works like
                |c_CTRL-R| but inserts the text literally.  For example, if
                register a contains "xy^Hz" (where ^H is a backspace),
                "CTRL-R a" will insert "xz" while "CTRL-R CTRL-R a" will
                insert "xy^Hz".

CTRL-\ e {expr}                                         *c_CTRL-\_e*
                Evaluate {expr} and replace the whole command line with the
                result.  You will be prompted for the expression, type <Enter>
                to finish it.  It's most useful in mappings though.  See
                |expression|.
                See |c_CTRL-R_=| for inserting the result of an expression.
                Useful functions are |getcmdtype()|, |getcmdline()| and
                |getcmdpos()|.
                The cursor position is unchanged, except when the cursor was
                at the end of the line, then it stays at the end.
                |setcmdpos()| can be used to set the cursor position.
                The |sandbox| is used for evaluating the expression to avoid
                nasty side effects.
                Example: >
                        :cmap <F7> <C-\>eAppendSome()<CR>
                        :func AppendSome()
                           :let cmd = getcmdline() . " Some()"
                           :" place the cursor on the )
                           :call setcmdpos(strlen(cmd))
                           :return cmd
                        :endfunc
<               This doesn't work recursively, thus not when already editing
                an expression.  But it is possible to use in a mapping.

                                                        *c_CTRL-Y*
CTRL-Y          When there is a modeless selection, copy the selection into
                the clipboard. |modeless-selection|
```

```
                If there is no selection CTRL-Y is inserted as a character.

CTRL-M or CTRL-J                *c_CTRL-M* *c_CTRL-J* *c_<NL>* *c_<CR>* *c_CR*
<CR> or <NL>    start entered command

CTRL-[                                          *c_CTRL-[* *c_<Esc>* *c_Esc*
<Esc>           When typed and 'x' not present in 'cpoptions', quit
                Command-line mode without executing.  In macros or when 'x'
                present in 'cpoptions', start entered command.
                Note: If your <Esc> key is hard to hit on your keyboard, train
                yourself to use CTRL-[.
                                                        *c_CTRL-C*
CTRL-C          quit command-line without executing

                                                        *c_<Up>* *c_Up*
<Up>            recall older command-line from history, whose beginning
                matches the current command-line (see below).
                {not available when compiled without the |+cmdline_hist|
                feature}
                                                        *c_<Down>* *c_Down*
<Down>          recall more recent command-line from history, whose beginning
                matches the current command-line (see below).
                {not available when compiled without the |+cmdline_hist|
                feature}

                                                        *c_<S-Up>* *c_<PageUp>*
<S-Up> or <PageUp>
                recall older command-line from history
                {not available when compiled without the |+cmdline_hist|
                feature}
                                                *c_<S-Down>* *c_<PageDown>*
<S-Down> or <PageDown>
                recall more recent command-line from history
                {not available when compiled without the |+cmdline_hist|
                feature}

CTRL-D          command-line completion (see |cmdline-completion|)
'wildchar' option
                command-line completion (see |cmdline-completion|)
CTRL-N          command-line completion (see |cmdline-completion|)
CTRL-P          command-line completion (see |cmdline-completion|)
CTRL-A          command-line completion (see |cmdline-completion|)
CTRL-L          command-line completion (see |cmdline-completion|)

                                                        *c_CTRL-_*
CTRL-_          a - switch between Hebrew and English keyboard mode, which is
                private to the command-line and not related to hkmap.
                This is useful when Hebrew text entry is required in the
                command-line, searches, abbreviations, etc.  Applies only if
                Vim is compiled with the |+rightleft| feature and the
                'allowrevins' option is set.
                See |rileft.txt|.

                b - switch between Farsi and English keyboard mode, which is
                private to the command-line and not related to fkmap.  In
                Farsi keyboard mode the characters are inserted in reverse
                insert manner.  This is useful when Farsi text entry is
                required in the command-line, searches, abbreviations, etc.
                Applies only if Vim is compiled with the |+farsi| feature.
                See |farsi.txt|.

                                                        *c_CTRL-^*
```

```
CTRL-^          Toggle the use of language |:lmap| mappings and/or Input
                Method.
                When typing a pattern for a search command and 'imsearch' is
                not -1, VAL is the value of 'imsearch', otherwise VAL is the
                value of 'iminsert'.
                When language mappings are defined:
                - If VAL is 1 (langmap mappings used) it becomes 0 (no langmap
                  mappings used).
                - If VAL was not 1 it becomes 1, thus langmap mappings are
                  enabled.
                When no language mappings are defined:
                - If VAL is 2 (Input Method is used) it becomes 0 (no input
                  method used)
                - If VAL has another value it becomes 2, thus the Input Method
                  is enabled.
                These language mappings are normally used to type characters
                that are different from what the keyboard produces.  The
                'keymap' option can be used to install a whole number of them.
                When entering a command line, langmap mappings are switched
                off, since you are expected to type a command.  After
                switching it on with CTRL-^, the new state is not used again
                for the next command or Search pattern.
                {not in Vi}

                                                        *c_CTRL-]*
CTRL-]          Trigger abbreviation, without inserting a character.  {not in
                Vi}

For Emacs-style editing on the command-line see |emacs-keys|.

The <Up> and <Down> keys take the current command-line as a search string.
The beginning of the next/previous command-lines are compared with this
string.  The first line that matches is the new command-line.  When typing
these two keys repeatedly, the same string is used again.  For example, this
can be used to find the previous substitute command: Type ":s" and then <Up>.
The same could be done by typing <S-Up> a number of times until the desired
command-line is shown.  (Note: the shifted arrow keys do not work on all
terminals)

                                                              *:his* *:history*
:his[tory]      Print the history of last entered commands.
                {not in Vi}
                {not available when compiled without the |+cmdline_hist|
                feature}

:his[tory] [{name}] [{first}][, [{last}]]
                List the contents of history {name} which can be:
                c[md]    or :           command-line history
                s[earch] or / or ?      search string history
                e[xpr]   or =           expression register history
                i[nput]  or @           input line history
                d[ebug]  or >           debug command history
                a[ll]                   all of the above
                {not in Vi}

                If the numbers {first} and/or {last} are given, the respective
                range of entries from a history is listed.  These numbers can
                be specified in the following form:
                                                              *:history-indexing*
                A positive number represents the absolute index of an entry
                as it is given in the first column of a :history listing.
                This number remains fixed even if other entries are deleted.
```

```
                A negative number means the relative position of an entry,
                counted from the newest entry (which has index -1) backwards.

                Examples:
                List entries 6 to 12 from the search history: >
                        :history / 6,12
<
                List the penultimate entry from all histories: >
                        :history all -2
<
                List the most recent two entries from all histories: >
                        :history all -2,

:keepp[atterns] {command}                       *:keepp* *:keeppatterns*
                Execute {command}, without adding anything to the search
                history

==============================================================================
2. Command-line completion                                *cmdline-completion*

When editing the command-line, a few commands can be used to complete the
word before the cursor.  This is available for:

- Command names: At the start of the command-line.
- Tags: Only after the ":tag" command.
- File names: Only after a command that accepts a file name or a setting for
  an option that can be set to a file name.  This is called file name
  completion.
- Shell command names: After ":!cmd", ":r !cmd" and ":w !cmd".  $PATH is used.
- Options: Only after the ":set" command.
- Mappings: Only after a ":map" or similar command.
- Variable and function names: Only after a ":if", ":call" or similar command.

When Vim was compiled without the |+cmdline_compl| feature only file names,
directories and help items can be completed.  The number of help item matches
is limited (currently to 300) to avoid a long delay when there are very many
matches.

These are the commands that can be used:

                                                          *c_CTRL-D*
CTRL-D          List names that match the pattern in front of the cursor.
                When showing file names, directories are highlighted (see
                'highlight' option).  Names where 'suffixes' matches are moved
                to the end.
                The 'wildoptions' option can be set to "tagfile" to list the
                file of matching tags.
                                        *c_CTRL-I* *c_wildchar* *c_<Tab>*
'wildchar' option
                A match is done on the pattern in front of the cursor.  The
                match (if there are several, the first match) is inserted
                in place of the pattern.  (Note: does not work inside a
                macro, because <Tab> or <Esc> are mostly used as 'wildchar',
                and these have a special meaning in some macros.) When typed
                again and there were multiple matches, the next
                match is inserted.  After the last match, the first is used
                again (wrap around).
                The behavior can be changed with the 'wildmode' option.
                                                          *c_CTRL-N*
CTRL-N          After using 'wildchar' which got multiple matches, go to next
                match.  Otherwise recall more recent command-line from history.
```

```
<S-Tab>                                                 *c_CTRL-P* *c_<S-Tab>*
CTRL-P          After using 'wildchar' which got multiple matches, go to
                previous match.  Otherwise recall older command-line from
                history.  <S-Tab> only works with the GUI, on the Amiga and
                with MS-DOS.
                                                        *c_CTRL-A*
CTRL-A          All names that match the pattern in front of the cursor are
                inserted.
                                                        *c_CTRL-L*
CTRL-L          A match is done on the pattern in front of the cursor.  If
                there is one match, it is inserted in place of the pattern.
                If there are multiple matches the longest common part is
                inserted in place of the pattern.  If the result is shorter
                than the pattern, no completion is done.
                                                        */_CTRL-L*
                When 'incsearch' is set, entering a search pattern for "/" or
                "?" and the current match is displayed then CTRL-L will add
                one character from the end of the current match.  If
                'ignorecase' and 'smartcase' are set and the command line has
                no uppercase characters, the added character is converted to
                lowercase.
                                                    *c_CTRL-G* */_CTRL-G*
CTRL-G          When 'incsearch' is set, entering a search pattern for "/" or
                "?" and the current match is displayed then CTRL-G will move
                to the next match (does not take |search-offset| into account)
                Use CTRL-T to move to the previous match.  Hint: on a regular
                keyboard T is above G.
                                                    *c_CTRL-T* */_CTRL-T*
CTRL-T          When 'incsearch' is set, entering a search pattern for "/" or
                "?" and the current match is displayed then CTRL-T will move
                to the previous match (does not take |search-offset| into
                account).
                Use CTRL-G to move to the next match.  Hint: on a regular
                keyboard T is above G.

The 'wildchar' option defaults to <Tab> (CTRL-E when in Vi compatible mode; in
a previous version <Esc> was used).  In the pattern standard wildcards '*' and
'?' are accepted when matching file names.  '*' matches any string, '?'
matches exactly one character.

The 'wildignorecase' option can be set to ignore case in filenames.

The 'wildmenu' option can be set to show the matches just above the command
line.

If you like tcsh's autolist completion, you can use this mapping:
        :cnoremap X <C-L><C-D>
(Where X is the command key to use, <C-L> is CTRL-L and <C-D> is CTRL-D)
This will find the longest match and then list all matching files.

If you like tcsh's autolist completion, you can use the 'wildmode' option to
emulate it.  For example, this mimics autolist=ambiguous:
        :set wildmode=longest,list
This will find the longest match with the first 'wildchar', then list all
matching files with the next.

                                                        *suffixes*
For file name completion you can use the 'suffixes' option to set a priority
between files with almost the same name.  If there are multiple matches,
those files with an extension that is in the 'suffixes' option are ignored.
The default is ".bak,~,.o,.h,.info,.swp,.obj", which means that files ending
in ".bak", "~", ".o", ".h", ".info", ".swp" and ".obj" are sometimes ignored.
```

An empty entry, two consecutive commas, match a file name that does not
contain a ".", thus has no suffix. This is useful to ignore "prog" and prefer
"prog.c".

Examples:

```
  pattern:      files:                          match:  ~
   test*        test.c test.h test.o            test.c
   test*        test.h test.o                   test.h and test.o
   test*        test.i test.h test.c            test.i and test.c
```

It is impossible to ignore suffixes with two dots.

If there is more than one matching file (after ignoring the ones matching
the 'suffixes' option) the first file name is inserted. You can see that
there is only one match when you type 'wildchar' twice and the completed
match stays the same. You can get to the other matches by entering
'wildchar', CTRL-N or CTRL-P. All files are included, also the ones with
extensions matching the 'suffixes' option.

To completely ignore files with some extension use 'wildignore'.

To match only files that end at the end of the typed text append a "$". For
example, to match only files that end in ".c": >
```
        :e *.c$
```
This will not match a file ending in ".cpp". Without the "$" it does match.

The old value of an option can be obtained by hitting 'wildchar' just after
the '='. For example, typing 'wildchar' after ":set dir=" will insert the
current value of 'dir'. This overrules file name completion for the options
that take a file name.

If you would like using <S-Tab> for CTRL-P in an xterm, put this command in
your .cshrc: >
```
        xmodmap -e "keysym Tab = Tab Find"
```
And this in your .vimrc: >
```
        :cmap <Esc>[1~ <C-P>
```

==
3. Ex command-lines *cmdline-lines*

The Ex commands have a few specialties:

:quote *:comment*
'"' at the start of a line causes the whole line to be ignored. '"'
after a command causes the rest of the line to be ignored. This can be used
to add comments. Example: >
```
        :set ai         "set 'autoindent' option
```
It is not possible to add a comment to a shell command ":!cmd" or to the
":map" command and a few others, because they see the '"' as part of their
argument. This is mentioned where the command is explained.

:bar *:\bar*
'|' can be used to separate commands, so you can give multiple commands in one
line. If you want to use '|' in an argument, precede it with '\'.

These commands see the '|' as their argument, and can therefore not be
followed by another Vim command:
```
    :argdo
    :autocmd
    :bufdo
```

```
    :cdo
    :cfdo
    :command
    :cscope
    :debug
    :folddoopen
    :folddoclosed
    :function
    :global
    :help
    :helpfind
    :lcscope
    :ldo
    :lfdo
    :make
    :normal
    :perl
    :perldo
    :promptfind
    :promptrepl
    :pyfile
    :python
    :registers
    :read !
    :scscope
    :sign
    :tcl
    :tcldo
    :tclfile
    :vglobal
    :windo
    :write !
    :[range]!
    a user defined command without the "-bar" argument |:command|

Note that this is confusing (inherited from Vi): With ":g" the '|' is included
in the command, with ":s" it is not.

To be able to use another command anyway, use the ":execute" command.
Example (append the output of "ls" and jump to the first line): >
        :execute 'r !ls' | '[

There is one exception: When the 'b' flag is present in 'cpoptions', with the
":map" and ":abbr" commands and friends CTRL-V needs to be used instead of
'\'.  You can also use "<Bar>" instead.  See also |map_bar|.

Examples: >
        :!ls | wc               view the output of two commands
        :r !ls | wc             insert the same output in the text
        :%g/foo/p|>             moves all matching lines one shiftwidth
        :%s/foo/bar/|>          moves one line one shiftwidth
        :map q 10^V|            map "q" to "10|"
        :map q 10\| map \ l     map "q" to "10\" and map "\" to "l"
                                        (when 'b' is present in 'cpoptions')

You can also use <NL> to separate commands in the same way as with '|'.  To
insert a <NL> use CTRL-V CTRL-J.  "^@" will be shown.  Using '|' is the
preferred method.  But for external commands a <NL> must be used, because a
'|' is included in the external command.  To avoid the special meaning of <NL>
it must be preceded with a backslash.  Example: >
        :r !date<NL>-join
This reads the current date into the file and joins it with the previous line.
```

```
Note that when the command before the '|' generates an error, the following
commands will not be executed.


Because of Vi compatibility the following strange commands are supported: >
        :|                      print current line (like ":p")
        :3|                     print line 3 (like ":3p")
        :3                      goto line 3

A colon is allowed between the range and the command name.  It is ignored
(this is Vi compatible).  For example: >
        :1,$:s/pat/string

When the character '%' or '#' is used where a file name is expected, they are
expanded to the current and alternate file name (see the chapter "editing
files" |:_%| |:_#|).

Embedded spaces in file names are allowed on the Amiga if one file name is
expected as argument.  Trailing spaces will be ignored, unless escaped with a
backslash or CTRL-V.  Note that the ":next" command uses spaces to separate
file names.  Escape the spaces to include them in a file name.  Example: >
        :next foo\ bar goes\ to school\
starts editing the three files "foo bar", "goes to" and "school ".

When you want to use the special characters '"' or '|' in a command, or want
to use '%' or '#' in a file name, precede them with a backslash.  The
backslash is not required in a range and in the ":substitute" command.
See also |`=|.

                                                              *:_!*
The '!' (bang) character after an Ex command makes the command behave in a
different way.  The '!' should be placed immediately after the command, without
any blanks in between.  If you insert blanks the '!' will be seen as an
argument for the command, which has a different meaning.  For example:
        :w! name        write the current buffer to file "name", overwriting
                        any existing file
        :w !name        send the current buffer as standard input to command
                        "name"

==============================================================================
4. Ex command-line ranges       *cmdline-ranges* *[range]* *E16*

Some Ex commands accept a line range in front of them.  This is noted as
[range].  It consists of one or more line specifiers, separated with ',' or
';'.

The basics are explained in section |10.3| of the user manual.

                                                      *:,* *:;*
When separated with ';' the cursor position will be set to that line
before interpreting the next line specifier.  This doesn't happen for ','.
Examples: >
   4,/this line/
<       from line 4 till match with "this line" after the cursor line. >
   5;/that line/
<       from line 5 till match with "that line" after line 5.

The default line specifier for most commands is the cursor position, but the
commands ":write" and ":global" have the whole file (1,$) as default.

If more line specifiers are given than required for the command, the first
```

```
one(s) will be ignored.

Line numbers may be specified with:             *:range* *E14* *{address}*
        {number}        an absolute line number
        .               the current line                        *:.*
        $               the last line in the file               *:$*
        %               equal to 1,$ (the entire file)          *:%*
        't              position of mark t (lowercase)          *:'*
        'T              position of mark T (uppercase); when the mark is in
                        another file it cannot be used in a range
        /{pattern}[/]   the next line where {pattern} matches     *:/*
        ?{pattern}[?]   the previous line where {pattern} matches *:?*
        \/              the next line where the previously used search
                        pattern matches
        \?              the previous line where the previously used search
                        pattern matches
        \&              the next line where the previously used substitute
                        pattern matches

Each may be followed (several times) by '+' or '-' and an optional number.
This number is added or subtracted from the preceding line number.  If the
number is omitted, 1 is used.

The "/" and "?" after {pattern} are required to separate the pattern from
anything that follows.

The "/" and "?" may be preceded with another address.  The search starts from
there.  The difference from using ';' is that the cursor isn't moved.
Examples: >
        /pat1//pat2/    Find line containing "pat2" after line containing
                        "pat1", without moving the cursor.
        7;/pat2/        Find line containing "pat2", after line 7, leaving
                        the cursor in line 7.

The {number} must be between 0 and the number of lines in the file.  When
using a 0 (zero) this is interpreted as a 1 by most commands.  Commands that
use it as a count do use it as a zero (|:tag|, |:pop|, etc).  Some commands
interpret the zero as "before the first line" (|:read|, search pattern, etc).

Examples: >
        .+3             three lines below the cursor
        /that/+1        the line below the next line containing "that"
        .,$             from current line until end of file
        0;/that         the first line containing "that", also matches in the
                        first line.
        1;/that         the first line after line 1 containing "that"

Some commands allow for a count after the command.  This count is used as the
number of lines to be used, starting with the line given in the last line
specifier (the default is the cursor line).  The commands that accept a count
are the ones that use a range but do not have a file name argument (because
a file name can also be a number).

Examples: >
        :s/x/X/g 5      substitute 'x' by 'X' in the current line and four
                        following lines
        :23d 4          delete lines 23, 24, 25 and 26


Folds and Range

When folds are active the line numbers are rounded off to include the whole
```

closed fold. See |fold-behavior|.

Reverse Range *E493*

A range should have the lower line number first. If this is not the case, Vim will ask you if it should swap the line numbers.
        Backwards range given, OK to swap ~
This is not done within the global command ":g".

You can use ":silent" before a command to avoid the question, the range will always be swapped then.

Count and Range *N:*

When giving a count before entering ":", this is translated into:
                :.,.+(count - 1)
In words: The 'count' lines at and after the cursor. Example: To delete three lines: >
                3:d<CR>   is translated into: .,.+2d<CR>
<

Visual Mode and Range *v_:*

{Visual}:   Starts a command-line with the Visual selected lines as a
                range. The code `:'<,'>` is used for this range, which makes
                it possible to select a similar line from the command-line
                history for repeating a command on different Visually selected
                lines.
                When Visual mode was already ended, a short way to use the
                Visual area for a range is `:*`. This requires that "*" does
                not appear in 'cpo', see |cpo-star|. Otherwise you will have
                to type `:'<,'>`

==
5. Ex command-line flags *ex-flags*

These flags are supported by a selection of Ex commands. They print the line that the cursor ends up after executing the command:

        l   output like for |:list|
        #   add line number
        p   output like for |:print|

The flags can be combined, thus "l#" uses both a line number and |:list| style output.

==
6. Ex special characters *cmdline-special*

Note: These are special characters in the executed command line. If you want to insert special things while typing you can use the CTRL-R command. For example, "%" stands for the current file name, while CTRL-R % inserts the current file name right away. See |c_CTRL-R|.

Note: If you want to avoid the effects of special characters in a Vim script you may want to use |fnameescape()|. Also see |`=|.

In Ex commands, at places where a file name can be used, the following

```
characters have a special meaning.  These can also be used in the expression
function |expand()|.
        %       Is replaced with the current file name.           *:_%* *c_%*
        #       Is replaced with the alternate file name.         *:_#* *c_#*
                This is remembered for every window.
        #n      (where n is a number) is replaced with            *:_#0* *:_#n*
                the file name of buffer n.  "#0" is the same as "#".     *c_#n*
        ##      Is replaced with all names in the argument list   *:_##* *c_##*
                concatenated, separated by spaces.  Each space in a name
                is preceded with a backslash.
        #<n     (where n is a number > 0) is replaced with old    *:_#<* *c_#<*
                file name n.  See |:oldfiles| or |v:oldfiles| to get the
                number.                                                 *E809*
                {only when compiled with the |+eval| and |+viminfo| features}

Note that these, except "#<n", give the file name as it was typed.  If an
absolute path is needed (when using the file name from a different directory),
you need to add ":p".  See |filename-modifiers|.

The "#<n" item returns an absolute path, but it will start with "~/" for files
below your home directory.

Note that backslashes are inserted before spaces, so that the command will
correctly interpret the file name.  But this doesn't happen for shell
commands.  For those you probably have to use quotes (this fails for files
that contain a quote and wildcards): >
        :!ls "%"
        :r !spell "%"

To avoid the special meaning of '%' and '#' insert a backslash before it.
Detail: The special meaning is always escaped when there is a backslash before
it, no matter how many backslashes.
        you type:               result  ~
           #                    alternate.file
           \#                   #
           \\#                  \#
Also see |`=|.

                               *:<cword>* *:<cWORD>* *:<cfile>* *<cfile>*
                               *:<sfile>* *<sfile>* *:<afile>* *<afile>*
                               *:<abuf>* *<abuf>* *:<amatch>* *<amatch>*
                               *:<cexpr>* *<cexpr>*
                               *<slnum>* *E495* *E496* *E497* *E499* *E500*
Note: these are typed literally, they are not special keys!
        <cword>    is replaced with the word under the cursor (like |star|)
        <cWORD>    is replaced with the WORD under the cursor (see |WORD|)
        <cexpr>    is replaced with the word under the cursor, including more
                   to form a C expression.  E.g., when the cursor is on "arg"
                   of "ptr->arg" then the result is "ptr->arg"; when the
                   cursor is on "]" of "list[idx]" then the result is
                   "list[idx]".  This is used for |v:beval_text|.
        <cfile>    is replaced with the path name under the cursor (like what
                   |gf| uses)
        <afile>    When executing autocommands, is replaced with the file name
                   for a file read or write.
        <abuf>     When executing autocommands, is replaced with the currently
                   effective buffer number (for ":r file" and ":so file" it is
                   the current buffer, the file being read/sourced is not in a
                   buffer).
        <amatch>   When executing autocommands, is replaced with the match for
                   which this autocommand was executed.  It differs from
                   <afile> only when the file name isn't used to match with
```

```
                (for FileType, Syntax and SpellFileMissing events).
        <sfile>    When executing a ":source" command, is replaced with the
                file name of the sourced file.  *E498*
                When executing a function, is replaced with:
                "function {function-name}[{lnum}]"
                function call nesting is indicated like this:
                "function {function-name1}[{lnum}]..{function-name2}[{lnum}]"
                Note that filename-modifiers are useless when <sfile> is
                used inside a function.
        <slnum>    When executing a ":source" command, is replaced with the
                line number.  *E842*
                When executing a function it's the line number relative to
                the start of the function.

                                                        *filename-modifiers*
*:_%:* *::8* *::p* *::.* *::~* *::h* *::t* *::r* *::e* *::s* *::gs* *::S*
     *%:8* *%:p* *%:.* *%:~* *%:h* *%:t* *%:r* *%:e* *%:s* *%:gs* *%:S*
The file name modifiers can be used after "%", "#", "#n", "<cfile>", "<sfile>",
"<afile>" or "<abuf>".  They are also used with the |fnamemodify()| function.
These are not available when Vim has been compiled without the |+modify_fname|
feature.
These modifiers can be given, in this order:
        :p      Make file name a full path.  Must be the first modifier.  Also
                changes "~/" (and "~user/" for Unix and VMS) to the path for
                the home directory.  If the name is a directory a path
                separator is added at the end.  For a file name that does not
                exist and does not have an absolute path the result is
                unpredictable.  On MS-Windows an 8.3 filename is expanded to
                the long name.
        :8      Converts the path to 8.3 short format (currently only on
                MS-Windows).  Will act on as much of a path that is an
                existing path.
        :~      Reduce file name to be relative to the home directory, if
                possible.  File name is unmodified if it is not below the home
                directory.
        :.      Reduce file name to be relative to current directory, if
                possible.  File name is unmodified if it is not below the
                current directory.
                For maximum shortness, use ":~:.".
        :h      Head of the file name (the last component and any separators
                removed).  Cannot be used with :e, :r or :t.
                Can be repeated to remove several components at the end.
                When the file name ends in a path separator, only the path
                separator is removed.  Thus ":p:h" on a directory name results
                on the directory name itself (without trailing slash).
                When the file name is an absolute path (starts with "/" for
                Unix; "x:\" for MS-DOS, WIN32, OS/2; "drive:" for Amiga), that
                part is not removed.  When there is no head (path is relative
                to current directory) the result is empty.
        :t      Tail of the file name (last component of the name).  Must
                precede any :r or :e.
        :r      Root of the file name (the last extension removed).  When
                there is only an extension (file name that starts with '.',
                e.g., ".vimrc"), it is not removed.  Can be repeated to remove
                several extensions (last one first).
        :e      Extension of the file name.  Only makes sense when used alone.
                When there is no extension the result is empty.
                When there is only an extension (file name that starts with
                '.'), the result is empty.  Can be repeated to include more
                extensions.  If there are not enough extensions (but at least
                one) as much as possible are included.
        :s?pat?sub?
```

```
                Substitute the first occurrence of "pat" with "sub".  This
                works like the |:s| command.  "pat" is a regular expression.
                Any character can be used for '?', but it must not occur in
                "pat" or "sub".
                After this, the previous modifiers can be used again.  For
                example ":p", to make a full path after the substitution.
        :gs?pat?sub?
                Substitute all occurrences of "pat" with "sub".  Otherwise
                this works like ":s".
        :S      Escape special characters for use with a shell command (see
                |shellescape()|). Must be the last one. Examples: >
                    :!dir <cfile>:S
                    :call system('chmod +w -- ' . expand('%:S'))

Examples, when the file name is "src/version.c", current dir
"/home/mool/vim": >
  :p                    /home/mool/vim/src/version.c
  :p:.                             src/version.c
  :p:~                      ~/vim/src/version.c
  :h                               src
  :p:h                  /home/mool/vim/src
  :p:h:h                /home/mool/vim
  :t                                   version.c
  :p:t                                 version.c
  :r                               src/version
  :p:r                  /home/mool/vim/src/version
  :t:r                                 version
  :e                                             c
  :s?version?main?                 src/main.c
  :s?version?main?:p    /home/mool/vim/src/main.c
  :p:gs?/?\\?           \home\mool\vim\src\version.c

Examples, when the file name is "src/version.c.gz": >
  :p                    /home/mool/vim/src/version.c.gz
  :e                                               gz
  :e:e                                           c.gz
  :e:e:e                                         c.gz
  :e:e:r                                         c
  :r                                   src/version.c
  :r:e                                           c
  :r:r                                 src/version
  :r:r:r                               src/version
<
                                        *extension-removal* *:_%<*
If a "<" is appended to "%", "#", "#n" or "CTRL-V p" the extension of the file
name is removed (everything after and including the last '.' in the file
name).  This is included for backwards compatibility with version 3.0, the
":r" form is preferred.  Examples: >

        %               current file name
        %<              current file name without extension
        #               alternate file name for current window
        #<              idem, without extension
        #31             alternate file number 31
        #31<            idem, without extension
        <cword>         word under the cursor
        <cWORD>         WORD under the cursor (see |WORD|)
        <cfile>         path name under the cursor
        <cfile><        idem, without extension

Note: Where a file name is expected wildcards expansion is done.  On Unix the
shell is used for this, unless it can be done internally (for speed).
```

```
Unless in |restricted-mode|, backticks work also, like in >
        :n `echo *.c`
But expansion is only done if there are any wildcards before expanding the
'%', '#', etc..  This avoids expanding wildcards inside a file name.  If you
want to expand the result of <cfile>, add a wildcard character to it.
Examples: (alternate file name is "?readme?")
        command         expands to  ~
        :e #            :e ?readme?
        :e `ls #`       :e {files matching "?readme?"}
        :e #.*          :e {files matching "?readme?.*"}
        :cd <cfile>     :cd {file name under cursor}
        :cd <cfile>*    :cd {file name under cursor plus "*" and then expanded}
Also see |`=|.

When the expanded argument contains a "!" and it is used for a shell command
(":!cmd", ":r !cmd" or ":w !cmd"), the "!" is escaped with a backslash to
avoid it being expanded into a previously used command.  When the 'shell'
option contains "sh", this is done twice, to avoid the shell trying to expand
the "!".

                                                            *filename-backslash*
For filesystems that use a backslash as directory separator (MS-DOS, Windows,
OS/2), it's a bit difficult to recognize a backslash that is used to escape
the special meaning of the next character.  The general rule is: If the
backslash is followed by a normal file name character, it does not have a
special meaning.  Therefore "\file\foo" is a valid file name, you don't have
to type the backslash twice.

An exception is the '$' sign.  It is a valid character in a file name.  But
to avoid a file name like "$home" to be interpreted as an environment variable,
it needs to be preceded by a backslash.  Therefore you need to use "/\$home"
for the file "$home" in the root directory.  A few examples:

        FILE NAME       INTERPRETED AS  ~
        $home           expanded to value of environment var $home
        \$home          file "$home" in current directory
        /\$home         file "$home" in root directory
        \\$home         file "\\", followed by expanded $home

Also see |`=|.

==============================================================================
7. Command-line window                          *cmdline-window* *cmdwin*
                                                        *command-line-window*
In the command-line window the command line can be edited just like editing
text in any window.  It is a special kind of window, because you cannot leave
it in a normal way.
{not available when compiled without the |+cmdline_hist| or |+vertsplit|
feature}


OPEN                                            *c_CTRL-F* *q:* *q/* *q?*

There are two ways to open the command-line window:
1. From Command-line mode, use the key specified with the 'cedit' option.
   The default is CTRL-F when 'compatible' is not set.
2. From Normal mode, use the "q:", "q/" or "q?" command.
   This starts editing an Ex command-line ("q:") or search string ("q/" or
   "q?").  Note that this is not possible while recording is in progress (the
   "q" stops recording then).

When the window opens it is filled with the command-line history.  The last
```

line contains the command as typed so far. The left column will show a
character that indicates the type of command-line being edited, see
|cmdwin-char|.

Vim will be in Normal mode when the editor is opened, except when 'insertmode'
is set.

The height of the window is specified with 'cmdwinheight' (or smaller if there
is no room). The window is always full width and is positioned just above the
command-line.

EDIT

You can now use commands to move around and edit the text in the window. Both
in Normal mode and Insert mode.

It is possible to use ":", "/" and other commands that use the command-line,
but it's not possible to open another command-line window then. There is no
nesting.
E11
The command-line window is not a normal window. It is not possible to move to
another window or edit another buffer. All commands that would do this are
disabled in the command-line window. Of course it _is_ possible to execute
any command that you entered in the command-line window. Other text edits are
discarded when closing the window.

CLOSE *E199*

There are several ways to leave the command-line window:

<CR> Execute the command-line under the cursor. Works both in Insert and in Normal mode.
CTRL-C Continue in Command-line mode. The command-line under the cursor is used as the command-line. Works both in Insert and in Normal mode. There is no redraw, thus the window will remain visible.
:quit Discard the command line and go back to Normal mode. ":close", ":exit", ":xit" and CTRL-\ CTRL-N also work.
:qall Quit Vim, unless there are changes in some buffer.
:qall! Quit Vim, discarding changes to any buffer.

Once the command-line window is closed the old window sizes are restored. The
executed command applies to the window and buffer where the command-line was
started from. This works as if the command-line window was not there, except
that there will be an extra screen redraw.
The buffer used for the command-line window is deleted. Any changes to lines
other than the one that is executed with <CR> are lost.

If you would like to execute the command under the cursor and then have the
command-line window open again, you may find this mapping useful: >

```
:autocmd CmdwinEnter * map <buffer> <F5> <CR>q:
```

VARIOUS

The command-line window cannot be used:
- when there already is a command-line window (no nesting)
- for entering an encryption key or when using inputsecret()
- when Vim was not compiled with the |+vertsplit| feature

```
Some options are set when the command-line window is opened:
'filetype'      "vim", when editing an Ex command-line; this starts Vim syntax
                highlighting if it was enabled
'rightleft'     off
'modifiable'    on
'buftype'       "nofile"
'swapfile'      off

It is allowed to write the buffer contents to a file.  This is an easy way to
save the command-line history and read it back later.

If the 'wildchar' option is set to <Tab>, and the command-line window is used
for an Ex command, then two mappings will be added to use <Tab> for completion
in the command-line window, like this: >
        :imap <buffer> <Tab> <C-X><C-V>
        :nmap <buffer> <Tab> a<C-X><C-V>
Note that hitting <Tab> in Normal mode will do completion on the next
character.  That way it works at the end of the line.
If you don't want these mappings, disable them with: >
        au CmdwinEnter [:>] iunmap <Tab>
        au CmdwinEnter [:>] nunmap <Tab>
You could put these lines in your vimrc file.

While in the command-line window you cannot use the mouse to put the cursor in
another window, or drag statuslines of other windows.  You can drag the
statusline of the command-line window itself and the statusline above it.
Thus you can resize the command-line window, but not others.

The |getcmdwintype()| function returns the type of the command-line being
edited as described in |cmdwin-char|.


AUTOCOMMANDS

Two autocommand events are used: |CmdwinEnter| and |CmdwinLeave|.  Since this
window is of a special type, the WinEnter, WinLeave, BufEnter and BufLeave
events are not triggered.  You can use the Cmdwin events to do settings
specifically for the command-line window.  Be careful not to cause side
effects!
Example: >
        :au CmdwinEnter :  let b:cpt_save = &cpt | set cpt=.
        :au CmdwinLeave :  let &cpt = b:cpt_save
This sets 'complete' to use completion in the current window for |i_CTRL-N|.
Another example: >
        :au CmdwinEnter [/?]  startinsert
This will make Vim start in Insert mode in the command-line window.

                                                *cmdwin-char*
The character used for the pattern indicates the type of command-line:
        :       normal Ex command
        >       debug mode command |debug-mode|
        /       forward search string
        ?       backward search string
        =       expression for "= |expr-register|
        @       string for |input()|
        -       text for |:insert| or |:append|

 vim:tw=78:ts=8:ft=help:norl:
*options.txt*   For Vim version 8.0.  Last change: 2017 Sep 24
```

VIM REFERENCE MANUAL by Bram Moolenaar

Options *options*

1. Setting options |set-option|
2. Automatically setting options |auto-setting|
3. Options summary |option-summary|

For an overview of options see quickref.txt |option-list|.

Vim has a number of internal variables and switches which can be set to achieve special effects. These options come in three forms:

boolean	can only be on or off	*boolean* *toggle*
number	has a numeric value	
string	has a string value	

==========

1. Setting options *set-option* *E764*

:se *:set*

:se[t] Show all options that differ from their default value.

:se[t] all Show all but terminal options.

:se[t] termcap Show all terminal options. Note that in the GUI the key codes are not shown, because they are generated internally and can't be changed. Changing the terminal codes in the GUI is not useful either...

E518 *E519*

:se[t] {option}? Show value of {option}.

:se[t] {option} Toggle option: set, switch it on.
Number option: show value.
String option: show value.

:se[t] no{option} Toggle option: Reset, switch it off.

:set-! *:set-inv*

:se[t] {option}! or
:se[t] inv{option} Toggle option: Invert value. {not in Vi}

:set-default *:set-&* *:set-&vi* *:set-&vim*

:se[t] {option}& Reset option to its default value. May depend on the current value of 'compatible'. {not in Vi}
:se[t] {option}&vi Reset option to its Vi default value. {not in Vi}
:se[t] {option}&vim Reset option to its Vim default value. {not in Vi}

:se[t] all& Set all options to their default value. The values of these options are not changed:
- all terminal options, starting with t_
- 'columns'
- 'cryptmethod'
- 'encoding'
- 'key'
- 'lines'
- 'term'
- 'ttymouse'
- 'ttytype'

Warning: This may have a lot of side effects. {not in Vi}

:set-args *E487* *E521*
:se[t] {option}={value} or
:se[t] {option}:{value}
Set string or number option to {value}.
For numeric options the value can be given in decimal, hex (preceded with 0x) or octal (preceded with '0'). The old value can be inserted by typing 'wildchar' (by default this is a <Tab> or CTRL-E if 'compatible' is set). See |cmdline-completion|.
White space between {option} and '=' is allowed and will be ignored. White space between '=' and {value} is not allowed.
See |option-backslash| for using white space and backslashes in {value}.

:se[t] {option}+={value} *:set+=*
Add the {value} to a number option, or append the {value} to a string option. When the option is a comma separated list, a comma is added, unless the value was empty.
If the option is a list of flags, superfluous flags are removed. When adding a flag that was already present the option value doesn't change.
Also see |:set-args| above.
{not in Vi}

:se[t] {option}^={value} *:set^=*
Multiply the {value} to a number option, or prepend the {value} to a string option. When the option is a comma separated list, a comma is added, unless the value was empty.
Also see |:set-args| above.
{not in Vi}

:se[t] {option}-={value} *:set-=*
Subtract the {value} from a number option, or remove the {value} from a string option, if it is there.
If the {value} is not found in a string option, there is no error or warning. When the option is a comma separated list, a comma is deleted, unless the option becomes empty.
When the option is a list of flags, {value} must be exactly as they appear in the option. Remove flags one by one to avoid problems.
Also see |:set-args| above.
{not in Vi}

The {option} arguments to ":set" may be repeated. For example: >
:set ai nosi sw=3 ts=3
If you make an error in one of the arguments, an error message will be given and the following arguments will be ignored.

:set-verbose
When 'verbose' is non-zero, displaying an option value will also tell where it was last set. Example: >
:verbose set shiftwidth cindent?
< shiftwidth=4 ~
Last set from modeline ~
cindent ~
Last set from /usr/local/share/vim/vim60/ftplugin/c.vim ~
This is only done when specific option values are requested, not for ":verbose

```
set all" or ":verbose set" without an argument.
When the option was set by hand there is no "Last set" message.
When the option was set while executing a function, user command or
autocommand, the script in which it was defined is reported.
Note that an option may also have been set as a side effect of setting
'compatible'.
A few special texts:
        Last set from modeline ~
                Option was set in a |modeline|.
        Last set from --cmd argument ~
                Option was set with command line argument |--cmd| or +.
        Last set from -c argument ~
                Option was set with command line argument |-c|, +, |-S| or
                |-q|.
        Last set from environment variable ~
                Option was set from an environment variable, $VIMINIT,
                $GVIMINIT or $EXINIT.
        Last set from error handler ~
                Option was cleared when evaluating it resulted in an error.

{not available when compiled without the |+eval| feature}

                                                        *:set-termcap* *E522*
For {option} the form "t_xx" may be used to set a terminal option.  This will
override the value from the termcap.  You can then use it in a mapping.  If
the "xx" part contains special characters, use the <t_xx> form: >
        :set <t_#4>=^[Ot
This can also be used to translate a special code for a normal key.  For
example, if Alt-b produces <Esc>b, use this: >
        :set <M-b>=^[b
(the ^[ is a real <Esc> here, use CTRL-V <Esc> to enter it)
The advantage over a mapping is that it works in all situations.

You can define any key codes, e.g.: >
        :set t_xy=^[foo;
There is no warning for using a name that isn't recognized.  You can map these
codes as you like: >
        :map <t_xy> something
<                                                           *E846*
When a key code is not set, it's like it does not exist.  Trying to get its
value will result in an error: >
        :set t_kb=
        :set t_kb
        E846: Key code not set: t_kb

The t_xx options cannot be set from a |modeline| or in the |sandbox|, for
security reasons.

The listing from ":set" looks different from Vi.  Long string options are put
at the end of the list.  The number of options is quite large.  The output of
"set all" probably does not fit on the screen, causing Vim to give the
|more-prompt|.

                                                        *option-backslash*
To include white space in a string option value it has to be preceded with a
backslash.  To include a backslash you have to use two.  Effectively this
means that the number of backslashes in an option value is halved (rounded
down).
A few examples: >
   :set tags=tags\ /usr/tags        results in "tags /usr/tags"
   :set tags=tags\\,file            results in "tags\,file"
   :set tags=tags\\\ file           results in "tags\ file"
```

```
The "|" character separates a ":set" command from a following command.  To
include the "|" in the option value, use "\|" instead.  This example sets the
'titlestring' option to "hi|there": >
   :set titlestring=hi\|there
This sets the 'titlestring' option to "hi" and 'iconstring' to "there": >
   :set titlestring=hi|set iconstring=there

Similarly, the double quote character starts a comment.  To include the '"' in
the option value, use '\"' instead.  This example sets the 'titlestring'
option to 'hi "there"': >
   :set titlestring=hi\ \"there\"

For MS-DOS and WIN32 backslashes in file names are mostly not removed.  More
precise: For options that expect a file name (those where environment
variables are expanded) a backslash before a normal file name character is not
removed.  But a backslash before a special character (space, backslash, comma,
etc.) is used like explained above.
There is one special situation, when the value starts with "\\": >
   :set dir=\\machine\path          results in "\\machine\path"
   :set dir=\\\\machine\\path       results in "\\machine\path"
   :set dir=\\path\\file            results in "\\path\file" (wrong!)
For the first one the start is kept, but for the second one the backslashes
are halved.  This makes sure it works both when you expect backslashes to be
halved and when you expect the backslashes to be kept.  The third gives a
result which is probably not what you want.  Avoid it.

                                *add-option-flags* *remove-option-flags*
                                *E539* *E550* *E551* *E552*
Some options are a list of flags.  When you want to add a flag to such an
option, without changing the existing ones, you can do it like this: >
   :set guioptions+=a
Remove a flag from an option like this: >
   :set guioptions-=a
This removes the 'a' flag from 'guioptions'.
Note that you should add or remove one flag at a time.  If 'guioptions' has
the value "ab", using "set guioptions-=ba" won't work, because the string "ba"
doesn't appear.

                            *:set_env* *expand-env* *expand-environment-var*
Environment variables in specific string options will be expanded.  If the
environment variable exists the '$' and the following environment variable
name is replaced with its value.  If it does not exist the '$' and the name
are not modified.  Any non-id character (not a letter, digit or '_') may
follow the environment variable name.  That character and what follows is
appended to the value of the environment variable.  Examples: >
   :set term=$TERM.new
   :set path=/usr/$INCLUDE,$HOME/include,.
When adding or removing a string from an option with ":set opt-=val" or ":set
opt+=val" the expansion is done before the adding or removing.


Handling of local options                       *local-options*

Some of the options only apply to a window or buffer.  Each window or buffer
has its own copy of this option, thus each can have its own value.  This
allows you to set 'list' in one window but not in another.  And set
'shiftwidth' to 3 in one buffer and 4 in another.

The following explains what happens to these local options in specific
situations.  You don't really need to know all of this, since Vim mostly uses
the option values you would expect.  Unfortunately, doing what the user
```

expects is a bit complicated...

When splitting a window, the local options are copied to the new window. Thus right after the split the contents of the two windows look the same.

When editing a new buffer, its local option values must be initialized. Since the local options of the current buffer might be specifically for that buffer, these are not used. Instead, for each buffer-local option there also is a global value, which is used for new buffers. With ":set" both the local and global value is changed. With "setlocal" only the local value is changed, thus this value is not used when editing a new buffer.

When editing a buffer that has been edited before, the options from the window that was last closed are used again. If this buffer has been edited in this window, the values from back then are used. Otherwise the values from the last closed window where the buffer was edited last are used.

It's possible to set a local window option specifically for a type of buffer. When you edit another buffer in the same window, you don't want to keep using these local window options. Therefore Vim keeps a global value of the local window options, which is used when editing another buffer. Each window has its own copy of these values. Thus these are local to the window, but global to all buffers in the window. With this you can do: >

```
:e one
:set list
:e two
```

Now the 'list' option will also be set in "two", since with the ":set list" command you have also set the global value. >

```
:set nolist
:e one
:setlocal list
:e two
```

Now the 'list' option is not set, because ":set nolist" resets the global value, ":setlocal list" only changes the local value and ":e two" gets the global value. Note that if you do this next: >

```
:e one
```

You will get back the 'list' value as it was the last time you edited "one". The options local to a window are remembered for each buffer. This also happens when the buffer is not loaded, but they are lost when the buffer is wiped out |:bwipe|.

:setl *:setlocal*

`:setl[ocal] ...` Like ":set" but set only the value local to the current buffer or window. Not all options have a local value. If the option does not have a local value the global value is set.
With the "all" argument: display local values for all local options.
Without argument: Display local values for all local options which are different from the default.
When displaying a specific local option, show the local value. For a global/local boolean option, when the global value is being used, "--" is displayed before the option name.
For a global option the global value is shown (but that might change in the future).
{not in Vi}

`:setl[ocal] {option}<` Set the local value of {option} to its global value by copying the value.
{not in Vi}

```
:se[t] {option}<        For |global-local| options: Remove the local value of
                        {option}, so that the global value will be used.
                        {not in Vi}

                                                        *:setg* *:setglobal*
:setg[lobal] ...        Like ":set" but set only the global value for a local
                        option without changing the local value.
                        When displaying an option, the global value is shown.
                        With the "all" argument: display global values for all
                        local options.
                        Without argument: display global values for all local
                        options which are different from the default.
                        {not in Vi}

For buffer-local and window-local options:
        Command          global value       local value ~
      :set option=value      set                set
 :setlocal option=value       -                 set
:setglobal option=value      set                 -
      :set option?            -                display
 :setlocal option?            -                display
:setglobal option?          display              -


Global options with a local value                       *global-local*

Options are global when you mostly use one value for all buffers and windows.
For some global options it's useful to sometimes have a different local value.
You can set the local value with ":setlocal".  That buffer or window will then
use the local value, while other buffers and windows continue using the global
value.

For example, you have two windows, both on C source code.  They use the global
'makeprg' option.  If you do this in one of the two windows: >
        :set makeprg=gmake
then the other window will switch to the same value.  There is no need to set
the 'makeprg' option in the other C source window too.
However, if you start editing a Perl file in a new window, you want to use
another 'makeprg' for it, without changing the value used for the C source
files.  You use this command: >
        :setlocal makeprg=perlmake
You can switch back to using the global value by making the local value empty: >
        :setlocal makeprg=
This only works for a string option.  For a boolean option you need to use the
"<" flag, like this: >
        :setlocal autoread<
Note that for non-boolean options using "<" copies the global value to the
local value, it doesn't switch back to using the global value (that matters
when the global value changes later).  You can also use: >
        :set path<
This will make the local value of 'path' empty, so that the global value is
used.  Thus it does the same as: >
        :setlocal path=
Note: In the future more global options can be made global-local.  Using
":setlocal" on a global option might work differently then.


Setting the filetype

:setf[iletype] [FALLBACK] {filetype}                    *:setf* *:setfiletype*
                        Set the 'filetype' option to {filetype}, but only if
                        not done yet in a sequence of (nested) autocommands.
```

```
                        This is short for: >
                                :if !did_filetype()
                                :  setlocal filetype={filetype}
                                :endif
<                       This command is used in a filetype.vim file to avoid
                        setting the 'filetype' option twice, causing different
                        settings and syntax files to be loaded.

                        When the optional FALLBACK argument is present, a
                        later :setfiletype command will override the
                        'filetype'.  This is to used for filetype detections
                        that are just a guess.  |did_filetype()| will return
                        false after this command.

                        {not in Vi}

                                *option-window* *optwin*
:bro[wse] se[t]                 *:set-browse* *:browse-set* *:opt* *:options*
:opt[ions]              Open a window for viewing and setting all options.
                        Options are grouped by function.
                        Offers short help for each option.  Hit <CR> on the
                        short help to open a help window with more help for
                        the option.
                        Modify the value of the option and hit <CR> on the
                        "set" line to set the new value.  For window and
                        buffer specific options, the last accessed window is
                        used to set the option value in, unless this is a help
                        window, in which case the window below help window is
                        used (skipping the option-window).
                        {not available when compiled without the |+eval| or
                        |+autocmd| features}

                                                                *$HOME*
Using "~" is like using "$HOME", but it is only recognized at the start of an
option and after a space or comma.

On Unix systems "~user" can be used too.  It is replaced by the home directory
of user "user".  Example: >
    :set path=~mool/include,/usr/include,.

On Unix systems the form "${HOME}" can be used too.  The name between {} can
contain non-id characters then.  Note that if you want to use this for the
"gf" command, you need to add the '{' and '}' characters to 'isfname'.

NOTE: expanding environment variables and "~/" is only done with the ":set"
command, not when assigning a value to an option with ":let".

                                                        *$HOME-windows*
On MS-Windows, if $HOME is not defined as an environment variable, then
at runtime Vim will set it to the expansion of $HOMEDRIVE$HOMEPATH.
If $HOMEDRIVE is not set then $USERPROFILE is used.

This expanded value is not exported to the environment, this matters when
running an external command: >
        :echo system('set | findstr ^HOME=')
and >
        :echo luaeval('os.getenv("HOME")')
should echo nothing (an empty string) despite exists('$HOME') being true.
When setting $HOME to a non-empty string it will be exported to the
subprocesses.
```

```
Note the maximum length of an expanded option is limited.  How much depends on
the system, mostly it is something like 256 or 1024 characters.

                                                        *:fix* *:fixdel*
:fix[del]               Set the value of 't_kD':
                                't_kb' is     't_kD' becomes    ~
                                  CTRL-?        CTRL-H
                                not CTRL-?      CTRL-?

                        (CTRL-? is 0177 octal, 0x7f hex) {not in Vi}

                        If your delete key terminal code is wrong, but the
                        code for backspace is alright, you can put this in
                        your .vimrc: >
                                :fixdel
<                       This works no matter what the actual code for
                        backspace is.

                        If the backspace key terminal code is wrong you can
                        use this: >
                                :if &term == "termname"
                                :  set t_kb=^V<BS>
                                :  fixdel
                                :endif
<                       Where "^V" is CTRL-V and "<BS>" is the backspace key
                        (don't type four characters!).  Replace "termname"
                        with your terminal name.

                        If your <Delete> key sends a strange key sequence (not
                        CTRL-? or CTRL-H) you cannot use ":fixdel".  Then use: >
                                :if &term == "termname"
                                :  set t_kD=^V<Delete>
                                :endif
<                       Where "^V" is CTRL-V and "<Delete>" is the delete key
                        (don't type eight characters!).  Replace "termname"
                        with your terminal name.

                                                        *Linux-backspace*
                        Note about Linux: By default the backspace key
                        produces CTRL-?, which is wrong.  You can fix it by
                        putting this line in your rc.local: >
                                echo "keycode 14 = BackSpace" | loadkeys
<
                                                        *NetBSD-backspace*
                        Note about NetBSD: If your backspace doesn't produce
                        the right code, try this: >
                                xmodmap -e "keycode 22 = BackSpace"
<                       If this works, add this in your .Xmodmap file: >
                                keysym 22 = BackSpace
<                       You need to restart for this to take effect.

==============================================================================
2. Automatically setting options                        *auto-setting*

Besides changing options with the ":set" command, there are three alternatives
to set options automatically for one or more files:

1. When starting Vim initializations are read from various places.  See
   |initialization|.  Most of them are performed for all editing sessions,
   and some of them depend on the directory where Vim is started.
   You can create an initialization file with |:mkvimrc|, |:mkview| and
   |:mksession|.
```

2. If you start editing a new file, the automatic commands are executed.
   This can be used to set options for files matching a particular pattern and
   many other things. See |autocommand|.
3. If you start editing a new file, and the 'modeline' option is on, a
   number of lines at the beginning and end of the file are checked for
   modelines. This is explained here.

modeline *vim:* *vi:* *ex:* *E520*

There are two forms of modelines. The first form:

```
        [text]{white}{vi:|vim:|ex:}[white]{options}
```

[text]	any text or empty
{white}	at least one blank character (<Space> or <Tab>)
{vi:|vim:|ex:}	the string "vi:", "vim:" or "ex:"
[white]	optional white space
{options}	a list of option settings, separated with white space or ':', where each part between ':' is the argument for a ":set" command (can be empty)

Examples:

```
   vi:noai:sw=3 ts=6 ~
   vim: tw=77 ~
```

The second form (this is compatible with some versions of Vi):

```
        [text]{white}{vi:|vim:|Vim:|ex:}[white]se[t] {options}:[text]
```

[text]	any text or empty
{white}	at least one blank character (<Space> or <Tab>)
{vi:|vim:|Vim:|ex:}	the string "vi:", "vim:", "Vim:" or "ex:"
[white]	optional white space
se[t]	the string "set " or "se " (note the space); When "Vim" is used it must be "set".
{options}	a list of options, separated with white space, which is the argument for a ":set" command
:	a colon
[text]	any text or empty

Examples:

```
   /* vim: set ai tw=75: */ ~
   /* Vim: set ai tw=75: */ ~
```

The white space before {vi:|vim:|Vim:|ex:} is required. This minimizes the
chance that a normal word like "lex:" is caught. There is one exception:
"vi:" and "vim:" can also be at the start of the line (for compatibility with
version 3.0). Using "ex:" at the start of the line will be ignored (this
could be short for "example:").

modeline-local

The options are set like with ":setlocal": The new value only applies to the
buffer and window that contain the file. Although it's possible to set global
options from a modeline, this is unusual. If you have two windows open and
the files in it set the same global option to a different value, the result
depends on which one was opened last.

When editing a file that was already loaded, only the window-local options
from the modeline are used. Thus if you manually changed a buffer-local
option after opening the file, it won't be changed if you edit the same buffer
in another window. But window-local options will be set.

modeline-version

If the modeline is only to be used for some versions of Vim, the version

```
number can be specified where "vim:" or "Vim:" is used:
        vim{vers}:      version {vers} or later
        vim<{vers}:     version before {vers}
        vim={vers}:     version {vers}
        vim>{vers}:     version after {vers}
{vers} is 700 for Vim 7.0 (hundred times the major version plus minor).
For example, to use a modeline only for Vim 7.0:
        /* vim700: set foldmethod=marker */ ~
To use a modeline for Vim after version 7.2:
        /* vim>702: set cole=2: */ ~
There can be no blanks between "vim" and the ":".


The number of lines that are checked can be set with the 'modelines' option.
If 'modeline' is off or 'modelines' is 0 no lines are checked.

Note that for the first form all of the rest of the line is used, thus a line
like:
   /* vi:ts=4: */ ~
will give an error message for the trailing "*/".  This line is OK:
   /* vi:set ts=4: */ ~

If an error is detected the rest of the line is skipped.

If you want to include a ':' in a set command precede it with a '\'.  The
backslash in front of the ':' will be removed.  Example:
   /* vi:set dir=c\:\tmp: */ ~
This sets the 'dir' option to "c:\tmp".  Only a single backslash before the
':' is removed.  Thus to include "\:" you have to specify "\\:".

No other commands than "set" are supported, for security reasons (somebody
might create a Trojan horse text file with modelines).  And not all options
can be set.  For some options a flag is set, so that when it's used the
|sandbox| is effective.  Still, there is always a small risk that a modeline
causes trouble.  E.g., when some joker sets 'textwidth' to 5 all your lines
are wrapped unexpectedly.  So disable modelines before editing untrusted text.
The mail ftplugin does this, for example.

Hint: If you would like to do something else than setting an option, you could
define an autocommand that checks the file for a specific string.  For
example: >
        au BufReadPost * if getline(1) =~ "VAR" | call SetVar() | endif
And define a function SetVar() that does something with the line containing
"VAR".

==============================================================================
3. Options summary                                      *option-summary*

In the list below all the options are mentioned with their full name and with
an abbreviation if there is one.  Both forms may be used.

In this document when a boolean option is "set" that means that ":set option"
is entered.  When an option is "reset", ":set nooption" is used.

For some options there are two default values: The "Vim default", which is
used when 'compatible' is not set, and the "Vi default", which is used when
'compatible' is set.

Most options are the same in all windows and buffers.  There are a few that
are specific to how the text is presented in a window.  These can be set to a
different value in each window.  For example the 'list' option can be set in
one window and reset in another for the same text, giving both types of view
```

at the same time. There are a few options that are specific to a certain file. These can have a different value for each file or buffer. For example the 'textwidth' option can be 78 for a normal text file and 0 for a C program.

global	one option for all buffers and windows
local to window	each window has its own copy of this option
local to buffer	each buffer has its own copy of this option

When creating a new window the option values from the currently active window are used as a default value for the window-specific options. For the buffer-specific options this depends on the 's' and 'S' flags in the 'cpoptions' option. If 's' is included (which is the default) the values for buffer options are copied from the currently active buffer when a buffer is first entered. If 'S' is present the options are copied each time the buffer is entered, this is almost like having global options. If 's' and 'S' are not present, the options are copied from the currently active buffer when the buffer is created.

Hidden options *hidden-options*

Not all options are supported in all versions. This depends on the supported features and sometimes on the system. A remark about this is in curly braces below. When an option is not supported it may still be set without getting an error, this is called a hidden option. You can't get the value of a hidden option though, it is not stored.

To test if option "foo" can be used with ":set" use something like this: >
	if exists('&foo')
This also returns true for a hidden option. To test if option "foo" is really supported use something like this: >
	if exists('+foo')
<
E355
A jump table for the options with a short description can be found at |Q_op|.

'aleph' *'al'* *aleph* *Aleph*
'aleph' 'al' number (default 128 for MS-DOS, 224 otherwise)
global
{not in Vi}
{only available when compiled with the |+rightleft| feature}
The ASCII code for the first letter of the Hebrew alphabet. The routine that maps the keyboard in Hebrew mode, both in Insert mode (when hkmap is set) and on the command-line (when hitting CTRL-_) outputs the Hebrew characters in the range [aleph..aleph+26]. aleph=128 applies to PC code, and aleph=224 applies to ISO 8859-8. See |rileft.txt|.

'allowrevins' *'ari'* *'noallowrevins'* *'noari'*
'allowrevins' 'ari' boolean (default off)
global
{not in Vi}
{only available when compiled with the |+rightleft| feature}
Allow CTRL-_ in Insert and Command-line mode. This is default off, to avoid that users that accidentally type CTRL-_ instead of SHIFT-_ get into reverse Insert mode, and don't know how to get out. See 'revins'.
NOTE: This option is reset when 'compatible' is set.

'altkeymap' *'akm'* *'noaltkeymap'* *'noakm'*

```
'altkeymap' 'akm'       boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+farsi|
                        feature}
        When on, the second language is Farsi.  In editing mode CTRL-_ toggles
        the keyboard map between Farsi and English, when 'allowrevins' set.

        When off, the keyboard map toggles between Hebrew and English.  This
        is useful to start the Vim in native mode i.e. English (left-to-right
        mode) and have default second language Farsi or Hebrew (right-to-left
        mode).  See |farsi.txt|.

                                                *'ambiwidth'* *'ambw'*
'ambiwidth' 'ambw'      string (default: "single")
                        global
                        {not in Vi}
                        {only available when compiled with the |+multi_byte|
                        feature}
        Only effective when 'encoding' is "utf-8" or another Unicode encoding.
        Tells Vim what to do with characters with East Asian Width Class
        Ambiguous (such as Euro, Registered Sign, Copyright Sign, Greek
        letters, Cyrillic letters).

        There are currently two possible values:
        "single":       Use the same width as characters in US-ASCII.  This is
                        expected by most users.
        "double":       Use twice the width of ASCII characters.
                                                        *E834* *E835*
        The value "double" cannot be used if 'listchars' or 'fillchars'
        contains a character that would be double width.

        There are a number of CJK fonts for which the width of glyphs for
        those characters are solely based on how many octets they take in
        legacy/traditional CJK encodings.  In those encodings, Euro,
        Registered sign, Greek/Cyrillic letters are represented by two octets,
        therefore those fonts have "wide" glyphs for them.  This is also
        true of some line drawing characters used to make tables in text
        file.  Therefore, when a CJK font is used for GUI Vim or
        Vim is running inside a terminal (emulators) that uses a CJK font
        (or Vim is run inside an xterm invoked with "-cjkwidth" option.),
        this option should be set to "double" to match the width perceived
        by Vim with the width of glyphs in the font.  Perhaps it also has
        to be set to "double" under CJK Windows 9x/ME or Windows 2k/XP
        when the system locale is set to one of CJK locales.  See Unicode
        Standard Annex #11 (http://www.unicode.org/reports/tr11).

        Vim may set this option automatically at startup time when Vim is
        compiled with the |+termresponse| feature and if |t_u7| is set to the
        escape sequence to request cursor position report.  The response can
        be found in |v:termu7resp|.

                        *'antialias'* *'anti'* *'noantialias'* *'noanti'*
'antialias' 'anti'      boolean (default: off)
                        global
                        {not in Vi}
                        {only available when compiled with GUI enabled
                        on Mac OS X}
        This option only has an effect in the GUI version of Vim on Mac OS X
        v10.2 or later.  When on, Vim will use smooth ("antialiased") fonts,
        which can be easier to read at certain sizes on certain displays.
        Setting this option can sometimes cause problems if 'guifont' is set
```

```
        to its default (empty string).
        NOTE: This option is reset when 'compatible' is set.

                        *'autochdir'* *'acd'* *'noautochdir'* *'noacd'*
'autochdir' 'acd'       boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with it, use
                        exists("+autochdir") to check}
        When on, Vim will change the current working directory whenever you
        open a file, switch buffers, delete a buffer or open/close a window.
        It will change to the directory containing the file which was opened
        or selected.
        Note: When this option is on some plugins may not work.

                                *'arabic'* *'arab'* *'noarabic'* *'noarab'*
'arabic' 'arab'         boolean (default off)
                        local to window
                        {not in Vi}
                        {only available when compiled with the |+arabic|
                        feature}
        This option can be set to start editing Arabic text.
        Setting this option will:
        - Set the 'rightleft' option, unless 'termbidi' is set.
        - Set the 'arabicshape' option, unless 'termbidi' is set.
        - Set the 'keymap' option to "arabic"; in Insert mode CTRL-^ toggles
          between typing English and Arabic key mapping.
        - Set the 'delcombine' option
        Note that 'encoding' must be "utf-8" for working with Arabic text.

        Resetting this option will:
        - Reset the 'rightleft' option.
        - Disable the use of 'keymap' (without changing its value).
        Note that 'arabicshape' and 'delcombine' are not reset (it is a global
        option).
        NOTE: This option is reset when 'compatible' is set.
        Also see |arabic.txt|.

                                          *'arabicshape'* *'arshape'*
                                          *'noarabicshape'* *'noarshape'*
'arabicshape' 'arshape' boolean (default on)
                        global
                        {not in Vi}
                        {only available when compiled with the |+arabic|
                        feature}
        When on and 'termbidi' is off, the required visual character
        corrections that need to take place for displaying the Arabic language
        take effect.  Shaping, in essence, gets enabled; the term is a broad
        one which encompasses:
          a) the changing/morphing of characters based on their location
             within a word (initial, medial, final and stand-alone).
          b) the enabling of the ability to compose characters
          c) the enabling of the required combining of some characters
        When disabled the display shows each character's true stand-alone
        form.
        Arabic is a complex language which requires other settings, for
        further details see |arabic.txt|.
        NOTE: This option is set when 'compatible' is set.

                        *'autoindent'* *'ai'* *'noautoindent'* *'noai'*
'autoindent' 'ai'       boolean (default off)
                        local to buffer
```

```
        Copy indent from current line when starting a new line (typing <CR>
        in Insert mode or when using the "o" or "O" command).  If you do not
        type anything on the new line except <BS> or CTRL-D and then type
        <Esc>, CTRL-O or <CR>, the indent is deleted again.  Moving the cursor
        to another line has the same effect, unless the 'I' flag is included
        in 'cpoptions'.
        When autoindent is on, formatting (with the "gq" command or when you
        reach 'textwidth' in Insert mode) uses the indentation of the first
        line.
        When 'smartindent' or 'cindent' is on the indent is changed in
        a different way.
        The 'autoindent' option is reset when the 'paste' option is set and
        restored when 'paste' is reset.
        {small difference from Vi: After the indent is deleted when typing
        <Esc> or <CR>, the cursor position when moving up or down is after the
        deleted indent; Vi puts the cursor somewhere in the deleted indent}.

                                    *'autoread'* *'ar'* *'noautoread'* *'noar'*
'autoread' 'ar'         boolean (default off)
                        global or local to buffer |global-local|
                        {not in Vi}
        When a file has been detected to have been changed outside of Vim and
        it has not been changed inside of Vim, automatically read it again.
        When the file has been deleted this is not done.  |timestamp|
        If this option has a local value, use this command to switch back to
        using the global value: >
                :set autoread<
<
                                    *'autowrite'* *'aw'* *'noautowrite'* *'noaw'*
'autowrite' 'aw'        boolean (default off)
                        global
        Write the contents of the file, if it has been modified, on each
        :next, :rewind, :last, :first, :previous, :stop, :suspend, :tag, :!,
        :make, CTRL-] and CTRL-^ command; and when a :buffer, CTRL-O, CTRL-I,
        '{A-Z0-9}, or `{A-Z0-9} command takes one to another file.
        Note that for some commands the 'autowrite' option is not used, see
        'autowriteall' for that.

                         *'autowriteall'* *'awa'* *'noautowriteall'* *'noawa'*
'autowriteall' 'awa'    boolean (default off)
                        global
                        {not in Vi}
        Like 'autowrite', but also used for commands ":edit", ":enew", ":quit",
        ":qall", ":exit", ":xit", ":recover" and closing the Vim window.
        Setting this option also implies that Vim behaves like 'autowrite' has
        been set.

                                                        *'background'* *'bg'*
'background' 'bg'       string  (default "dark" or "light", see below)
                        global
                        {not in Vi}
        When set to "dark", Vim will try to use colors that look good on a
        dark background.  When set to "light", Vim will try to use colors that
        look good on a light background.  Any other value is illegal.
        Vim tries to set the default value according to the terminal used.
        This will not always be correct.
        Setting this option does not change the background color, it tells Vim
        what the background color looks like.  For changing the background
        color, see |:hi-normal|.

        When 'background' is set Vim will adjust the default color groups for
        the new value.  But the colors used for syntax highlighting will not
```

```
        change.                                     *g:colors_name*
        When a color scheme is loaded (the "g:colors_name" variable is set)
        setting 'background' will cause the color scheme to be reloaded.  If
        the color scheme adjusts to the value of 'background' this will work.
        However, if the color scheme sets 'background' itself the effect may
        be undone.  First delete the "g:colors_name" variable when needed.

        When setting 'background' to the default value with: >
                :set background&
<       Vim will guess the value.  In the GUI this should work correctly,
        in other cases Vim might not be able to guess the right value.

        When the |t_RB| option is set, Vim will use it to request the background
        color from the terminal.  If the returned RGB value is dark/light and
        'background' is not dark/light, 'background' will be set and the
        screen is redrawn.  This may have side effects, make t_BG empty in
        your .vimrc if you suspect this problem.  The response to |t_RB| can
        be found in |v:termrgbresp|.

        When starting the GUI, the default value for 'background' will be
        "light".  When the value is not set in the .gvimrc, and Vim detects
        that the background is actually quite dark, 'background' is set to
        "dark".  But this happens only AFTER the .gvimrc file has been read
        (because the window needs to be opened to find the actual background
        color).  To get around this, force the GUI window to be opened by
        putting a ":gui" command in the .gvimrc file, before where the value
        of 'background' is used (e.g., before ":syntax on").

        For MS-DOS, Windows and OS/2 the default is "dark".
        For other systems "dark" is used when 'term' is "linux",
        "screen.linux", "cygwin" or "putty", or $COLORFGBG suggests a dark
        background.  Otherwise the default is "light".

        The |:terminal| command and the |term_start()| function use the
        'background' value to decide whether the terminal window will start
        with a white or black background.

        Normally this option would be set in the .vimrc file.  Possibly
        depending on the terminal name.  Example: >
                :if &term == "pcterm"
                :  set background=dark
                :endif
<       When this option is set, the default settings for the highlight groups
        will change.  To use other settings, place ":highlight" commands AFTER
        the setting of the 'background' option.
        This option is also used in the "$VIMRUNTIME/syntax/syntax.vim" file
        to select the colors for syntax highlighting.  After changing this
        option, you must load syntax.vim again to see the result.  This can be
        done with ":syntax on".

                                                        *'backspace'* *'bs'*
'backspace' 'bs'        string  (default "", set to "indent,eol,start"
                                                             in |defaults.vim|)
                        global
                        {not in Vi}
        Influences the working of <BS>, <Del>, CTRL-W and CTRL-U in Insert
        mode.  This is a list of items, separated by commas.  Each item allows
        a way to backspace over something:
        value   effect  ~
        indent  allow backspacing over autoindent
        eol     allow backspacing over line breaks (join lines)
        start   allow backspacing over the start of insert; CTRL-W and CTRL-U
```

```
                stop once at the start of insert.

        When the value is empty, Vi compatible backspacing is used.

        For backwards compatibility with version 5.4 and earlier:
        value   effect  ~
          0     same as ":set backspace=" (Vi compatible)
          1     same as ":set backspace=indent,eol"
          2     same as ":set backspace=indent,eol,start"

        See |:fixdel| if your <BS> or <Del> key does not do what you want.
        NOTE: This option is set to "" when 'compatible' is set.

                                *'backup'* *'bk'* *'nobackup'* *'nobk'*
'backup' 'bk'           boolean (default off)
                        global
                        {not in Vi}
        Make a backup before overwriting a file.  Leave it around after the
        file has been successfully written.  If you do not want to keep the
        backup file, but you do want a backup while the file is being
        written, reset this option and set the 'writebackup' option (this is
        the default).  If you do not want a backup file at all reset both
        options (use this if your file system is almost full).  See the
        |backup-table| for more explanations.
        When the 'backupskip' pattern matches, a backup is not made anyway.
        When 'patchmode' is set, the backup may be renamed to become the
        oldest version of a file.
        NOTE: This option is reset when 'compatible' is set.

                                                *'backupcopy'* *'bkc'*
'backupcopy' 'bkc'      string  (Vi default for Unix: "yes", otherwise: "auto")
                        global or local to buffer |global-local|
                        {not in Vi}
        When writing a file and a backup is made, this option tells how it's
        done.  This is a comma separated list of words.

        The main values are:
        "yes"   make a copy of the file and overwrite the original one
        "no"    rename the file and write a new one
        "auto"  one of the previous, what works best

        Extra values that can be combined with the ones above are:
        "breaksymlink"  always break symlinks when writing
        "breakhardlink" always break hardlinks when writing

        Making a copy and overwriting the original file:
        - Takes extra time to copy the file.
        + When the file has special attributes, is a (hard/symbolic) link or
          has a resource fork, all this is preserved.
        - When the file is a link the backup will have the name of the link,
          not of the real file.

        Renaming the file and writing a new one:
        + It's fast.
        - Sometimes not all attributes of the file can be copied to the new
          file.
        - When the file is a link the new file will not be a link.

        The "auto" value is the middle way: When Vim sees that renaming file
        is possible without side effects (the attributes can be passed on and
        the file is not a link) that is used.  When problems are expected, a
        copy will be made.
```

The "breaksymlink" and "breakhardlink" values can be used in
combination with any of "yes", "no" and "auto". When included, they
force Vim to always break either symbolic or hard links by doing
exactly what the "no" option does, renaming the original file to
become the backup and writing a new file in its place. This can be
useful for example in source trees where all the files are symbolic or
hard links and any changes should stay in the local source tree, not
be propagated back to the original source.
crontab
One situation where "no" and "auto" will cause problems: A program
that opens a file, invokes Vim to edit that file, and then tests if
the open file was changed (through the file descriptor) will check the
backup file instead of the newly created file. "crontab -e" is an
example.

When a copy is made, the original file is truncated and then filled
with the new text. This means that protection bits, owner and
symbolic links of the original file are unmodified. The backup file
however, is a new file, owned by the user who edited the file. The
group of the backup is set to the group of the original file. If this
fails, the protection bits for the group are made the same as for
others.

When the file is renamed this is the other way around: The backup has
the same attributes of the original file, and the newly written file
is owned by the current user. When the file was a (hard/symbolic)
link, the new file will not! That's why the "auto" value doesn't
rename when the file is a link. The owner and group of the newly
written file will be set to the same ones as the original file, but
the system may refuse to do this. In that case the "auto" value will
again not rename the file.

NOTE: This option is set to the Vi default value when 'compatible' is
set and to the Vim default value when 'compatible' is reset.

'backupdir' *'bdir'*
'backupdir' 'bdir' string (default for Amiga: ".,t:",
for MS-DOS and Win32: ".,$TEMP,c:/tmp,c:/temp"
for Unix: ".,~/tmp,~/")
global
{not in Vi}
List of directories for the backup file, separated with commas.
- The backup file will be created in the first directory in the list
where this is possible. The directory must exist, Vim will not
create it for you.
- Empty means that no backup file will be created ('patchmode' is
impossible!). Writing may fail because of this.
- A directory "." means to put the backup file in the same directory
as the edited file.
- A directory starting with "./" (or ".\" for MS-DOS et al.) means to
put the backup file relative to where the edited file is. The
leading "." is replaced with the path name of the edited file.
("." inside a directory name has no special meaning).
- Spaces after the comma are ignored, other spaces are considered part
of the directory name. To have a space at the start of a directory
name, precede it with a backslash.
- To include a comma in a directory name precede it with a backslash.
- A directory name may end in an '/'.
- Environment variables are expanded |:set_env|.
- Careful with '\' characters, type one before a space, type two to
get one in the option (see |option-backslash|), for example: >

```
            :set bdir=c:\\tmp,\ dir\\,with\\,commas,\\\ dir\ with\ spaces
<       - For backwards compatibility with Vim version 3.0 a '>' at the start
          of the option is removed.
        See also 'backup' and 'writebackup' options.
        If you want to hide your backup files on Unix, consider this value: >
                :set backupdir=./.backup,~/.backup,.,/tmp
<       You must create a ".backup" directory in each directory and in your
        home directory for this to work properly.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        directories from the list.  This avoids problems when a future version
        uses another default.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'backupext'* *'bex'* *E589*
'backupext' 'bex'       string  (default "~", for VMS: "_")
                        global
                        {not in Vi}
        String which is appended to a file name to make the name of the
        backup file.  The default is quite unusual, because this avoids
        accidentally overwriting existing files with a backup file.  You might
        prefer using ".bak", but make sure that you don't have files with
        ".bak" that you want to keep.
        Only normal file name characters can be used, "/\*?[|<>" are illegal.

        If you like to keep a lot of backups, you could use a BufWritePre
        autocommand to change 'backupext' just before writing the file to
        include a timestamp. >
                :au BufWritePre * let &bex = '-' . strftime("%Y%b%d%X") . '~'
<       Use 'backupdir' to put the backup in a different directory.

                                                *'backupskip'* *'bsk'*
'backupskip' 'bsk'      string  (default: "/tmp/*,$TMPDIR/*,$TMP/*,$TEMP/*")
                        global
                        {not in Vi}
                        {not available when compiled without the |+wildignore|
                        feature}
        A list of file patterns.  When one of the patterns matches with the
        name of the file which is written, no backup file is created.  Both
        the specified file name and the full path name of the file are used.
        The pattern is used like with |:autocmd|, see |autocmd-patterns|.
        Watch out for special characters, see |option-backslash|.
        When $TMPDIR, $TMP or $TEMP is not defined, it is not used for the
        default value.  "/tmp/*" is only used for Unix.

        WARNING: Not having a backup file means that when Vim fails to write
        your buffer correctly and then, for whatever reason, Vim exits, you
        lose both the original file and what you were writing.  Only disable
        backups if you don't care about losing the file.

        Note that environment variables are not expanded.  If you want to use
        $HOME you must expand it explicitly, e.g.: >
                :let &backupskip = escape(expand('$HOME'), '\') . '/tmp/*'

<       Note that the default also makes sure that "crontab -e" works (when a
        backup would be made by renaming the original file crontab won't see
        the newly created file).  Also see 'backupcopy' and |crontab|.

                                                *'balloondelay'* *'bdlay'*
'balloondelay' 'bdlay'  number  (default: 600)
                        global
                        {not in Vi}
```

```
                        {only available when compiled with the |+balloon_eval|
                        feature}
        Delay in milliseconds before a balloon may pop up.  See |balloon-eval|.

                       *'ballooneval'* *'beval'* *'noballooneval'* *'nobeval'*
'ballooneval' 'beval'   boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+balloon_eval|
                        feature}
        Switch on the |balloon-eval| functionality.

                                                      *'balloonexpr'* *'bexpr'*
'balloonexpr' 'bexpr'   string  (default "")
                        global or local to buffer |global-local|
                        {not in Vi}
                        {only available when compiled with the |+balloon_eval|
                        feature}
        Expression for text to show in evaluation balloon.  It is only used
        when 'ballooneval' is on.  These variables can be used:

        v:beval_bufnr   number of the buffer in which balloon is going to show
        v:beval_winnr   number of the window
        v:beval_winid   ID of the window
        v:beval_lnum    line number
        v:beval_col     column number (byte index)
        v:beval_text    word under or after the mouse pointer

        The evaluation of the expression must not have side effects!
        Example: >
    function! MyBalloonExpr()
        return 'Cursor is at line ' . v:beval_lnum .
               \', column ' . v:beval_col .
               \ ' of file ' .  bufname(v:beval_bufnr) .
               \ ' on word "' . v:beval_text . '"'
    endfunction
    set bexpr=MyBalloonExpr()
    set ballooneval
<
        Also see |balloon_show()|, can be used if the content of the balloon
        is to be fetched asynchronously.

        NOTE: The balloon is displayed only if the cursor is on a text
        character.  If the result of evaluating 'balloonexpr' is not empty,
        Vim does not try to send a message to an external debugger (Netbeans
        or Sun Workshop).

        The expression will be evaluated in the |sandbox| when set from a
        modeline, see |sandbox-option|.

        It is not allowed to change text or jump to another window while
        evaluating 'balloonexpr' |textlock|.

        To check whether line breaks in the balloon text work use this check: >
                if has("balloon_multiline")
<       When they are supported "\n" characters will start a new line.  If the
        expression evaluates to a |List| this is equal to using each List item
        as a string and putting "\n" in between them.
        NOTE: This option is set to "" when 'compatible' is set.

                                                  *'belloff'* *'bo'*
'belloff' 'bo'          string  (default "")
```

```
                        global
                        {not in Vi}
        Specifies for which events the bell will not be rung. It is a comma
        separated list of items. For each item that is present, the bell
        will be silenced. This is most useful to specify specific events in
        insert mode to be silenced.

        item        meaning when present       ~
        all         All events.
        backspace   When hitting <BS> or <Del> and deleting results in an
                    error.
        cursor      Fail to move around using the cursor keys or
                    <PageUp>/<PageDown> in |Insert-mode|.
        complete    Error occurred when using |i_CTRL-X_CTRL-K| or
                    |i_CTRL-X_CTRL-T|.
        copy        Cannot copy char from insert mode using |i_CTRL-Y| or
                    |i_CTRL-E|.
        ctrlg       Unknown Char after <C-G> in Insert mode.
        error       Other Error occurred (e.g. try to join last line)
                    (mostly used in |Normal-mode| or |Cmdline-mode|).
        esc         hitting <Esc> in |Normal-mode|.
        ex          In |Visual-mode|, hitting |Q| results in an error.
        hangul      Error occurred when using hangul input.
        insertmode  Pressing <Esc> in 'insertmode'.
        lang        Calling the beep module for Lua/Mzscheme/TCL.
        mess        No output available for |g<|.
        showmatch   Error occurred for 'showmatch' function.
        operator    Empty region error |cpo-E|.
        register    Unknown register after <C-R> in |Insert-mode|.
        shell       Bell from shell output |:!|.
        spell       Error happened on spell suggest.
        wildmode    More matches in |cmdline-completion| available
                    (depends on the 'wildmode' setting).

        This is most useful to fine tune when in Insert mode the bell should
        be rung. For Normal mode and Ex commands, the bell is often rung to
        indicate that an error occurred. It can be silenced by adding the
        "error" keyword.

                                        *'binary'* *'bin'* *'nobinary'* *'nobin'*
'binary' 'bin'          boolean (default off)
                        local to buffer
                        {not in Vi}
        This option should be set before editing a binary file.  You can also
        use the |-b| Vim argument.  When this option is switched on a few
        options will be changed (also when it already was on):
                'textwidth'  will be set to 0
                'wrapmargin' will be set to 0
                'modeline'   will be off
                'expandtab'  will be off
        Also, 'fileformat' and 'fileformats' options will not be used, the
        file is read and written like 'fileformat' was "unix" (a single <NL>
        separates lines).
        The 'fileencoding' and 'fileencodings' options will not be used, the
        file is read without conversion.
        NOTE: When you start editing a(nother) file while the 'bin' option is
        on, settings from autocommands may change the settings again (e.g.,
        'textwidth'), causing trouble when editing.  You might want to set
        'bin' again when the file has been loaded.
        The previous values of these options are remembered and restored when
        'bin' is switched from on to off.  Each buffer has its own set of
        saved option values.
```

```
        To edit a file with 'binary' set you can use the |++bin| argument.
        This avoids you have to do ":set bin", which would have effect for all
        files you edit.
        When writing a file the <EOL> for the last line is only written if
        there was one in the original file (normally Vim appends an <EOL> to
        the last line if there is none; this would make the file longer).  See
        the 'endofline' option.

                              *'bioskey'* *'biosk'* *'nobioskey'* *'nobiosk'*
'bioskey' 'biosk'             boolean (default on)
                              global
                              {not in Vi}  {only for MS-DOS}
        This was for MS-DOS and is no longer supported.

                                                              *'bomb'* *'nobomb'*
'bomb'                        boolean (default off)
                              local to buffer
                              {not in Vi}
                              {only available when compiled with the |+multi_byte|
                              feature}
        When writing a file and the following conditions are met, a BOM (Byte
        Order Mark) is prepended to the file:
        - this option is on
        - the 'binary' option is off
        - 'fileencoding' is "utf-8", "ucs-2", "ucs-4" or one of the little/big
          endian variants.
        Some applications use the BOM to recognize the encoding of the file.
        Often used for UCS-2 files on MS-Windows.  For other applications it
        causes trouble, for example: "cat file1 file2" makes the BOM of file2
        appear halfway the resulting file.  Gcc doesn't accept a BOM.
        When Vim reads a file and 'fileencodings' starts with "ucs-bom", a
        check for the presence of the BOM is done and 'bomb' set accordingly.
        Unless 'binary' is set, it is removed from the first line, so that you
        don't see it when editing.  When you don't change the options, the BOM
        will be restored when writing the file.

                                                    *'breakat'* *'brk'*
'breakat' 'brk'               string  (default " ^I!@*-+;:,./?")
                              global
                              {not in Vi}
                              {not available when compiled without the |+linebreak|
                              feature}
        This option lets you choose which characters might cause a line
        break if 'linebreak' is on.  Only works for ASCII and also for 8-bit
        characters when 'encoding' is an 8-bit encoding.

                              *'breakindent'* *'bri'* *'nobreakindent'* *'nobri'*
'breakindent' 'bri'           boolean (default off)
                              local to window
                              {not in Vi}
                              {not available when compiled without the |+linebreak|
                              feature}
        Every wrapped line will continue visually indented (same amount of
        space as the beginning of that line), thus preserving horizontal blocks
        of text.
        NOTE: This option is reset when 'compatible' is set.

                                                    *'breakindentopt'* *'briopt'*
'breakindentopt' 'briopt' string (default empty)
                              local to window
                              {not in Vi}
                              {not available when compiled without the |+linebreak|
```

```
                        feature}
        Settings for 'breakindent'. It can consist of the following optional
        items and must be separated by a comma:
                min:{n}     Minimum text width that will be kept after
                            applying 'breakindent', even if the resulting
                            text should normally be narrower. This prevents
                            text indented almost to the right window border
                            occupying lot of vertical space when broken.
                shift:{n}   After applying 'breakindent', the wrapped line's
                            beginning will be shifted by the given number of
                            characters.  It permits dynamic French paragraph
                            indentation (negative) or emphasizing the line
                            continuation (positive).
                sbr         Display the 'showbreak' value before applying the
                            additional indent.
        The default value for min is 20 and shift is 0.

                                                    *'browsedir'* *'bsdir'*
'browsedir' 'bsdir'     string  (default: "last")
                        global
                        {not in Vi} {only for Motif, Athena, GTK, Mac and
                        Win32 GUI}
        Which directory to use for the file browser:
           last         Use same directory as with last file browser, where a
                        file was opened or saved.
           buffer       Use the directory of the related buffer.
           current      Use the current directory.
           {path}       Use the specified directory

                                                    *'bufhidden'* *'bh'*
'bufhidden' 'bh'        string (default: "")
                        local to buffer
                        {not in Vi}
        This option specifies what happens when a buffer is no longer
        displayed in a window:
          <empty>       follow the global 'hidden' option
          hide          hide the buffer (don't unload it), also when 'hidden'
                        is not set
          unload        unload the buffer, also when 'hidden' is set or using
                        |:hide|
          delete        delete the buffer from the buffer list, also when
                        'hidden' is set or using |:hide|, like using
                        |:bdelete|
          wipe          wipe out the buffer from the buffer list, also when
                        'hidden' is set or using |:hide|, like using
                        |:bwipeout|

        CAREFUL: when "unload", "delete" or "wipe" is used changes in a buffer
        are lost without a warning.  Also, these values may break autocommands
        that switch between buffers temporarily.
        This option is used together with 'buftype' and 'swapfile' to specify
        special kinds of buffers.   See |special-buffers|.

                        *'buflisted'* *'bl'* *'nobuflisted'* *'nobl'* *E85*
'buflisted' 'bl'        boolean (default: on)
                        local to buffer
                        {not in Vi}
        When this option is set, the buffer shows up in the buffer list.  If
        it is reset it is not used for ":bnext", "ls", the Buffers menu, etc.
        This option is reset by Vim for buffers that are only used to remember
        a file name or marks.  Vim sets it when starting to edit a buffer.
        But not when moving to a buffer with ":buffer".
```

'buftype' *'bt'* *E382*
'buftype' 'bt' string (default: "")
local to buffer
{not in Vi}
The value of this option specifies the type of a buffer:
<empty> normal buffer
nofile buffer which is not related to a file and will not be written
nowrite buffer which will not be written
acwrite buffer which will always be written with BufWriteCmd autocommands. {not available when compiled without the |+autocmd| feature}
quickfix quickfix buffer, contains list of errors |:cwindow| or list of locations |:lwindow|
help help buffer (you are not supposed to set this manually)
terminal buffer for a |terminal| (you are not supposed to set this manually)

This option is used together with 'bufhidden' and 'swapfile' to specify special kinds of buffers. See |special-buffers|.

Be careful with changing this option, it can have many side effects!

A "quickfix" buffer is only used for the error list and the location list. This value is set by the |:cwindow| and |:lwindow| commands and you are not supposed to change it.

"nofile" and "nowrite" buffers are similar:
both: The buffer is not to be written to disk, ":w" doesn't work (":w filename" does work though).
both: The buffer is never considered to be |'modified'|. There is no warning when the changes will be lost, for example when you quit Vim.
both: A swap file is only created when using too much memory (when 'swapfile' has been reset there is never a swap file).
nofile only: The buffer name is fixed, it is not handled like a file name. It is not modified in response to a |:cd| command.
both: When using ":e bufname" and already editing "bufname" the buffer is made empty and autocommands are triggered as usual for |:edit|.
E676
"acwrite" implies that the buffer name is not related to a file, like "nofile", but it will be written. Thus, in contrast to "nofile" and "nowrite", ":w" does work and a modified buffer can't be abandoned without saving. For writing there must be matching |BufWriteCmd|, |FileWriteCmd| or |FileAppendCmd| autocommands.

'casemap' *'cmp'*
'casemap' 'cmp' string (default: "internal,keepascii")
global
{not in Vi}
{only available when compiled with the |+multi_byte| feature}
Specifies details about changing the case of letters. It may contain these words, separated by a comma:
internal Use internal case mapping functions, the current locale does not change the case mapping. This only matters when 'encoding' is a Unicode encoding,

```
                        "latin1" or "iso-8859-15".  When "internal" is
                        omitted, the towupper() and towlower() system library
                        functions are used when available.
        keepascii       For the ASCII characters (0x00 to 0x7f) use the US
                        case mapping, the current locale is not effective.
                        This probably only matters for Turkish.

                                                *'cdpath'* *'cd'* *E344* *E346*
'cdpath' 'cd'           string  (default: equivalent to $CDPATH or ",,")
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+file_in_path| feature}
        This is a list of directories which will be searched when using the
        |:cd| and |:lcd| commands, provided that the directory being searched
        for has a relative path, not an absolute part starting with "/", "./"
        or "../", the 'cdpath' option is not used then.
        The 'cdpath' option's value has the same form and semantics as
        |'path'|.  Also see |file-searching|.
        The default value is taken from $CDPATH, with a "," prepended to look
        in the current directory first.
        If the default value taken from $CDPATH is not what you want, include
        a modified version of the following command in your vimrc file to
        override it: >
          :let &cdpath = ',' . substitute(substitute($CDPATH, '[, ]', '\\\0', 'g'),
':', ',', 'g')
<       This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.
        (parts of 'cdpath' can be passed to the shell to expand file names).

                                                *'cedit'*
'cedit'                 string  (Vi default: "", Vim default: CTRL-F)
                        global
                        {not in Vi}
                        {not available when compiled without the |+vertsplit|
                        feature}
        The key used in Command-line Mode to open the command-line window.
        The default is CTRL-F when 'compatible' is off.
        Only non-printable keys are allowed.
        The key can be specified as a single character, but it is difficult to
        type.  The preferred way is to use the <> notation.  Examples: >
                :exe "set cedit=\<C-Y>"
                :exe "set cedit=\<Esc>"
<       |Nvi| also has this option, but it only uses the first character.
        See |cmdwin|.
        NOTE: This option is set to the Vim default value when 'compatible'
        is reset.

                                *'charconvert'* *'ccv'* *E202* *E214* *E513*
'charconvert' 'ccv'     string (default "")
                        global
                        {only available when compiled with the |+multi_byte|
                        and |+eval| features}
                        {not in Vi}
        An expression that is used for character encoding conversion.  It is
        evaluated when a file that is to be read or has been written has a
        different encoding from what is desired.
        'charconvert' is not used when the internal iconv() function is
        supported and is able to do the conversion.  Using iconv() is
        preferred, because it is much faster.
        'charconvert' is not used when reading stdin |--|, because there is no
        file to convert from.  You will have to save the text in a file first.
```

```
        The expression must return zero or an empty string for success,
        non-zero for failure.
        The possible encoding names encountered are in 'encoding'.
        Additionally, names given in 'fileencodings' and 'fileencoding' are
        used.
        Conversion between "latin1", "unicode", "ucs-2", "ucs-4" and "utf-8"
        is done internally by Vim, 'charconvert' is not used for this.
        'charconvert' is also used to convert the viminfo file, if the 'c'
        flag is present in 'viminfo'.  Also used for Unicode conversion.
        Example: >
                set charconvert=CharConvert()
                fun CharConvert()
                  system("recode "
                        \ . v:charconvert_from . ".." . v:charconvert_to
                        \ . " <" . v:fname_in . " >" v:fname_out)
                  return v:shell_error
                endfun
<       The related Vim variables are:
                v:charconvert_from      name of the current encoding
                v:charconvert_to        name of the desired encoding
                v:fname_in              name of the input file
                v:fname_out             name of the output file
        Note that v:fname_in and v:fname_out will never be the same.
        Note that v:charconvert_from and v:charconvert_to may be different
        from 'encoding'.  Vim internally uses UTF-8 instead of UCS-2 or UCS-4.
        Encryption is not done by Vim when using 'charconvert'.  If you want
        to encrypt the file after conversion, 'charconvert' should take care
        of this.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                    *'cindent'* *'cin'* *'nocindent'* *'nocin'*
'cindent' 'cin'         boolean (default off)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+cindent|
                        feature}
        Enables automatic C program indenting.  See 'cinkeys' to set the keys
        that trigger reindenting in insert mode and 'cinoptions' to set your
        preferred indent style.
        If 'indentexpr' is not empty, it overrules 'cindent'.
        If 'lisp' is not on and both 'indentexpr' and 'equalprg' are empty,
        the "=" operator indents using this algorithm rather than calling an
        external program.
        See |C-indenting|.
        When you don't like the way 'cindent' works, try the 'smartindent'
        option or 'indentexpr'.
        This option is not used when 'paste' is set.
        NOTE: This option is reset when 'compatible' is set.

                                                        *'cinkeys'* *'cink'*
'cinkeys' 'cink'        string  (default "0{,0},0),:,0#,!^F,o,O,e")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+cindent|
                        feature}
        A list of keys that, when typed in Insert mode, cause reindenting of
        the current line.  Only used if 'cindent' is on and 'indentexpr' is
        empty.
        For the format of this option see |cinkeys-format|.
        See |C-indenting|.
```

'cinoptions' *'cino'*
'cinoptions' 'cino'	string (default "")
			local to buffer
			{not in Vi}
			{not available when compiled without the |+cindent|
			feature}
	The 'cinoptions' affect the way 'cindent' reindents lines in a C
	program. See |cinoptions-values| for the values of this option, and
	|C-indenting| for info on C indenting in general.

'cinwords' *'cinw'*
'cinwords' 'cinw'	string (default "if,else,while,do,for,switch")
			local to buffer
			{not in Vi}
			{not available when compiled without both the
			|+cindent| and the |+smartindent| features}
	These keywords start an extra indent in the next line when
	'smartindent' or 'cindent' is set. For 'cindent' this is only done at
	an appropriate place (inside {}).
	Note that 'ignorecase' isn't used for 'cinwords'. If case doesn't
	matter, include the keyword both the uppercase and lowercase:
	"if,If,IF".

'clipboard' *'cb'*
'clipboard' 'cb'	string (default "autoselect,exclude:cons\|linux"
					  for X-windows, "" otherwise)
			global
			{not in Vi}
			{only in GUI versions or when the |+xterm_clipboard|
			feature is included}
	This option is a list of comma separated names.
	These names are recognized:

clipboard-unnamed
	unnamed		When included, Vim will use the clipboard register '*'
			for all yank, delete, change and put operations which
			would normally go to the unnamed register. When a
			register is explicitly specified, it will always be
			used regardless of whether "unnamed" is in 'clipboard'
			or not. The clipboard register can always be
			explicitly accessed using the "* notation. Also see
			|gui-clipboard|.

clipboard-unnamedplus
	unnamedplus	A variant of the "unnamed" flag which uses the
			clipboard register '+' (|quoteplus|) instead of
			register '*' for all yank, delete, change and put
			operations which would normally go to the unnamed
			register. When "unnamed" is also included to the
			option, yank operations (but not delete, change or
			put) will additionally copy the text into register
			'*'.
			Only available with the |+X11| feature.
			Availability can be checked with: >
				if has('unnamedplus')
<
clipboard-autoselect
	autoselect	Works like the 'a' flag in 'guioptions': If present,
			then whenever Visual mode is started, or the Visual
			area extended, Vim tries to become the owner of the
			windowing system's global selection or put the

```
                        selected text on the clipboard used by the selection
                        register "*.  See |guioptions_a| and |quotestar| for
                        details.  When the GUI is active, the 'a' flag in
                        'guioptions' is used, when the GUI is not active, this
                        "autoselect" flag is used.
                        Also applies to the modeless selection.

                                                *clipboard-autoselectplus*
        autoselectplus  Like "autoselect" but using the + register instead of
                        the * register.  Compare to the 'P' flag in
                        'guioptions'.

                                                *clipboard-autoselectml*
        autoselectml    Like "autoselect", but for the modeless selection
                        only.  Compare to the 'A' flag in 'guioptions'.

                                                *clipboard-html*
        html            When the clipboard contains HTML, use this when
                        pasting.  When putting text on the clipboard, mark it
                        as HTML.  This works to copy rendered HTML from
                        Firefox, paste it as raw HTML in Vim, select the HTML
                        in Vim and paste it in a rich edit box in Firefox.
                        You probably want to add this only temporarily,
                        possibly use BufEnter autocommands.
                        Only supported for GTK version 2 and later.
                        Only available with the |+multi_byte| feature.

                                                *clipboard-exclude*
        exclude:{pattern}
                        Defines a pattern that is matched against the name of
                        the terminal 'term'.  If there is a match, no
                        connection will be made to the X server.  This is
                        useful in this situation:
                        - Running Vim in a console.
                        - $DISPLAY is set to start applications on another
                          display.
                        - You do not want to connect to the X server in the
                          console, but do want this in a terminal emulator.
                        To never connect to the X server use: >
                                exclude:.*
<                       This has the same effect as using the |-X| argument.
                        Note that when there is no connection to the X server
                        the window title won't be restored and the clipboard
                        cannot be accessed.
                        The value of 'magic' is ignored, {pattern} is
                        interpreted as if 'magic' was on.
                        The rest of the option value will be used for
                        {pattern}, this must be the last entry.

                                                *'cmdheight'* *'ch'*
'cmdheight' 'ch'        number  (default 1)
                        global
                        {not in Vi}
        Number of screen lines to use for the command-line.  Helps avoiding
        |hit-enter| prompts.
        The value of this option is stored with the tab page, so that each tab
        page can have a different value.

                                                *'cmdwinheight'* *'cwh'*
'cmdwinheight' 'cwh'    number  (default 7)
                        global
                        {not in Vi}
```

```
                        {not available when compiled without the |+vertsplit|
                        feature}
        Number of screen lines to use for the command-line window. |cmdwin|

                                                *'colorcolumn'* *'cc'*
'colorcolumn' 'cc'      string  (default "")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        'colorcolumn' is a comma separated list of screen columns that are
        highlighted with ColorColumn |hl-ColorColumn|.  Useful to align
        text.  Will make screen redrawing slower.
        The screen column can be an absolute number, or a number preceded with
        '+' or '-', which is added to or subtracted from 'textwidth'. >

                :set cc=+1  " highlight column after 'textwidth'
                :set cc=+1,+2,+3  " highlight three columns after 'textwidth'
                :hi ColorColumn ctermbg=lightgrey guibg=lightgrey
<
        When 'textwidth' is zero then the items with '-' and '+' are not used.
        A maximum of 256 columns are highlighted.

                                                *'columns'* *'co'* *E594*
'columns' 'co'          number  (default 80 or terminal width)
                        global
                        {not in Vi}
        Number of columns of the screen.  Normally this is set by the terminal
        initialization and does not have to be set by hand.  Also see
        |posix-screen-size|.
        When Vim is running in the GUI or in a resizable window, setting this
        option will cause the window size to be changed.  When you only want
        to use the size for the GUI, put the command in your |gvimrc| file.
        When you set this option and Vim is unable to change the physical
        number of columns of the display, the display may be messed up.  For
        the GUI it is always possible and Vim limits the number of columns to
        what fits on the screen.  You can use this command to get the widest
        window possible: >
                :set columns=9999
<       Minimum value is 12, maximum value is 10000.

                                        *'comments'* *'com'* *E524* *E525*
'comments' 'com'        string  (default
                                "s1:/*,mb:*,ex:*/,://,b:#,:%,:XCOMM,n:>,fb:-")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+comments|
                        feature}
        A comma separated list of strings that can start a comment line.  See
        |format-comments|.  See |option-backslash| about using backslashes to
        insert a space.

                                        *'commentstring'* *'cms'* *E537*
'commentstring' 'cms'   string  (default "/*%s*/")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        A template for a comment.  The "%s" in the value is replaced with the
        comment text.  Currently only used to add markers for folding, see
        |fold-marker|.
```

```
                        *'compatible'* *'cp'* *'nocompatible'* *'nocp'*
'compatible' 'cp'       boolean (default on, off when a |vimrc| or |gvimrc|
                                     file is found, reset in |defaults.vim|)
                        global
                        {not in Vi}
        This option has the effect of making Vim either more Vi-compatible, or
        make Vim behave in a more useful way.

        This is a special kind of option, because when it's set or reset,
        other options are also changed as a side effect.
        NOTE: Setting or resetting this option can have a lot of unexpected
        effects: Mappings are interpreted in another way, undo behaves
        differently, etc.  If you set this option in your vimrc file, you
        should probably put it at the very start.

        By default this option is on and the Vi defaults are used for the
        options.  This default was chosen for those people who want to use Vim
        just like Vi, and don't even (want to) know about the 'compatible'
        option.
        When a |vimrc| or |gvimrc| file is found while Vim is starting up,
        this option is switched off, and all options that have not been
        modified will be set to the Vim defaults.  Effectively, this means
        that when a |vimrc| or |gvimrc| file exists, Vim will use the Vim
        defaults, otherwise it will use the Vi defaults.  (Note: This doesn't
        happen for the system-wide vimrc or gvimrc file, nor for a file given
        with the |-u| argument).  Also see |compatible-default| and
        |posix-compliance|.
        You can also set this option with the "-C" argument, and reset it with
        "-N".  See |-C| and |-N|.
        See 'cpoptions' for more fine tuning of Vi compatibility.

        When this option is set, numerous other options are set to make Vim as
        Vi-compatible as possible.  When this option is unset, various options
        are set to make Vim more useful.  The table below lists all the
        options affected.
        The {?} column indicates when the options are affected:
        +  Means that the option is set to the value given in {set value} when
           'compatible' is set.
        &  Means that the option is set to the value given in {set value} when
           'compatible' is set AND is set to its Vim default value when
           'compatible' is unset.
        -  Means the option is NOT changed when setting 'compatible' but IS
           set to its Vim default when 'compatible' is unset.
        The {effect} column summarises the change when 'compatible' is set.

        option          ? set value     effect ~

        'allowrevins'   + off           no CTRL-_ command
        'antialias'     + off           don't use antialiased fonts
        'arabic'        + off           reset arabic-related options
        'arabicshape'   + on            correct character shapes
        'backspace'     + ""            normal backspace
        'backup'        + off           no backup file
        'backupcopy'    & Unix: "yes"   backup file is a copy
                          else: "auto"  copy or rename backup file
        'balloonexpr'   + ""            text to show in evaluation balloon
        'breakindent'   + off           don't indent when wrapping lines
        'cedit'         - {unchanged}   {set vim default only on resetting 'cp'}
        'cindent'       + off           no C code indentation
        'compatible'    - {unchanged}   {set vim default only on resetting 'cp'}
        'copyindent'    + off           don't copy indent structure
        'cpoptions'     & (all flags)   Vi-compatible flags
```

```
        'cscopepathcomp'+ 0             don't show directories in tags list
        'cscoperelative'+ off
        'cscopetag'     + off           don't use cscope for ":tag"
        'cscopetagorder'+ 0             see |cscopetagorder|
        'cscopeverbose' + off           see |cscopeverbose|
        'delcombine'    + off           unicode: delete whole char combination
        'digraph'       + off           no digraphs
        'esckeys'       & off           no <Esc>-keys in Insert mode
        'expandtab'     + off           tabs not expanded to spaces
        'fileformats'   & ""            no automatic file format detection,
                          "dos,unix"    except for DOS, Windows and OS/2
        'formatexpr'    + ""            use 'formatprg' for auto-formatting
        'formatoptions' & "vt"          Vi compatible formatting
        'gdefault'      + off           no default 'g' flag for ":s"
        'history'       & 0             no commandline history
        'hkmap'         + off           no Hebrew keyboard mapping
        'hkmapp'        + off           no phonetic Hebrew keyboard mapping
        'hlsearch'      + off           no highlighting of search matches
        'incsearch'     + off           no incremental searching
        'indentexpr'    + ""            no indenting by expression
        'insertmode'    + off           do not start in Insert mode
        'iskeyword'     & "@,48-57,_"   keywords contain alphanumeric
                                                characters and '_'
        'joinspaces'    + on            insert 2 spaces after period
        'modeline'      & off           no modelines
        'more'          & off           no pauses in listings
        'mzquantum'     - {unchanged}   {set vim default only on resetting 'cp'}
        'numberwidth'   & 8             min number of columns for line number
        'preserveindent'+ off           don't preserve current indent structure
                                                when changing it
        'revins'        + off           no reverse insert
        'ruler'         + off           no ruler
        'scrolljump'    + 1             no jump scroll
        'scrolloff'     + 0             no scroll offset
        'shelltemp'     - {unchanged}   {set vim default only on resetting 'cp'}
        'shiftround'    + off           indent not rounded to shiftwidth
        'shortmess'     & ""            no shortening of messages
        'showcmd'       & off           command characters not shown
        'showmode'      & off           current mode not shown
        'sidescrolloff' + 0             cursor moves to edge of screen in scroll
        'smartcase'     + off           no automatic ignore case switch
        'smartindent'   + off           no smart indentation
        'smarttab'      + off           no smart tab size
        'softtabstop'   + 0             tabs are always 'tabstop' positions
        'startofline'   + on            goto startofline with some commands
        'tagcase'       & "followic"    'ignorecase' when searching tags file
        'tagrelative'   & off           tag file names are not relative
        'termguicolors' + off           don't use highlight-(guifg|guibg)
        'textauto'      & off           no automatic textmode detection
        'textwidth'     + 0             no automatic line wrap
        'tildeop'       + off           tilde is not an operator
        'ttimeout'      + off           no terminal timeout
        'undofile'      + off           don't use an undo file
        'viminfo'       - {unchanged}   {set Vim default only on resetting 'cp'}
        'virtualedit'   + ""            cursor can only be placed on characters
        'whichwrap'     & ""            left-right movements don't wrap
        'wildchar'      & CTRL-E        only when the current value is <Tab>
                                        use CTRL-E for cmdline completion
        'writebackup'   + on or off     depends on the |+writebackup| feature

                                                *'complete'* *'cpt'* *E535*
'complete' 'cpt'        string  (default: ".,w,b,u,t,i")
```

```
                        local to buffer
                        {not in Vi}
        This option specifies how keyword completion |ins-completion| works
        when CTRL-P or CTRL-N are used.  It is also used for whole-line
        completion |i_CTRL-X_CTRL-L|.  It indicates the type of completion
        and the places to scan.  It is a comma separated list of flags:
        .       scan the current buffer ('wrapscan' is ignored)
        w       scan buffers from other windows
        b       scan other loaded buffers that are in the buffer list
        u       scan the unloaded buffers that are in the buffer list
        U       scan the buffers that are not in the buffer list
        k       scan the files given with the 'dictionary' option
        kspell  use the currently active spell checking |spell|
        k{dict} scan the file {dict}.  Several "k" flags can be given,
                patterns are valid too.  For example: >
                        :set cpt=k/usr/dict/*,k~/spanish
<       s       scan the files given with the 'thesaurus' option
        s{tsr}  scan the file {tsr}.  Several "s" flags can be given, patterns
                are valid too.
        i       scan current and included files
        d       scan current and included files for defined name or macro
                |i_CTRL-X_CTRL-D|
        ]       tag completion
        t       same as "]"

        Unloaded buffers are not loaded, thus their autocmds |:autocmd| are
        not executed, this may lead to unexpected completions from some files
        (gzipped files for example).  Unloaded buffers are not scanned for
        whole-line completion.

        The default is ".,w,b,u,t,i", which means to scan:
           1. the current buffer
           2. buffers in other windows
           3. other loaded buffers
           4. unloaded buffers
           5. tags
           6. included files

        As you can see, CTRL-N and CTRL-P can be used to do any 'iskeyword'-
        based expansion (e.g., dictionary |i_CTRL-X_CTRL-K|, included patterns
        |i_CTRL-X_CTRL-I|, tags |i_CTRL-X_CTRL-]| and normal expansions).

                                                *'completefunc'* *'cfu'*
'completefunc' 'cfu'    string  (default: empty)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+eval|
                        or |+insert_expand| features}
        This option specifies a function to be used for Insert mode completion
        with CTRL-X CTRL-U. |i_CTRL-X_CTRL-U|
        See |complete-functions| for an explanation of how the function is
        invoked and what it should return.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'completeopt'* *'cot'*
'completeopt' 'cot'     string  (default: "menu,preview")
                        global
                        {not available when compiled without the
                        |+insert_expand| feature}
                        {not in Vi}
        A comma separated list of options for Insert mode completion
```

```
        |ins-completion|.  The supported values are:

           menu     Use a popup menu to show the possible completions.  The
                    menu is only shown when there is more than one match and
                    sufficient colors are available.  |ins-completion-menu|

           menuone  Use the popup menu also when there is only one match.
                    Useful when there is additional information about the
                    match, e.g., what file it comes from.

           longest  Only insert the longest common text of the matches.  If
                    the menu is displayed you can use CTRL-L to add more
                    characters.  Whether case is ignored depends on the kind
                    of completion.  For buffer text the 'ignorecase' option is
                    used.

           preview  Show extra information about the currently selected
                    completion in the preview window.  Only works in
                    combination with "menu" or "menuone".

           noinsert Do not insert any text for a match until the user selects
                    a match from the menu. Only works in combination with
                    "menu" or "menuone". No effect if "longest" is present.

           noselect Do not select a match in the menu, force the user to
                    select one from the menu. Only works in combination with
                    "menu" or "menuone".


                                                  *'concealcursor'* *'cocu'*
'concealcursor' 'cocu'  string (default: "")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+conceal|
                        feature}
        Sets the modes in which text in the cursor line can also be concealed.
        When the current mode is listed then concealing happens just like in
        other lines.
          n             Normal mode
          v             Visual mode
          i             Insert mode
          c             Command line editing, for 'incsearch'

        'v' applies to all lines in the Visual area, not only the cursor.
        A useful value is "nc".  This is used in help files.  So long as you
        are moving around text is concealed, but when starting to insert text
        or selecting a Visual area the concealed text is displayed, so that
        you can see what you are doing.
        Keep in mind that the cursor position is not always where it's
        displayed.  E.g., when moving vertically it may change column.


'conceallevel' 'cole'           *'conceallevel'* *'cole'*
                        number (default 0)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+conceal|
                        feature}
        Determine how text with the "conceal" syntax attribute |:syn-conceal|
        is shown:

        Value           Effect ~
```

```
        0               Text is shown normally
        1               Each block of concealed text is replaced with one
                        character.  If the syntax item does not have a custom
                        replacement character defined (see |:syn-cchar|) the
                        character defined in 'listchars' is used (default is a
                        space).
                        It is highlighted with the "Conceal" highlight group.
        2               Concealed text is completely hidden unless it has a
                        custom replacement character defined (see
                        |:syn-cchar|).
        3               Concealed text is completely hidden.

        Note: in the cursor line concealed text is not hidden, so that you can
        edit and copy the text.  This can be changed with the 'concealcursor'
        option.

                                *'confirm'* *'cf'* *'noconfirm'* *'nocf'*
'confirm' 'cf'          boolean (default off)
                        global
                        {not in Vi}
        When 'confirm' is on, certain operations that would normally
        fail because of unsaved changes to a buffer, e.g. ":q" and ":e",
        instead raise a |dialog| asking if you wish to save the current
        file(s).  You can still use a ! to unconditionally |abandon| a buffer.
        If 'confirm' is off you can still activate confirmation for one
        command only (this is most useful in mappings) with the |:confirm|
        command.
        Also see the |confirm()| function and the 'v' flag in 'guioptions'.

                        *'conskey'* *'consk'* *'noconskey'* *'noconsk'*
'conskey' 'consk'       boolean (default off)
                        global
                        {not in Vi}  {only for MS-DOS}
        This was for MS-DOS and is no longer supported.

                        *'copyindent'* *'ci'* *'nocopyindent'* *'noci'*
'copyindent' 'ci'       boolean (default off)
                        local to buffer
                        {not in Vi}
        Copy the structure of the existing lines indent when autoindenting a
        new line.  Normally the new indent is reconstructed by a series of
        tabs followed by spaces as required (unless |'expandtab'| is enabled,
        in which case only spaces are used).  Enabling this option makes the
        new line copy whatever characters were used for indenting on the
        existing line.  'expandtab' has no effect on these characters, a Tab
        remains a Tab.  If the new indent is greater than on the existing
        line, the remaining space is filled in the normal manner.
        NOTE: This option is reset when 'compatible' is set.
        Also see 'preserveindent'.

                                                *'cpoptions'* *'cpo'* *cpo*
'cpoptions' 'cpo'       string  (Vim default: "aABceFs",
                                 Vi default:  all flags)
                        global
                        {not in Vi}
        A sequence of single character flags.  When a character is present
        this indicates Vi-compatible behavior.  This is used for things where
        not being Vi-compatible is mostly or sometimes preferred.
        'cpoptions' stands for "compatible-options".
        Commas can be added for readability.
        To avoid problems with flags that are added in the future, use the
        "+=" and "-=" feature of ":set" |add-option-flags|.
```

NOTE: This option is set to the Vi default value when 'compatible' is
set and to the Vim default value when 'compatible' is reset.
NOTE: This option is set to the POSIX default value at startup when
the Vi default value would be used and the $VIM_POSIX environment
variable exists |posix|. This means Vim tries to behave like the
POSIX specification.

contains	behavior ~	
		cpo-a
a	When included, a ":read" command with a file name argument will set the alternate file name for the current window.	
		cpo-A
A	When included, a ":write" command with a file name argument will set the alternate file name for the current window.	
		cpo-b
b	"|" in a ":map" command is recognized as the end of the map command. The '\' is included in the mapping, the text after the '|' is interpreted as the next command. Use a CTRL-V instead of a backslash to include the '|' in the mapping. Applies to all mapping, abbreviation, menu and autocmd commands. See also |map_bar|.	
		cpo-B
B	A backslash has no special meaning in mappings, abbreviations and the "to" part of the menu commands. Remove this flag to be able to use a backslash like a CTRL-V. For example, the command ":map X \<Esc>" results in X being mapped to: 'B' included: "\^[" (^[is a real <Esc>) 'B' excluded: "<Esc>" (5 characters) ('<' excluded in both cases)	
		cpo-c
c	Searching continues at the end of any match at the cursor position, but not further than the start of the next line. When not present searching continues one character from the cursor position. With 'c' "abababababab" only gets three matches when repeating "/abab", without 'c' there are five matches.	
		cpo-C
C	Do not concatenate sourced lines that start with a backslash. See |line-continuation|.	
		cpo-d
d	Using "./" in the 'tags' option doesn't mean to use the tags file relative to the current file, but the tags file in the current directory.	
		cpo-D
D	Can't use CTRL-K to enter a digraph after Normal mode commands with a character argument, like |r|, |f| and |t|.	
		cpo-e
e	When executing a register with ":@r", always add a <CR> to the last line, also when the register is not linewise. If this flag is not present, the register is not linewise and the last line does not end in a <CR>, then the last line is put on the command-line and can be edited before hitting <CR>.	
		cpo-E
E	It is an error when using "y", "d", "c", "g~", "gu" or "gU" on an Empty region. The operators only work when at least one character is to be operate on. Example:	

```
        This makes "y0" fail in the first column.
                                                *cpo-f*
f       When included, a ":read" command with a file name
        argument will set the file name for the current buffer,
        if the current buffer doesn't have a file name yet.
                                                *cpo-F*
F       When included, a ":write" command with a file name
        argument will set the file name for the current
        buffer, if the current buffer doesn't have a file name
        yet.  Also see |cpo-P|.
                                                *cpo-g*
g       Goto line 1 when using ":edit" without argument.
                                                *cpo-H*
H       When using "I" on a line with only blanks, insert
        before the last blank.  Without this flag insert after
        the last blank.
                                                *cpo-i*
i       When included, interrupting the reading of a file will
        leave it modified.
                                                *cpo-I*
I       When moving the cursor up or down just after inserting
        indent for 'autoindent', do not delete the indent.
                                                *cpo-j*
j       When joining lines, only add two spaces after a '.',
        not after '!' or '?'.  Also see 'joinspaces'.
                                                *cpo-J*
J       A |sentence| has to be followed by two spaces after
        the '.', '!' or '?'.  A <Tab> is not recognized as
        white space.
                                                *cpo-k*
k       Disable the recognition of raw key codes in
        mappings, abbreviations, and the "to" part of menu
        commands.  For example, if <Key> sends ^[OA (where ^[
        is <Esc>), the command ":map X ^[OA" results in X
        being mapped to:
                'k' included:   "^[OA"   (3 characters)
                'k' excluded:   "<Key>"  (one key code)
        Also see the '<' flag below.
                                                *cpo-K*
K       Don't wait for a key code to complete when it is
        halfway a mapping.  This breaks mapping <F1><F1> when
        only part of the second <F1> has been read.  It
        enables cancelling the mapping by typing <F1><Esc>.
                                                *cpo-l*
l       Backslash in a [] range in a search pattern is taken
        literally, only "\]", "\^", "\-" and "\\" are special.
        See |/[]|
           'l' included: "/[ \t]"  finds <Space>, '\' and 't'
           'l' excluded: "/[ \t]"  finds <Space> and <Tab>
        Also see |cpo-\|.
                                                *cpo-L*
L       When the 'list' option is set, 'wrapmargin',
        'textwidth', 'softtabstop' and Virtual Replace mode
        (see |gR|) count a <Tab> as two characters, instead of
        the normal behavior of a <Tab>.
                                                *cpo-m*
m       When included, a showmatch will always wait half a
        second.  When not included, a showmatch will wait half
        a second or until a character is typed.  |'showmatch'|
                                                *cpo-M*
M       When excluded, "%" matching will take backslashes into
        account.  Thus in "( \( )" and "\( ( \)" the outer
```

parenthesis match. When included "%" ignores backslashes, which is Vi compatible.

cpo-n

n When included, the column used for 'number' and 'relativenumber' will also be used for text of wrapped lines.

cpo-o

o Line offset to search command is not remembered for next search.

cpo-O

O Don't complain if a file is being overwritten, even when it didn't exist when editing it. This is a protection against a file unexpectedly created by someone else. Vi didn't complain about this.

cpo-p

p Vi compatible Lisp indenting. When not present, a slightly better algorithm is used.

cpo-P

P When included, a ":write" command that appends to a file will set the file name for the current buffer, if the current buffer doesn't have a file name yet and the 'F' flag is also included |cpo-F|.

cpo-q

q When joining multiple lines leave the cursor at the position where it would be when joining two lines.

cpo-r

r Redo ("." command) uses "/" to repeat a search command, instead of the actually used search string.

cpo-R

R Remove marks from filtered lines. Without this flag marks are kept like |:keepmarks| was used.

cpo-s

s Set buffer options when entering the buffer for the first time. This is like it is in Vim version 3.0. And it is the default. If not present the options are set when the buffer is created.

cpo-S

S Set buffer options always when entering a buffer (except 'readonly', 'fileformat', 'filetype' and 'syntax'). This is the (most) Vi compatible setting. The options are set to the values in the current buffer. When you change an option and go to another buffer, the value is copied. Effectively makes the buffer options global to all buffers.

's'	'S'	copy buffer options
no	no	when buffer created
yes	no	when buffer first entered (default)
X	yes	each time when buffer entered (vi comp.)

cpo-t

t Search pattern for the tag command is remembered for "n" command. Otherwise Vim only puts the pattern in the history for search pattern, but doesn't change the last used search pattern.

cpo-u

u Undo is Vi compatible. See |undo-two-ways|.

cpo-v

v Backspaced characters remain visible on the screen in Insert mode. Without this flag the characters are erased from the screen right away. With this flag the screen newly typed text overwrites backspaced characters.

cpo-w
w When using "cw" on a blank character, only change one character and not all blanks until the start of the next word.
cpo-W
W Don't overwrite a readonly file. When omitted, ":w!" overwrites a readonly file, if possible.
cpo-x
x <Esc> on the command-line executes the command-line. The default in Vim is to abandon the command-line, because <Esc> normally aborts a command. |c_<Esc>|
cpo-X
X When using a count with "R" the replaced text is deleted only once. Also when repeating "R" with "." and a count.
cpo-y
y A yank command can be redone with ".".
cpo-Z
Z When using "w!" while the 'readonly' option is set, don't reset 'readonly'.
cpo-!
! When redoing a filter command, use the last used external command, whatever it was. Otherwise the last used -filter- command is used.
cpo-$
$ When making a change to one line, don't redisplay the line, but put a '$' at the end of the changed text. The changed text will be overwritten when you type the new text. The line is redisplayed if you type any command that moves the cursor from the insertion point.
cpo-%
% Vi-compatible matching is done for the "%" command. Does not recognize "#if", "#endif", etc. Does not recognize "/*" and "*/". Parens inside single and double quotes are also counted, causing a string that contains a paren to disturb the matching. For example, in a line like "if (strcmp("foo(", s))" the first paren does not match the last one. When this flag is not included, parens inside single and double quotes are treated specially. When matching a paren outside of quotes, everything inside quotes is ignored. When matching a paren inside quotes, it will find the matching one (if there is one). This works very well for C programs. This flag is also used for other features, such as C-indenting.
cpo--
- When included, a vertical movement command fails when it would go above the first line or below the last line. Without it the cursor moves to the first or last line, unless it already was in that line. Applies to the commands "-", "k", CTRL-P, "+", "j", CTRL-N, CTRL-J and ":1234".
cpo-+
+ When included, a ":write file" command will reset the 'modified' flag of the buffer, even though the buffer itself may still be different from its file.
cpo-star
* Use ":*" in the same way as ":@". When not included, ":*" is an alias for ":'<,'>", select the Visual area.
cpo-<

```
                <       Disable the recognition of special key codes in |<>|
                        form in mappings, abbreviations, and the "to" part of
                        menu commands.  For example, the command
                        ":map X <Tab>" results in X being mapped to:
                                '<' included:   "<Tab>"  (5 characters)
                                '<' excluded:   "^I"     (^I is a real <Tab>)
                        Also see the 'k' flag above.
                                                                *cpo->*
                >       When appending to a register, put a line break before
                        the appended text.
                                                                *cpo-;*
                ;       When using |,| or |;| to repeat the last |t| search
                        and the cursor is right in front of the searched
                        character, the cursor won't move. When not included,
                        the cursor would skip over it and jump to the
                        following occurrence.

        POSIX flags.  These are not included in the Vi default value, except
        when $VIM_POSIX was set on startup. |posix|

            contains    behavior        ~
                                                                *cpo-#*
                #       A count before "D", "o" and "O" has no effect.
                                                                *cpo-&*
                &       When ":preserve" was used keep the swap file when
                        exiting normally while this buffer is still loaded.
                        This flag is tested when exiting.
                                                                *cpo-\*
                \       Backslash in a [] range in a search pattern is taken
                        literally, only "\]" is special  See |/[]|
                           '\' included: "/[ \-]"  finds <Space>, '\' and '-'
                           '\' excluded: "/[ \-]"  finds <Space> and '-'
                        Also see |cpo-l|.
                                                                *cpo-/*
                /       When "%" is used as the replacement string in a |:s|
                        command, use the previous replacement string. |:s%|
                                                                *cpo-{*
                {       The |{| and |}| commands also stop at a "{" character
                        at the start of a line.
                                                                *cpo-.*
                .       The ":chdir" and ":cd" commands fail if the current
                        buffer is modified, unless ! is used.  Vim doesn't
                        need this, since it remembers the full path of an
                        opened file.
                                                                *cpo-bar*
                |       The value of the $LINES and $COLUMNS environment
                        variables overrule the terminal size values obtained
                        with system specific functions.


                                                *'cryptmethod'* *'cm'*
'cryptmethod' 'cm'      string  (default "zip")
                        global or local to buffer |global-local|
                        {not in Vi}
        Method used for encryption when the buffer is written to a file:
                                                        *pkzip*
           zip          PkZip compatible method.  A weak kind of encryption.
                        Backwards compatible with Vim 7.2 and older.
                                                        *blowfish*
           blowfish     Blowfish method.  Medium strong encryption but it has
                        an implementation flaw.  Requires Vim 7.3 or later,
                        files can NOT be read by Vim 7.2 and older.  This adds
```

```
                        a "seed" to the file, every time you write the file
                        the encrypted bytes will be different.
                                                        *blowfish2*
            blowfish2   Blowfish method.  Medium strong encryption.  Requires
                        Vim 7.4.401 or later, files can NOT be read by Vim 7.3
                        and older.  This adds a "seed" to the file, every time
                        you write the file the encrypted bytes will be
                        different.  The whole undo file is encrypted, not just
                        the pieces of text.

        You should use "blowfish2", also to re-encrypt older files.

        When reading an encrypted file 'cryptmethod' will be set automatically
        to the detected method of the file being read.  Thus if you write it
        without changing 'cryptmethod' the same method will be used.
        Changing 'cryptmethod' does not mark the file as modified, you have to
        explicitly write it, you don't get a warning unless there are other
        modifications.  Also see |:X|.

        When setting the global value to an empty string, it will end up with
        the value "zip".  When setting the local value to an empty string the
        buffer will use the global value.

        When a new encryption method is added in a later version of Vim, and
        the current version does not recognize it, you will get *E821* .
        You need to edit this file with the later version of Vim.


                                                *'cscopepathcomp'* *'cspc'*
'cscopepathcomp' 'cspc' number  (default 0)
                        global
                        {not available when compiled without the |+cscope|
                        feature}
                        {not in Vi}
        Determines how many components of the path to show in a list of tags.
        See |cscopepathcomp|.
        NOTE: This option is set to 0 when 'compatible' is set.

                                                *'cscopeprg'* *'csprg'*
'cscopeprg' 'csprg'     string  (default "cscope")
                        global
                        {not available when compiled without the |+cscope|
                        feature}
                        {not in Vi}
        Specifies the command to execute cscope.  See |cscopeprg|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'cscopequickfix'* *'csqf'*
'cscopequickfix' 'csqf' string  (default "")
                        global
                        {not available when compiled without the |+cscope|
                        or |+quickfix| features}
                        {not in Vi}
        Specifies whether to use quickfix window to show cscope results.
        See |cscopequickfix|.

                *'cscoperelative'* *'csre'* *'nocscoperelative'* *'nocsre'*
'cscoperelative' 'csre' boolean (default off)
                        global
                        {not available when compiled without the |+cscope|
                        feature}
```

```
                        {not in Vi}
        In the absence of a prefix (-P) for cscope. setting this option enables
        to use the basename of cscope.out path as the prefix.
        See |cscoperelative|.
        NOTE: This option is reset when 'compatible' is set.

                                *'cscopetag'* *'cst'* *'nocscopetag'* *'nocst'*
'cscopetag' 'cst'       boolean (default off)
                        global
                        {not available when compiled without the |+cscope|
                        feature}
                        {not in Vi}
        Use cscope for tag commands.  See |cscope-options|.
        NOTE: This option is reset when 'compatible' is set.

                                                *'cscopetagorder'* *'csto'*
'cscopetagorder' 'csto' number  (default 0)
                        global
                        {not available when compiled without the |+cscope|
                        feature}
                        {not in Vi}
        Determines the order in which ":cstag" performs a search.  See
        |cscopetagorder|.
        NOTE: This option is set to 0 when 'compatible' is set.

                                        *'cscopeverbose'* *'csverb'*
                                        *'nocscopeverbose'* *'nocsverb'*
'cscopeverbose' 'csverb' boolean (default off)
                        global
                        {not available when compiled without the |+cscope|
                        feature}
                        {not in Vi}
        Give messages when adding a cscope database.  See |cscopeverbose|.
        NOTE: This option is reset when 'compatible' is set.

                        *'cursorbind'* *'crb'* *'nocursorbind'* *'nocrb'*
'cursorbind' 'crb'      boolean  (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+cursorbind|
                        feature}
        When this option is set, as the cursor in the current
        window moves other cursorbound windows (windows that also have
        this option set) move their cursors to the corresponding line and
        column.  This option is useful for viewing the
        differences between two versions of a file (see 'diff'); in diff mode,
        inserted and deleted lines (though not characters within a line) are
        taken into account.


                        *'cursorcolumn'* *'cuc'* *'nocursorcolumn'* *'nocuc'*
'cursorcolumn' 'cuc'    boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        Highlight the screen column of the cursor with CursorColumn
        |hl-CursorColumn|.  Useful to align text.  Will make screen redrawing
        slower.
        If you only want the highlighting in the current window you can use
        these autocommands: >
                au WinLeave * set nocursorline nocursorcolumn
```

```
                au WinEnter * set cursorline cursorcolumn
<

                        *'cursorline'* *'cul'* *'nocursorline'* *'nocul'*
'cursorline' 'cul'      boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        Highlight the screen line of the cursor with CursorLine
        |hl-CursorLine|.  Useful to easily spot the cursor.  Will make screen
        redrawing slower.
        When Visual mode is active the highlighting isn't used to make it
        easier to see the selected text.


                                                *'debug'*
'debug'                 string  (default "")
                        global
                        {not in Vi}
        These values can be used:
        msg     Error messages that would otherwise be omitted will be given
                anyway.
        throw   Error messages that would otherwise be omitted will be given
                anyway and also throw an exception and set |v:errmsg|.
        beep    A message will be given when otherwise only a beep would be
                produced.
        The values can be combined, separated by a comma.
        "msg" and "throw" are useful for debugging 'foldexpr', 'formatexpr' or
        'indentexpr'.

                                                *'define'* *'def'*
'define' 'def'          string  (default "^\s*#\s*define")
                        global or local to buffer |global-local|
                        {not in Vi}
        Pattern to be used to find a macro definition.  It is a search
        pattern, just like for the "/" command.  This option is used for the
        commands like "[i" and "[d" |include-search|.  The 'isident' option is
        used to recognize the defined name after the match:
                {match with 'define'}{non-ID chars}{defined name}{non-ID char}
        See |option-backslash| about inserting backslashes to include a space
        or backslash.
        The default value is for C programs.  For C++ this value would be
        useful, to include const type declarations: >
                ^\(#\s*define\|[a-z]*\s*const\s*[a-z]*\)
<       When using the ":set" command, you need to double the backslashes!

                        *'delcombine'* *'deco'* *'nodelcombine'* *'nodeco'*
'delcombine' 'deco'     boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+multi_byte|
                        feature}
        If editing Unicode and this option is set, backspace and Normal mode
        "x" delete each combining character on its own.  When it is off (the
        default) the character along with its combining characters are
        deleted.
        Note: When 'delcombine' is set "xx" may work different from "2x"!

        This is useful for Arabic, Hebrew and many other languages where one
        may have combining characters overtop of base characters, and want
        to remove only the combining ones.
```

```
        NOTE: This option is reset when 'compatible' is set.

                                                *'dictionary'* *'dict'*
'dictionary' 'dict'     string  (default "")
                        global or local to buffer |global-local|
                        {not in Vi}
        List of file names, separated by commas, that are used to lookup words
        for keyword completion commands |i_CTRL-X_CTRL-K|.  Each file should
        contain a list of words.  This can be one word per line, or several
        words per line, separated by non-keyword characters (white space is
        preferred).  Maximum line length is 510 bytes.
        When this option is empty, or an entry "spell" is present, spell
        checking is enabled the currently active spelling is used. |spell|
        To include a comma in a file name precede it with a backslash.  Spaces
        after a comma are ignored, otherwise spaces are included in the file
        name.  See |option-backslash| about using backslashes.
        This has nothing to do with the |Dictionary| variable type.
        Where to find a list of words?
        - On FreeBSD, there is the file "/usr/share/dict/words".
        - In the Simtel archive, look in the "msdos/linguist" directory.
        - In "miscfiles" of the GNU collection.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        directories from the list.  This avoids problems when a future version
        uses another default.
        Backticks cannot be used in this option for security reasons.

                                                           *'diff'* *'nodiff'*
'diff'                  boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+diff|
                        feature}
        Join the current window in the group of windows that shows differences
        between files.  See |vimdiff|.

                                                *'dex'* *'diffexpr'*
'diffexpr' 'dex'        string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+diff|
                        feature}
        Expression which is evaluated to obtain an ed-style diff file from two
        versions of a file.  See |diff-diffexpr|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'dip'* *'diffopt'*
'diffopt' 'dip'         string  (default "filler")
                        global
                        {not in Vi}
                        {not available when compiled without the |+diff|
                        feature}
        Option settings for diff mode.  It can consist of the following items.
        All are optional.  Items must be separated by a comma.

                filler          Show filler lines, to keep the text
                                synchronized with a window that has inserted
                                lines at the same position.  Mostly useful
                                when windows are side-by-side and 'scrollbind'
                                is set.

                context:{n}     Use a context of {n} lines between a change
```

```
                                and a fold that contains unchanged lines.
                                When omitted a context of six lines is used.
                                See |fold-diff|.

                icase           Ignore changes in case of text.  "a" and "A"
                                are considered the same.  Adds the "-i" flag
                                to the "diff" command if 'diffexpr' is empty.

                iwhite          Ignore changes in amount of white space.  Adds
                                the "-b" flag to the "diff" command if
                                'diffexpr' is empty.  Check the documentation
                                of the "diff" command for what this does
                                exactly.  It should ignore adding trailing
                                white space, but not leading white space.

                horizontal      Start diff mode with horizontal splits (unless
                                explicitly specified otherwise).

                vertical        Start diff mode with vertical splits (unless
                                explicitly specified otherwise).

                foldcolumn:{n}  Set the 'foldcolumn' option to {n} when
                                starting diff mode.  Without this 2 is used.

        Examples: >

                :set diffopt=filler,context:4
                :set diffopt=
                :set diffopt=filler,foldcolumn:3
<
                                        *'digraph'* *'dg'* *'nodigraph'* *'nodg'*
'digraph' 'dg'          boolean (default off)
                        global
                        {not in Vi}
                        {not available when compiled without the |+digraphs|
                        feature}
        Enable the entering of digraphs in Insert mode with {char1} <BS>
        {char2}.  See |digraphs|.
        NOTE: This option is reset when 'compatible' is set.

                                                *'directory'* *'dir'*
'directory' 'dir'       string  (default for Amiga: ".,t:",
                                 for MS-DOS and Win32: ".,$TEMP,c:\tmp,c:\temp"
                                 for Unix: ".,~/tmp,/var/tmp,/tmp")
                        global
        List of directory names for the swap file, separated with commas.
        - The swap file will be created in the first directory where this is
          possible.
        - Empty means that no swap file will be used (recovery is
          impossible!).
        - A directory "." means to put the swap file in the same directory as
          the edited file.  On Unix, a dot is prepended to the file name, so
          it doesn't show in a directory listing.  On MS-Windows the "hidden"
          attribute is set and a dot prepended if possible.
        - A directory starting with "./" (or ".\" for MS-DOS et al.) means to
          put the swap file relative to where the edited file is.  The leading
          "." is replaced with the path name of the edited file.
        - For Unix and Win32, if a directory ends in two path separators "//"
          or "\\", the swap file name will be built from the complete path to
          the file with all path separators substituted to percent '%' signs.
          This will ensure file name uniqueness in the preserve directory.
          On Win32, when a separating comma is following, you must use "//",
```

```
          since "\\" will include the comma in the file name.
        - Spaces after the comma are ignored, other spaces are considered part
          of the directory name.  To have a space at the start of a directory
          name, precede it with a backslash.
        - To include a comma in a directory name precede it with a backslash.
        - A directory name may end in an ':' or '/'.
        - Environment variables are expanded |:set_env|.
        - Careful with '\' characters, type one before a space, type two to
          get one in the option (see |option-backslash|), for example: >
            :set dir=c:\\tmp,\ dir\\,with\\,commas,\\\ dir\ with\ spaces
<       - For backwards compatibility with Vim version 3.0 a '>' at the start
          of the option is removed.
        Using "." first in the list is recommended.  This means that editing
        the same file twice will result in a warning.  Using "/tmp" on Unix is
        discouraged: When the system crashes you lose the swap file.
        "/var/tmp" is often not cleared when rebooting, thus is a better
        choice than "/tmp".  But it can contain a lot of files, your swap
        files get lost in the crowd.  That is why a "tmp" directory in your
        home directory is tried first.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        directories from the list.  This avoids problems when a future version
        uses another default.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.
        {Vi: directory to put temp file in, defaults to "/tmp"}

                                            *'display'* *'dy'*
'display' 'dy'          string  (default "", set to "truncate" in
                                                              |defaults.vim|)
                        global
                        {not in Vi}
        Change the way text is displayed.  This is comma separated list of
        flags:
        lastline        When included, as much as possible of the last line
                        in a window will be displayed.  "@@@" is put in the
                        last columns of the last screen line to indicate the
                        rest of the line is not displayed.
        truncate        Like "lastline", but "@@@" is displayed in the first
                        column of the last screen line.  Overrules "lastline".
        uhex            Show unprintable characters hexadecimal as <xx>
                        instead of using ^C and ~C.

        When neither "lastline" nor "truncate" is included, a last line that
        doesn't fit is replaced with "@" lines.

                                                  *'eadirection'* *'ead'*
'eadirection' 'ead'     string  (default "both")
                        global
                        {not in Vi}
                        {not available when compiled without the |+vertsplit|
                        feature}
        Tells when the 'equalalways' option applies:
                ver     vertically, width of windows is not affected
                hor     horizontally, height of windows is not affected
                both    width and height of windows is affected

                           *'ed'* *'edcompatible'* *'noed'* *'noedcompatible'*
'edcompatible' 'ed'     boolean (default off)
                        global
        Makes the 'g' and 'c' flags of the ":substitute" command to be
        toggled each time the flag is given.  See |complex-change|.  See
        also 'gdefault' option.
```

```
        Switching this option on may break plugins!

                                            *'emoji'* *'emo'* *'noemoji'* *'noemo'*
'emoji' 'emo'   boolean (default: on)
                        global
                        {not in Vi}
                        {only available when compiled with the |+multi_byte|
                        feature}
        When on all Unicode emoji characters are considered to be full width.


                                            *'encoding'* *'enc'* *E543*
'encoding' 'enc'        string (default: "latin1" or value from $LANG)
                        global
                        {only available when compiled with the |+multi_byte|
                        feature}
                        {not in Vi}
        Sets the character encoding used inside Vim.  It applies to text in
        the buffers, registers, Strings in expressions, text stored in the
        viminfo file, etc.  It sets the kind of characters which Vim can work
        with.  See |encoding-names| for the possible values.

        NOTE: Changing this option will not change the encoding of the
        existing text in Vim.  It may cause non-ASCII text to become invalid.
        It should normally be kept at its default value, or set when Vim
        starts up.  See |multibyte|.  To reload the menus see |:menutrans|.

        This option cannot be set from a |modeline|.  It would most likely
        corrupt the text.

        NOTE: For GTK+ 2 or later, it is highly recommended to set 'encoding'
        to "utf-8".  Although care has been taken to allow different values of
        'encoding', "utf-8" is the natural choice for the environment and
        avoids unnecessary conversion overhead.  "utf-8" has not been made
        the default to prevent different behavior of the GUI and terminal
        versions, and to avoid changing the encoding of newly created files
        without your knowledge (in case 'fileencodings' is empty).

        The character encoding of files can be different from 'encoding'.
        This is specified with 'fileencoding'.  The conversion is done with
        iconv() or as specified with 'charconvert'.

        If you need to know whether 'encoding' is a multi-byte encoding, you
        can use: >
                if has("multi_byte_encoding")
<
        Normally 'encoding' will be equal to your current locale.  This will
        be the default if Vim recognizes your environment settings.  If
        'encoding' is not set to the current locale, 'termencoding' must be
        set to convert typed and displayed text.  See |encoding-table|.

        When you set this option, it fires the |EncodingChanged| autocommand
        event so that you can set up fonts if necessary.

        When the option is set, the value is converted to lowercase.  Thus
        you can set it with uppercase values too.  Underscores are translated
        to '-' signs.
        When the encoding is recognized, it is changed to the standard name.
        For example "Latin-1" becomes "latin1", "ISO_88592" becomes
        "iso-8859-2" and "utf8" becomes "utf-8".

        Note: "latin1" is also used when the encoding could not be detected.
```

This only works when editing files in the same encoding! When the actual character set is not latin1, make sure 'fileencoding' and 'fileencodings' are empty. When conversion is needed, switch to using utf-8.

When "unicode", "ucs-2" or "ucs-4" is used, Vim internally uses utf-8. You don't notice this while editing, but it does matter for the |viminfo-file|. And Vim expects the terminal to use utf-8 too. Thus setting 'encoding' to one of these values instead of utf-8 only has effect for encoding used for files when 'fileencoding' is empty.

When 'encoding' is set to a Unicode encoding, and 'fileencodings' was not set yet, the default for 'fileencodings' is changed.

'endofline' *'eol'* *'noendofline'* *'noeol'*
'endofline' 'eol' boolean (default on)
local to buffer
{not in Vi}

When writing a file and this option is off and the 'binary' option is on, or 'fixeol' option is off, no <EOL> will be written for the last line in the file. This option is automatically set or reset when starting to edit a new file, depending on whether file has an <EOL> for the last line in the file. Normally you don't have to set or reset this option.
When 'binary' is off and 'fixeol' is on the value is not used when writing the file. When 'binary' is on or 'fixeol' is off it is used to remember the presence of a <EOL> for the last line in the file, so that when you write the file the situation from the original file can be kept. But you can change it if you want to.

'equalalways' *'ea'* *'noequalalways'* *'noea'*
'equalalways' 'ea' boolean (default on)
global
{not in Vi}

When on, all the windows are automatically made the same size after splitting or closing a window. This also happens the moment the option is switched on. When off, splitting a window will reduce the size of the current window and leave the other windows the same. When closing a window the extra lines are given to the window next to it (depending on 'splitbelow' and 'splitright').
When mixing vertically and horizontally split windows, a minimal size is computed and some windows may be larger if there is room. The 'eadirection' option tells in which direction the size is affected. Changing the height and width of a window can be avoided by setting 'winfixheight' and 'winfixwidth', respectively.
If a window size is specified when creating a new window sizes are currently not equalized (it's complicated, but may be implemented in the future).

'equalprg' *'ep'*
'equalprg' 'ep' string (default "")
global or local to buffer |global-local|
{not in Vi}

External program to use for "=" command. When this option is empty the internal formatting functions are used; either 'lisp', 'cindent' or 'indentexpr'. When Vim was compiled without internal formatting, the "indent" program is used.
Environment variables are expanded |:set_env|. See |option-backslash| about including spaces and backslashes.
This option cannot be set from a |modeline| or in the |sandbox|, for security reasons.

```
                        *'errorbells'* *'eb'* *'noerrorbells'* *'noeb'*
'errorbells' 'eb'       boolean (default off)
                        global
        Ring the bell (beep or screen flash) for error messages.  This only
        makes a difference for error messages, the bell will be used always
        for a lot of errors without a message (e.g., hitting <Esc> in Normal
        mode).  See 'visualbell' on how to make the bell behave like a beep,
        screen flash or do nothing. See 'belloff' to finetune when to ring the
        bell.

                                                *'errorfile'* *'ef'*
'errorfile' 'ef'        string  (Amiga default: "AztecC.Err",
                                        others: "errors.err")
                        global
                        {not in Vi}
                        {not available when compiled without the |+quickfix|
                        feature}
        Name of the errorfile for the QuickFix mode (see |:cf|).
        When the "-q" command-line argument is used, 'errorfile' is set to the
        following argument.  See |-q|.
        NOT used for the ":make" command.  See 'makeef' for that.
        Environment variables are expanded |:set_env|.
        See |option-backslash| about including spaces and backslashes.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'errorformat'* *'efm'*
'errorformat' 'efm'     string  (default is very long)
                        global or local to buffer |global-local|
                        {not in Vi}
                        {not available when compiled without the |+quickfix|
                        feature}
        Scanf-like description of the format for the lines in the error file
        (see |errorformat|).

                                    *'esckeys'* *'ek'* *'noesckeys'* *'noek'*
'esckeys' 'ek'          boolean (Vim default: on, Vi default: off)
                        global
                        {not in Vi}
        Function keys that start with an <Esc> are recognized in Insert
        mode.  When this option is off, the cursor and function keys cannot be
        used in Insert mode if they start with an <Esc>.  The advantage of
        this is that the single <Esc> is recognized immediately, instead of
        after one second.  Instead of resetting this option, you might want to
        try changing the values for 'timeoutlen' and 'ttimeoutlen'.  Note that
        when 'esckeys' is off, you can still map anything, but the cursor keys
        won't work by default.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'eventignore'* *'ei'*
'eventignore' 'ei'      string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+autocmd|
                        feature}
        A list of autocommand event names, which are to be ignored.
        When set to "all" or when "all" is one of the items, all autocommand
        events are ignored, autocommands will not be executed.
        Otherwise this is a comma separated list of event names.  Example: >
            :set ei=WinEnter,WinLeave
<
```

```
                                *'expandtab'* *'et'* *'noexpandtab'* *'noet'*
'expandtab' 'et'        boolean (default off)
                        local to buffer
                        {not in Vi}
        In Insert mode: Use the appropriate number of spaces to insert a
        <Tab>.  Spaces are used in indents with the '>' and '<' commands and
        when 'autoindent' is on.  To insert a real tab when 'expandtab' is
        on, use CTRL-V<Tab>.  See also |:retab| and |ins-expandtab|.
        This option is reset when the 'paste' option is set and restored when
        the 'paste' option is reset.
        NOTE: This option is reset when 'compatible' is set.

                                            *'exrc'* *'ex'* *'noexrc'* *'noex'*
'exrc' 'ex'             boolean (default off)
                        global
                        {not in Vi}
        Enables the reading of .vimrc, .exrc and .gvimrc in the current
        directory.

        Setting this option is a potential security leak.  E.g., consider
        unpacking a package or fetching files from github, a .vimrc in there
        might be a trojan horse.  BETTER NOT SET THIS OPTION!
        Instead, define an autocommand in your .vimrc to set options for a
        matching directory.

        If you do switch this option on you should also consider setting the
        'secure' option (see |initialization|).
        Also see |.vimrc| and |gui-init|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                *'fileencoding'* *'fenc'* *E213*
'fileencoding' 'fenc'   string (default: "")
                        local to buffer
                        {only available when compiled with the |+multi_byte|
                        feature}
                        {not in Vi}
        Sets the character encoding for the file of this buffer.

        When 'fileencoding' is different from 'encoding', conversion will be
        done when writing the file.  For reading see below.
        When 'fileencoding' is empty, the same value as 'encoding' will be
        used (no conversion when reading or writing a file).
        No error will be given when the value is set, only when it is used,
        only when writing a file.
        Conversion will also be done when 'encoding' and 'fileencoding' are
        both a Unicode encoding and 'fileencoding' is not utf-8.  That's
        because internally Unicode is always stored as utf-8.
                WARNING: Conversion can cause loss of information!  When
                'encoding' is "utf-8" or another Unicode encoding, conversion
                is most likely done in a way that the reverse conversion
                results in the same text.  When 'encoding' is not "utf-8" some
                characters may be lost!

        See 'encoding' for the possible values.  Additionally, values may be
        specified that can be handled by the converter, see
        |mbyte-conversion|.

        When reading a file 'fileencoding' will be set from 'fileencodings'.
        To read a file in a certain encoding it won't work by setting
        'fileencoding', use the |++enc| argument.  One exception: when
        'fileencodings' is empty the value of 'fileencoding' is used.
```

```
        For a new file the global value of 'fileencoding' is used.

        Prepending "8bit-" and "2byte-" has no meaning here, they are ignored.
        When the option is set, the value is converted to lowercase.  Thus
        you can set it with uppercase values too.  '_' characters are
        replaced with '-'.  If a name is recognized from the list for
        'encoding', it is replaced by the standard name.  For example
        "ISO8859-2" becomes "iso-8859-2".

        When this option is set, after starting to edit a file, the 'modified'
        option is set, because the file would be different when written.

        Keep in mind that changing 'fenc' from a modeline happens
        AFTER the text has been read, thus it applies to when the file will be
        written.  If you do set 'fenc' in a modeline, you might want to set
        'nomodified' to avoid not being able to ":q".

        This option can not be changed when 'modifiable' is off.

                                                              *'fe'*
        NOTE: Before version 6.0 this option specified the encoding for the
        whole of Vim, this was a mistake.  Now use 'encoding' instead.  The
        old short name was 'fe', which is no longer used.

                                          *'fileencodings'* *'fencs'*
'fileencodings' 'fencs' string (default: "ucs-bom",
                                      "ucs-bom,utf-8,default,latin1" when
                                      'encoding' is set to a Unicode value)
                        global
                        {only available when compiled with the |+multi_byte|
                        feature}
                        {not in Vi}
        This is a list of character encodings considered when starting to edit
        an existing file.  When a file is read, Vim tries to use the first
        mentioned character encoding.  If an error is detected, the next one
        in the list is tried.  When an encoding is found that works,
        'fileencoding' is set to it.  If all fail, 'fileencoding' is set to
        an empty string, which means the value of 'encoding' is used.
                WARNING: Conversion can cause loss of information!  When
                'encoding' is "utf-8" (or one of the other Unicode variants)
                conversion is most likely done in a way that the reverse
                conversion results in the same text.  When 'encoding' is not
                "utf-8" some non-ASCII characters may be lost!  You can use
                the |++bad| argument to specify what is done with characters
                that can't be converted.
        For an empty file or a file with only ASCII characters most encodings
        will work and the first entry of 'fileencodings' will be used (except
        "ucs-bom", which requires the BOM to be present).  If you prefer
        another encoding use an BufReadPost autocommand event to test if your
        preferred encoding is to be used.  Example: >
                au BufReadPost * if search('\S', 'w') == 0 |
                        \ set fenc=iso-2022-jp | endif
<       This sets 'fileencoding' to "iso-2022-jp" if the file does not contain
        non-blank characters.
        When the |++enc| argument is used then the value of 'fileencodings' is
        not used.
        Note that 'fileencodings' is not used for a new file, the global value
        of 'fileencoding' is used instead.  You can set it with: >
                :setglobal fenc=iso-8859-2
<       This means that a non-existing file may get a different encoding than
        an empty file.
        The special value "ucs-bom" can be used to check for a Unicode BOM
```

```
        (Byte Order Mark) at the start of the file.  It must not be preceded
        by "utf-8" or another Unicode encoding for this to work properly.
        An entry for an 8-bit encoding (e.g., "latin1") should be the last,
        because Vim cannot detect an error, thus the encoding is always
        accepted.
        The special value "default" can be used for the encoding from the
        environment.  This is the default value for 'encoding'.  It is useful
        when 'encoding' is set to "utf-8" and your environment uses a
        non-latin1 encoding, such as Russian.
        When 'encoding' is "utf-8" and a file contains an illegal byte
        sequence it won't be recognized as UTF-8.  You can use the |8g8|
        command to find the illegal byte sequence.
        WRONG VALUES:                   WHAT'S WRONG:
                latin1,utf-8            "latin1" will always be used
                utf-8,ucs-bom,latin1    BOM won't be recognized in an utf-8
                                        file
                cp1250,latin1           "cp1250" will always be used
        If 'fileencodings' is empty, 'fileencoding' is not modified.
        See 'fileencoding' for the possible values.
        Setting this option does not have an effect until the next time a file
        is read.

                                        *'fileformat'* *'ff'*
'fileformat' 'ff'       string (MS-DOS, MS-Windows, OS/2 default: "dos",
                                Unix default: "unix",
                                Macintosh default: "mac")
                        local to buffer
                        {not in Vi}
        This gives the <EOL> of the current buffer, which is used for
        reading/writing the buffer from/to a file:
            dos     <CR> <NL>
            unix    <NL>
            mac     <CR>
        When "dos" is used, CTRL-Z at the end of a file is ignored.
        See |file-formats| and |file-read|.
        For the character encoding of the file see 'fileencoding'.
        When 'binary' is set, the value of 'fileformat' is ignored, file I/O
        works like it was set to "unix".
        This option is set automatically when starting to edit a file and
        'fileformats' is not empty and 'binary' is off.
        When this option is set, after starting to edit a file, the 'modified'
        option is set, because the file would be different when written.
        This option can not be changed when 'modifiable' is off.
        For backwards compatibility: When this option is set to "dos",
        'textmode' is set, otherwise 'textmode' is reset.

                                        *'fileformats'* *'ffs'*
'fileformats' 'ffs'     string (default:
                                Vim+Vi  MS-DOS, MS-Windows OS/2: "dos,unix",
                                Vim     Unix: "unix,dos",
                                Vim     Mac: "mac,unix,dos",
                                Vi      Cygwin: "unix,dos",
                                Vi      others: "")
                        global
                        {not in Vi}
        This gives the end-of-line (<EOL>) formats that will be tried when
        starting to edit a new buffer and when reading a file into an existing
        buffer:
        - When empty, the format defined with 'fileformat' will be used
          always.  It is not set automatically.
        - When set to one name, that format will be used whenever a new buffer
          is opened.  'fileformat' is set accordingly for that buffer.  The
```

```
          'fileformats' name will be used when a file is read into an existing
          buffer, no matter what 'fileformat' for that buffer is set to.
        - When more than one name is present, separated by commas, automatic
          <EOL> detection will be done when reading a file.  When starting to
          edit a file, a check is done for the <EOL>:
          1. If all lines end in <CR><NL>, and 'fileformats' includes "dos",
             'fileformat' is set to "dos".
          2. If a <NL> is found and 'fileformats' includes "unix", 'fileformat'
             is set to "unix".  Note that when a <NL> is found without a
             preceding <CR>, "unix" is preferred over "dos".
          3. If 'fileformat' has not yet been set, and if a <CR> is found, and
             if 'fileformats' includes "mac", 'fileformat' is set to "mac".
             This means that "mac" is only chosen when:
              "unix" is not present or no <NL> is found in the file, and
              "dos" is not present or no <CR><NL> is found in the file.
             Except: if "unix" was chosen, but there is a <CR> before
             the first <NL>, and there appear to be more <CR>s than <NL>s in
             the first few lines, "mac" is used.
          4. If 'fileformat' is still not set, the first name from
             'fileformats' is used.
          When reading a file into an existing buffer, the same is done, but
          this happens like 'fileformat' has been set appropriately for that
          file only, the option is not changed.
        When 'binary' is set, the value of 'fileformats' is not used.

        When Vim starts up with an empty buffer the first item is used.  You
        can overrule this by setting 'fileformat' in your .vimrc.

        For systems with a Dos-like <EOL> (<CR><NL>), when reading files that
        are ":source"ed and for vimrc files, automatic <EOL> detection may be
        done:
        - When 'fileformats' is empty, there is no automatic detection.  Dos
          format will be used.
        - When 'fileformats' is set to one or more names, automatic detection
          is done.  This is based on the first <NL> in the file: If there is a
          <CR> in front of it, Dos format is used, otherwise Unix format is
          used.
        Also see |file-formats|.
        For backwards compatibility: When this option is set to an empty
        string or one format (no comma is included), 'textauto' is reset,
        otherwise 'textauto' is set.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                *'fileignorecase'* *'fic'* *'nofileignorecase'* *'nofic'*
'fileignorecase' 'fic'  boolean (default on for systems where case in file
                                 names is normally ignored)
                        global
                        {not in Vi}
        When set case is ignored when using file names and directories.
        See 'wildignorecase' for only ignoring case when doing completion.

                                        *'filetype'* *'ft'*
'filetype' 'ft'         string (default: "")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+autocmd|
                        feature}
        When this option is set, the FileType autocommand event is triggered.
        All autocommands that match with the value of this option will be
        executed.  Thus the value of 'filetype' is used in place of the file
        name.
```

```
        Otherwise this option does not always reflect the current file type.
        This option is normally set when the file type is detected.  To enable
        this use the ":filetype on" command. |:filetype|
        Setting this option to a different value is most useful in a modeline,
        for a file for which the file type is not automatically recognized.
        Example, for in an IDL file:
                /* vim: set filetype=idl : */ ~
        |FileType| |filetypes|
        When a dot appears in the value then this separates two filetype
        names.  Example:
                /* vim: set filetype=c.doxygen : */ ~
        This will use the "c" filetype first, then the "doxygen" filetype.
        This works both for filetype plugins and for syntax files.  More than
        one dot may appear.
        This option is not copied to another buffer, independent of the 's' or
        'S' flag in 'cpoptions'.
        Only normal file name characters can be used, "/\*?[|<>" are illegal.

                                                *'fillchars'* *'fcs'*
'fillchars' 'fcs'       string  (default "vert:|,fold:-")
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        and |+folding| features}
        Characters to fill the statuslines and vertical separators.
        It is a comma separated list of items:

          item          default         Used for ~
          stl:c         ' ' or '^'      statusline of the current window
          stlnc:c       ' ' or '='      statusline of the non-current windows
          vert:c        '|'             vertical separators |:vsplit|
          fold:c        '-'             filling 'foldtext'
          diff:c        '-'             deleted lines of the 'diff' option

        Any one that is omitted will fall back to the default.  For "stl" and
        "stlnc" the space will be used when there is highlighting, '^' or '='
        otherwise.

        Example: >
            :set fillchars=stl:^,stlnc:=,vert:\|,fold:-,diff:-
<       This is similar to the default, except that these characters will also
        be used when there is highlighting.

        for "stl" and "stlnc" only single-byte values are supported.

        The highlighting used for these items:
          item          highlight group ~
          stl:c         StatusLine              |hl-StatusLine|
          stlnc:c       StatusLineNC            |hl-StatusLineNC|
          vert:c        VertSplit               |hl-VertSplit|
          fold:c        Folded                  |hl-Folded|
          diff:c        DiffDelete              |hl-DiffDelete|

                *'fixendofline'* *'fixeol'* *'nofixendofline'* *'nofixeol'*
'fixendofline' 'fixeol' boolean (default on)
                        local to buffer
                        {not in Vi}
        When writing a file and this option is on, <EOL> at the end of file
        will be restored if missing. Turn this option off if you want to
        preserve the situation from the original file.
        When the 'binary' option is set the value of this option doesn't
        matter.
```

```
        See the 'endofline' option.

                                          *'fkmap'* *'fk'* *'nofkmap'* *'nofk'*
'fkmap' 'fk'            boolean (default off)                   *E198*
                        global
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        When on, the keyboard is mapped for the Farsi character set.
        Normally you would set 'allowrevins' and use CTRL-_ in insert mode to
        toggle this option |i_CTRL-_|.  See |farsi.txt|.

                                                  *'foldclose'* *'fcl'*
'foldclose' 'fcl'       string (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        When set to "all", a fold is closed when the cursor isn't in it and
        its level is higher than 'foldlevel'.  Useful if you want folds to
        automatically close when moving out of them.

                                                  *'foldcolumn'* *'fdc'*
'foldcolumn' 'fdc'      number (default 0)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        When non-zero, a column with the specified width is shown at the side
        of the window which indicates open and closed folds.  The maximum
        value is 12.
        See |folding|.

                        *'foldenable'* *'fen'* *'nofoldenable'* *'nofen'*
'foldenable' 'fen'      boolean (default on)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        When off, all folds are open.  This option can be used to quickly
        switch between showing all text unfolded and viewing the text with
        folds (including manually opened or closed folds).  It can be toggled
        with the |zi| command.  The 'foldcolumn' will remain blank when
        'foldenable' is off.
        This option is set by commands that create a new fold or close a fold.
        See |folding|.

                                                  *'foldexpr'* *'fde'*
'foldexpr' 'fde'        string (default: "0")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        or |+eval| features}
        The expression used for when 'foldmethod' is "expr".  It is evaluated
        for each line to obtain its fold level.  See |fold-expr|.

        The expression will be evaluated in the |sandbox| if set from a
        modeline, see |sandbox-option|.
        This option can't be set from a |modeline| when the 'diff' option is
        on.

        It is not allowed to change text or jump to another window while
```

```
        evaluating 'foldexpr' |textlock|.

                                                *'foldignore'* *'fdi'*
'foldignore' 'fdi'      string (default: "#")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Used only when 'foldmethod' is "indent".  Lines starting with
        characters in 'foldignore' will get their fold level from surrounding
        lines.  White space is skipped before checking for this character.
        The default "#" works well for C programs.  See |fold-indent|.

                                                *'foldlevel'* *'fdl'*
'foldlevel' 'fdl'       number (default: 0)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Sets the fold level: Folds with a higher level will be closed.
        Setting this option to zero will close all folds.  Higher numbers will
        close fewer folds.
        This option is set by commands like |zm|, |zM| and |zR|.
        See |fold-foldlevel|.

                                                *'foldlevelstart'* *'fdls'*
'foldlevelstart' 'fdls' number (default: -1)
                        global
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Sets 'foldlevel' when starting to edit another buffer in a window.
        Useful to always start editing with all folds closed (value zero),
        some folds closed (one) or no folds closed (99).
        This is done before reading any modeline, thus a setting in a modeline
        overrules this option.  Starting to edit a file for |diff-mode| also
        ignores this option and closes all folds.
        It is also done before BufReadPre autocommands, to allow an autocmd to
        overrule the 'foldlevel' value for specific files.
        When the value is negative, it is not used.

                                                *'foldmarker'* *'fmr'* *E536*
'foldmarker' 'fmr'      string (default: "{{{,}}}")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        The start and end marker used when 'foldmethod' is "marker".  There
        must be one comma, which separates the start and end marker.  The
        marker is a literal string (a regular expression would be too slow).
        See |fold-marker|.

                                                *'foldmethod'* *'fdm'*
'foldmethod' 'fdm'      string (default: "manual")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        The kind of folding used for the current window.  Possible values:
        |fold-manual|   manual      Folds are created manually.
        |fold-indent|   indent      Lines with equal indent form a fold.
        |fold-expr|     expr        'foldexpr' gives the fold level of a line.
```

```
        |fold-marker|   marker      Markers are used to specify folds.
        |fold-syntax|   syntax      Syntax highlighting items specify folds.
        |fold-diff|     diff        Fold text that is not changed.

                                                *'foldminlines'* *'fml'*
'foldminlines' 'fml'    number (default: 1)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Sets the number of screen lines above which a fold can be displayed
        closed.  Also for manually closed folds.  With the default value of
        one a fold can only be closed if it takes up two or more screen lines.
        Set to zero to be able to close folds of just one screen line.
        Note that this only has an effect on what is displayed.  After using
        "zc" to close a fold, which is displayed open because it's smaller
        than 'foldminlines', a following "zc" may close a containing fold.

                                                *'foldnestmax'* *'fdn'*
'foldnestmax' 'fdn'     number (default: 20)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Sets the maximum nesting of folds for the "indent" and "syntax"
        methods.  This avoids that too many folds will be created.  Using more
        than 20 doesn't work, because the internal limit is 20.

                                                *'foldopen'* *'fdo'*
'foldopen' 'fdo'        string (default: "block,hor,mark,percent,quickfix,
                                                          search,tag,undo")
                        global
                        {not in Vi}
                        {not available when compiled without the |+folding|
                        feature}
        Specifies for which type of commands folds will be opened, if the
        command moves the cursor into a closed fold.  It is a comma separated
        list of items.
        NOTE: When the command is part of a mapping this option is not used.
        Add the |zv| command to the mapping to get the same effect.
        (rationale: the mapping may want to control opening folds itself)

                item            commands ~
                all             any
                block           "(", "{", "[[", "[{", etc.
                hor             horizontal movements: "l", "w", "fx", etc.
                insert          any command in Insert mode
                jump            far jumps: "G", "gg", etc.
                mark            jumping to a mark: "'m", CTRL-O, etc.
                percent         "%"
                quickfix        ":cn", ":crew", ":make", etc.
                search          search for a pattern: "/", "n", "*", "gd", etc.
                                (not for a search pattern in a ":" command)
                                Also for |[s| and |]s|.
                tag             jumping to a tag: ":ta", CTRL-T, etc.
                undo            undo or redo: "u" and CTRL-R
        When a movement command is used for an operator (e.g., "dl" or "y%")
        this option is not used.  This means the operator will include the
        whole closed fold.
        Note that vertical movements are not here, because it would make it
        very difficult to move onto a closed fold.
        In insert mode the folds containing the cursor will always be open
```

when text is inserted.
To close folds you can re-apply 'foldlevel' with the |zx| command or
set the 'foldclose' option to "all".

'foldtext' *'fdt'*
'foldtext' 'fdt' string (default: "foldtext()")
local to window
{not in Vi}
{not available when compiled without the |+folding|
feature}
An expression which is used to specify the text displayed for a closed
fold. See |fold-foldtext|.

The expression will be evaluated in the |sandbox| if set from a
modeline, see |sandbox-option|.

It is not allowed to change text or jump to another window while
evaluating 'foldtext' |textlock|.

'formatexpr' *'fex'*
'formatexpr' 'fex' string (default "")
local to buffer
{not in Vi}
{not available when compiled without the |+eval|
feature}
Expression which is evaluated to format a range of lines for the |gq|
operator or automatic formatting (see 'formatoptions'). When this
option is empty 'formatprg' is used.

The |v:lnum| variable holds the first line to be formatted.
The |v:count| variable holds the number of lines to be formatted.
The |v:char| variable holds the character that is going to be
inserted if the expression is being evaluated due to
automatic formatting. This can be empty. Don't insert
it yet!

Example: >
:set formatexpr=mylang#Format()
< This will invoke the mylang#Format() function in the
autoload/mylang.vim file in 'runtimepath'. |autoload|

The expression is also evaluated when 'textwidth' is set and adding
text beyond that limit. This happens under the same conditions as
when internal formatting is used. Make sure the cursor is kept in the
same spot relative to the text then! The |mode()| function will
return "i" or "R" in this situation.

When the expression evaluates to non-zero Vim will fall back to using
the internal format mechanism.

The expression will be evaluated in the |sandbox| when set from a
modeline, see |sandbox-option|. That stops the option from working,
since changing the buffer text is not allowed.
NOTE: This option is set to "" when 'compatible' is set.

'formatoptions' *'fo'*
'formatoptions' 'fo' string (Vim default: "tcq", Vi default: "vt")
local to buffer
{not in Vi}
This is a sequence of letters which describes how automatic
formatting is to be done. See |fo-table|. When the 'paste' option is
on, no formatting is done (like 'formatoptions' is empty). Commas can

be inserted for readability.
To avoid problems with flags that are added in the future, use the
"+=" and "-=" feature of ":set" |add-option-flags|.
NOTE: This option is set to the Vi default value when 'compatible' is
set and to the Vim default value when 'compatible' is reset.

'formatlistpat' *'flp'*
'formatlistpat' 'flp' string (default: "^\s*\d\+[\]:.)}\t]\s*")
local to buffer
{not in Vi}
A pattern that is used to recognize a list header. This is used for
the "n" flag in 'formatoptions'.
The pattern must match exactly the text that will be the indent for
the line below it. You can use |/\ze| to mark the end of the match
while still checking more characters. There must be a character
following the pattern, when it matches the whole line it is handled
like there is no match.
The default recognizes a number, followed by an optional punctuation
character and white space.

'formatprg' *'fp'*
'formatprg' 'fp' string (default "")
global or local to buffer |global-local|
{not in Vi}
The name of an external program that will be used to format the lines
selected with the |gq| operator. The program must take the input on
stdin and produce the output on stdout. The Unix program "fmt" is
such a program.
If the 'formatexpr' option is not empty it will be used instead.
Otherwise, if 'formatprg' option is an empty string, the internal
format function will be used |C-indenting|.
Environment variables are expanded |:set_env|. See |option-backslash|
about including spaces and backslashes.
This option cannot be set from a |modeline| or in the |sandbox|, for
security reasons.

'fsync' *'fs'* *'nofsync'* *'nofs'*
'fsync' 'fs' boolean (default on)
global
{not in Vi}
When on, the library function fsync() will be called after writing a
file. This will flush a file to disk, ensuring that it is safely
written even on filesystems which do metadata-only journaling. This
will force the harddrive to spin up on Linux systems running in laptop
mode, so it may be undesirable in some situations. Be warned that
turning this off increases the chances of data loss after a crash. On
systems without an fsync() implementation, this variable is always
off.
Also see 'swapsync' for controlling fsync() on swap files.

'gdefault' *'gd'* *'nogdefault'* *'nogd'*
'gdefault' 'gd' boolean (default off)
global
{not in Vi}
When on, the ":substitute" flag 'g' is default on. This means that
all matches in a line are substituted instead of one. When a 'g' flag
is given to a ":substitute" command, this will toggle the substitution
of all or one match. See |complex-change|.

command 'gdefault' on 'gdefault' off ~
:s/// subst. all subst. one
:s///g subst. one subst. all

```
                :s///gg           subst. all      subst. one

        NOTE: This option is reset when 'compatible' is set.
        DEPRECATED: Setting this option may break plugins that are not aware
        of this option.  Also, many users get confused that adding the /g flag
        has the opposite effect of that it normally does.

                                                *'grepformat'* *'gfm'*
'grepformat' 'gfm'      string  (default "%f:%l:%m,%f:%l%m,%f  %l%m")
                        global
                        {not in Vi}
        Format to recognize for the ":grep" command output.
        This is a scanf-like string that uses the same format as the
        'errorformat' option: see |errorformat|.

                                                *'grepprg'* *'gp'*
'grepprg' 'gp'          string  (default "grep -n ",
                                        Unix: "grep -n $* /dev/null",
                                        Win32: "findstr /n" or "grep -n",
                                                VMS: "SEARCH/NUMBERS ")
                        global or local to buffer |global-local|
                        {not in Vi}
        Program to use for the |:grep| command.  This option may contain '%'
        and '#' characters, which are expanded like when used in a command-
        line.  The placeholder "$*" is allowed to specify where the arguments
        will be included.  Environment variables are expanded |:set_env|.  See
        |option-backslash| about including spaces and backslashes.
        When your "grep" accepts the "-H" argument, use this to make ":grep"
        also work well with a single file: >
                :set grepprg=grep\ -nH
<       Special value: When 'grepprg' is set to "internal" the |:grep| command
        works like |:vimgrep|, |:lgrep| like |:lvimgrep|, |:grepadd| like
        |:vimgrepadd| and |:lgrepadd| like |:lvimgrepadd|.
        See also the section |:make_makeprg|, since most of the comments there
        apply equally to 'grepprg'.
        For Win32, the default is "findstr /n" if "findstr.exe" can be found,
        otherwise it's "grep -n".
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                        *'guicursor'* *'gcr'* *E545* *E546* *E548* *E549*
'guicursor' 'gcr'       string  (default "n-v-c:block-Cursor/lCursor,
                                        ve:ver35-Cursor,
                                        o:hor50-Cursor,
                                        i-ci:ver25-Cursor/lCursor,
                                        r-cr:hor20-Cursor/lCursor,
                                        sm:block-Cursor
                                        -blinkwait175-blinkoff150-blinkon175",
                                for MS-DOS and Win32 console:
                                        "n-v-c:block,o:hor50,i-ci:hor15,
                                        r-cr:hor30,sm:block")
                        global
                        {not in Vi}
                        {only available when compiled with GUI enabled, and
                        for MS-DOS and Win32 console}
        This option tells Vim what the cursor should look like in different
        modes.  It fully works in the GUI.  In an MSDOS or Win32 console, only
        the height of the cursor can be changed.  This can be done by
        specifying a block cursor, or a percentage for a vertical or
        horizontal cursor.
        For a console the 't_SI', 't_SR', and 't_EI' escape sequences are
        used.
```

```
The option is a comma separated list of parts.  Each part consist of a
mode-list and an argument-list:
        mode-list:argument-list,mode-list:argument-list,..
The mode-list is a dash separated list of these modes:
        n       Normal mode
        v       Visual mode
        ve      Visual mode with 'selection' "exclusive" (same as 'v',
                if not specified)
        o       Operator-pending mode
        i       Insert mode
        r       Replace mode
        c       Command-line Normal (append) mode
        ci      Command-line Insert mode
        cr      Command-line Replace mode
        sm      showmatch in Insert mode
        a       all modes
The argument-list is a dash separated list of these arguments:
        hor{N}  horizontal bar, {N} percent of the character height
        ver{N}  vertical bar, {N} percent of the character width
        block   block cursor, fills the whole character
                [only one of the above three should be present]
        blinkwait{N}                            *cursor-blinking*
        blinkon{N}
        blinkoff{N}
                blink times for cursor: blinkwait is the delay before
                the cursor starts blinking, blinkon is the time that
                the cursor is shown and blinkoff is the time that the
                cursor is not shown.  The times are in msec.  When one
                of the numbers is zero, there is no blinking.  The
                default is: "blinkwait700-blinkon400-blinkoff250".
                These numbers are used for a missing entry.  This
                means that blinking is enabled by default.  To switch
                blinking off you can use "blinkon0".  The cursor only
                blinks when Vim is waiting for input, not while
                executing a command.
                To make the cursor blink in an xterm, see
                |xterm-blink|.
        {group-name}
                a highlight group name, that sets the color and font
                for the cursor
        {group-name}/{group-name}
                Two highlight group names, the first is used when
                no language mappings are used, the other when they
                are. |language-mapping|

Examples of parts:
   n-c-v:block-nCursor  in Normal, Command-line and Visual mode, use a
                        block cursor with colors from the "nCursor"
                        highlight group
   i-ci:ver30-iCursor-blinkwait300-blinkon200-blinkoff150
                        In Insert and Command-line Insert mode, use a
                        30% vertical bar cursor with colors from the
                        "iCursor" highlight group.  Blink a bit
                        faster.

The 'a' mode is different.  It will set the given argument-list for
all modes.  It does not reset anything to defaults.  This can be used
to do a common setting for all modes.  For example, to switch off
blinking: "a:blinkon0"

Examples of cursor highlighting: >
```

```
            :highlight Cursor gui=reverse guifg=NONE guibg=NONE
            :highlight Cursor gui=NONE guifg=bg guibg=fg
<
                                        *'guifont'* *'gfn'*
                                                    *E235* *E596*
'guifont' 'gfn'         string  (default "")
                        global
                        {not in Vi}
                        {only available when compiled with GUI enabled}
        This is a list of fonts which will be used for the GUI version of Vim.
        In its simplest form the value is just one font name.  When
        the font cannot be found you will get an error message.  To try other
        font names a list can be specified, font names separated with commas.
        The first valid font is used.

        On systems where 'guifontset' is supported (X11) and 'guifontset' is
        not empty, then 'guifont' is not used.

        Note: As to the GTK GUIs, no error is given against any invalid names,
        and the first element of the list is always picked up and made use of.
        This is because, instead of identifying a given name with a font, the
        GTK GUIs use it to construct a pattern and try to look up a font which
        best matches the pattern among available fonts, and this way, the
        matching never fails.  An invalid name doesn't matter because a number
        of font properties other than name will do to get the matching done.

        Spaces after a comma are ignored.  To include a comma in a font name
        precede it with a backslash.  Setting an option requires an extra
        backslash before a space and a backslash.  See also
        |option-backslash|.  For example: >
            :set guifont=Screen15,\ 7x13,font\\,with\\,commas
<       will make Vim try to use the font "Screen15" first, and if it fails it
        will try to use "7x13" and then "font,with,commas" instead.

        If none of the fonts can be loaded, Vim will keep the current setting.
        If an empty font list is given, Vim will try using other resource
        settings (for X, it will use the Vim.font resource), and finally it
        will try some builtin default which should always be there ("7x13" in
        the case of X).  The font names given should be "normal" fonts.  Vim
        will try to find the related bold and italic fonts.

        For Win32, GTK, Motif, Mac OS and Photon: >
            :set guifont=*
<       will bring up a font requester, where you can pick the font you want.

        The font name depends on the GUI used.  See |setting-guifont| for a
        way to set 'guifont' for various systems.

        For the GTK+ 2 and 3 GUIs, the font name looks like this: >
            :set guifont=Andale\ Mono\ 11
<       That's all.  XLFDs are not used.  For Chinese this is reported to work
        well: >
            if has("gui_gtk2")
              set guifont=Bitstream\ Vera\ Sans\ Mono\ 12,Fixed\ 12
              set guifontwide=Microsoft\ Yahei\ 12,WenQuanYi\ Zen\ Hei\ 12
            endif
<
        (Replace gui_gtk2 with gui_gtk3 for the GTK+ 3 GUI)

        For Mac OSX you can use something like this: >
            :set guifont=Monaco:h10
<       Also see 'macatsui', it can help fix display problems.
```

E236
Note that the fonts must be mono-spaced (all characters have the same width). An exception is GTK: all fonts are accepted, but mono-spaced fonts look best.

To preview a font on X11, you might be able to use the "xfontsel" program. The "xlsfonts" program gives a list of all available fonts.

For the Win32 GUI *E244* *E245*
- takes these options in the font name:
  - hXX - height is XX (points, can be floating-point)
  - wXX - width is XX (points, can be floating-point)
  - b - bold
  - i - italic
  - u - underline
  - s - strikeout
  - cXX - character set XX. Valid charsets are: ANSI, ARABIC, BALTIC, CHINESEBIG5, DEFAULT, EASTEUROPE, GB2312, GREEK, HANGEUL, HEBREW, JOHAB, MAC, OEM, RUSSIAN, SHIFTJIS, SYMBOL, THAI, TURKISH, VIETNAMESE ANSI and BALTIC. Normally you would use "cDEFAULT".
  - qXX - quality XX. Valid quality names are: PROOF, DRAFT, ANTIALIASED, NONANTIALIASED, CLEARTYPE, DEFAULT. Normally you would use "qDEFAULT". Some quality values are not supported in legacy OSs.

  Use a ':' to separate the options.
- A '_' can be used in the place of a space, so you don't need to use backslashes to escape the spaces.
- Examples: >
```
:set guifont=courier_new:h12:w5:b:cRUSSIAN
:set guifont=Andale_Mono:h7.5:w4.5
```
< See also |font-sizes|.

'guifontset' *'gfs'*
E250 *E252* *E234* *E597* *E598*
'guifontset' 'gfs' string (default "")
global
{not in Vi}
{only available when compiled with GUI enabled and with the |+xfontset| feature}
{not available in the GTK+ GUI}

When not empty, specifies two (or more) fonts to be used. The first one for normal English, the second one for your special language. See |xfontset|.
Setting this option also means that all font names will be handled as a fontset name. Also the ones used for the "font" argument of the |:highlight| command.
The fonts must match with the current locale. If fonts for the character sets that the current locale uses are not included, setting 'guifontset' will fail.
Note the difference between 'guifont' and 'guifontset': In 'guifont' the comma-separated names are alternative names, one of which will be used. In 'guifontset' the whole string is one fontset name, including the commas. It is not possible to specify alternative fontset names.
This example works on many X11 systems: >
```
:set guifontset=-*-*-medium-r-normal--16-*-*-*-c-*-*-*
```
<

'guifontwide' *'gfw'* *E231* *E533* *E534*
'guifontwide' 'gfw' string (default "")
global

```
                        {not in Vi}
                        {only available when compiled with GUI enabled}
        When not empty, specifies a comma-separated list of fonts to be used
        for double-width characters.  The first font that can be loaded is
        used.
        Note: The size of these fonts must be exactly twice as wide as the one
        specified with 'guifont' and the same height.

        All GUI versions but GTK+:

        'guifontwide' is only used when 'encoding' is set to "utf-8" and
        'guifontset' is empty or invalid.
        When 'guifont' is set and a valid font is found in it and
        'guifontwide' is empty Vim will attempt to find a matching
        double-width font and set 'guifontwide' to it.

        GTK+ GUI only:                          *guifontwide_gtk*

        If set and valid, 'guifontwide' is always used for double width
        characters, even if 'encoding' is not set to "utf-8".
        Vim does not attempt to find an appropriate value for 'guifontwide'
        automatically.  If 'guifontwide' is empty Pango/Xft will choose the
        font for characters not available in 'guifont'.  Thus you do not need
        to set 'guifontwide' at all unless you want to override the choice
        made by Pango/Xft.

        Windows +multibyte only:                *guifontwide_win_mbyte*

        If set and valid, 'guifontwide' is used for IME instead of 'guifont'.

                                                *'guiheadroom'* *'ghr'*
'guiheadroom' 'ghr'     number  (default 50)
                        global
                        {not in Vi} {only for GTK and X11 GUI}
        The number of pixels subtracted from the screen height when fitting
        the GUI window on the screen.  Set this before the GUI is started,
        e.g., in your |gvimrc| file.  When zero, the whole screen height will
        be used by the window.  When positive, the specified number of pixel
        lines will be left for window decorations and other items on the
        screen.  Set it to a negative value to allow windows taller than the
        screen.

                                                *'guioptions'* *'go'*
'guioptions' 'go'       string  (default "egmrLtT"   (MS-Windows, "t" is
                                                        removed in |defaults.vim|),
                                        "aegimrLtT" (GTK, Motif and Athena),
                                        )
                        global
                        {not in Vi}
                        {only available when compiled with GUI enabled}
        This option only has an effect in the GUI version of Vim.  It is a
        sequence of letters which describes what components and options of the
        GUI should be used.
        To avoid problems with flags that are added in the future, use the
        "+=" and "-=" feature of ":set" |add-option-flags|.

        Valid letters are as follows:
                                                        *guioptions_a* *'go-a'*
          'a'   Autoselect:  If present, then whenever VISUAL mode is started,
                or the Visual area extended, Vim tries to become the owner of
                the windowing system's global selection.  This means that the
                Visually highlighted text is available for pasting into other
```

applications as well as into Vim itself. When the Visual mode ends, possibly due to an operation on the text, or when an application wants to paste the selection, the highlighted text is automatically yanked into the "* selection register.
Thus the selection is still available for pasting into other applications after the VISUAL mode has ended.
If not present, then Vim won't become the owner of the windowing system's global selection unless explicitly told to by a yank or delete operation for the "* register.
The same applies to the modeless selection.
'go-P'

'P' Like autoselect but using the "+ register instead of the "* register.
'go-A'

'A' Autoselect for the modeless selection. Like 'a', but only applies to the modeless selection.

'guioptions' autoselect Visual autoselect modeless ~
"" - -
"a" yes yes
"A" - yes
"aA" yes yes

'go-c'

'c' Use console dialogs instead of popup dialogs for simple choices.
'go-e'

'e' Add tab pages when indicated with 'showtabline'.
'guitablabel' can be used to change the text in the labels.
When 'e' is missing a non-GUI tab pages line may be used.
The GUI tabs are only supported on some systems, currently GTK, Motif, Mac OS/X and MS-Windows.
'go-f'

'f' Foreground: Don't use fork() to detach the GUI from the shell where it was started. Use this for programs that wait for the editor to finish (e.g., an e-mail program). Alternatively you can use "gvim -f" or ":gui -f" to start the GUI in the foreground. |gui-fork|
Note: Set this option in the vimrc file. The forking may have happened already when the |gvimrc| file is read.
'go-i'

'i' Use a Vim icon. For GTK with KDE it is used in the left-upper corner of the window. It's black&white on non-GTK, because of limitations of X11. For a color icon, see |X11-icon|.
'go-m'

'm' Menu bar is present.
'go-M'

'M' The system menu "$VIMRUNTIME/menu.vim" is not sourced. Note that this flag must be added in the .vimrc file, before switching on syntax or filetype recognition (when the |gvimrc| file is sourced the system menu has already been loaded; the ":syntax on" and ":filetype on" commands load the menu too).
'go-g'

'g' Grey menu items: Make menu items that are not active grey. If 'g' is not included inactive menu items are not shown at all.
Exception: Athena will always use grey menu items.
'go-t'

't' Include tearoff menu items. Currently only works for Win32, GTK+, and Motif 1.2 GUI.
'go-T'

'T' Include Toolbar. Currently only in Win32, GTK+, Motif, Photon and Athena GUIs.

'go-r'
'r' Right-hand scrollbar is always present.
'go-R'
'R' Right-hand scrollbar is present when there is a vertically split window.
'go-l'
'l' Left-hand scrollbar is always present.
'go-L'
'L' Left-hand scrollbar is present when there is a vertically split window.
'go-b'
'b' Bottom (horizontal) scrollbar is present. Its size depends on the longest visible line, or on the cursor line if the 'h' flag is included. |gui-horiz-scroll|
'go-h'
'h' Limit horizontal scrollbar size to the length of the cursor line. Reduces computations. |gui-horiz-scroll|

And yes, you may even have scrollbars on the left AND the right if you really want to :-). See |gui-scrollbars| for more information.

'go-v'
'v' Use a vertical button layout for dialogs. When not included, a horizontal layout is preferred, but when it doesn't fit a vertical layout is used anyway.
'go-p'
'p' Use Pointer callbacks for X11 GUI. This is required for some window managers. If the cursor is not blinking or hollow at the right moment, try adding this flag. This must be done before starting the GUI. Set it in your |gvimrc|. Adding or removing it after the GUI has started has no effect.
'go-F'
'F' Add a footer. Only for Motif. See |gui-footer|.

'guipty' *'noguipty'*
'guipty' boolean (default on)
global
{not in Vi}
{only available when compiled with GUI enabled}
Only in the GUI: If on, an attempt is made to open a pseudo-tty for I/O to/from shell commands. See |gui-pty|.

'guitablabel' *'gtl'*
'guitablabel' 'gtl' string (default empty)
global
{not in Vi}
{only available when compiled with GUI enabled and with the |+windows| feature}
When nonempty describes the text to use in a label of the GUI tab pages line. When empty and when the result is empty Vim will use a default label. See |setting-guitablabel| for more info.

The format of this option is like that of 'statusline'.
'guitabtooltip' is used for the tooltip, see below.
The expression will be evaluated in the |sandbox| when set from a modeline, see |sandbox-option|.

Only used when the GUI tab pages line is displayed. 'e' must be present in 'guioptions'. For the non-GUI tab pages line 'tabline' is used.

```
                                                *'guitabtooltip'* *'gtt'*
'guitabtooltip' 'gtt'   string  (default empty)
                        global
                        {not in Vi}
                        {only available when compiled with GUI enabled and
                        with the |+windows| feature}
        When nonempty describes the text to use in a tooltip for the GUI tab
        pages line.  When empty Vim will use a default tooltip.
        This option is otherwise just like 'guitablabel' above.
        You can include a line break.  Simplest method is to use |:let|: >
                :let &guitabtooltip = "line one\nline two"
<

                                                *'helpfile'* *'hf'*
'helpfile' 'hf'         string  (default (MSDOS)  "$VIMRUNTIME\doc\help.txt"
                                        (others) "$VIMRUNTIME/doc/help.txt")
                        global
                        {not in Vi}
        Name of the main help file.  All distributed help files should be
        placed together in one directory.  Additionally, all "doc" directories
        in 'runtimepath' will be used.
        Environment variables are expanded |:set_env|.  For example:
        "$VIMRUNTIME/doc/help.txt".  If $VIMRUNTIME is not set, $VIM is also
        tried.  Also see |$VIMRUNTIME| and |option-backslash| about including
        spaces and backslashes.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'helpheight'* *'hh'*
'helpheight' 'hh'       number  (default 20)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        Minimal initial height of the help window when it is opened with the
        ":help" command.  The initial height of the help window is half of the
        current window, or (when the 'ea' option is on) the same as other
        windows.  When the height is less than 'helpheight', the height is
        set to 'helpheight'.  Set to zero to disable.

                                                *'helplang'* *'hlg'*
'helplang' 'hlg'        string  (default: messages language or empty)
                        global
                        {only available when compiled with the |+multi_lang|
                        feature}
                        {not in Vi}
        Comma separated list of languages.  Vim will use the first language
        for which the desired help can be found.  The English help will always
        be used as a last resort.  You can add "en" to prefer English over
        another language, but that will only find tags that exist in that
        language and not in the English help.
        Example: >
                :set helplang=de,it
<       This will first search German, then Italian and finally English help
        files.
        When using |CTRL-]| and ":help!" in a non-English help file Vim will
        try to find the tag in the current language before using this option.
        See |help-translated|.

                                        *'hidden'* *'hid'* *'nohidden'* *'nohid'*
'hidden' 'hid'          boolean (default off)
                        global
```

```
                    {not in Vi}
        When off a buffer is unloaded when it is |abandon|ed.  When on a
        buffer becomes hidden when it is |abandon|ed.  If the buffer is still
        displayed in another window, it does not become hidden, of course.
        The commands that move through the buffer list sometimes make a buffer
        hidden although the 'hidden' option is off: When the buffer is
        modified, 'autowrite' is off or writing is not possible, and the '!'
        flag was used.  See also |windows.txt|.
        To only make one buffer hidden use the 'bufhidden' option.
        This option is set for one command with ":hide {command}" |:hide|.
        WARNING: It's easy to forget that you have changes in hidden buffers.
        Think twice when using ":q!" or ":qa!".

                                                *'highlight'* *'hl'*
'highlight' 'hl'        string  (default (as a single string):
                                    "8:SpecialKey,~:EndOfBuffer,@:NonText,
                                    d:Directory,e:ErrorMsg,i:IncSearch,
                                    l:Search,m:MoreMsg,M:ModeMsg,n:LineNr,
                                    N:CursorLineNr,r:Question,s:StatusLine,
                                    S:StatusLineNC,c:VertSplit,t:Title,
                                    v:Visual,w:WarningMsg,W:WildMenu,f:Folded,
                                    F:FoldColumn,A:DiffAdd,C:DiffChange,
                                    D:DiffDelete,T:DiffText,>:SignColumn,
                                    B:SpellBad,P:SpellCap,R:SpellRare,
                                    L:SpellLocal,-:Conceal,+:Pmenu,=:PmenuSel,
                                    x:PmenuSbar,X:PmenuThumb,*:TabLine,
                                    #:TabLineSel,_:TabLineFill,!:CursorColumn,
                                    .:CursorLine,o:ColorColumn,q:QuickFixLine,
                                    z:StatusLineTerm,Z:StatusLineTermNC")
                    global
                    {not in Vi}
        This option can be used to set highlighting mode for various
        occasions.  It is a comma separated list of character pairs.  The
        first character in a pair gives the occasion, the second the mode to
        use for that occasion.  The occasions are:
        |hl-SpecialKey|  8  Meta and special keys listed with ":map"
        |hl-EndOfBuffer|   ~  lines after the last line in the buffer
        |hl-NonText|     @  '@' at the end of the window and
                            characters from 'showbreak'
        |hl-Directory|   d  directories in CTRL-D listing and other special
                            things in listings
        |hl-ErrorMsg|    e  error messages
                          h  (obsolete, ignored)
        |hl-IncSearch|   i  'incsearch' highlighting
        |hl-Search|      l  last search pattern highlighting (see 'hlsearch')
        |hl-MoreMsg|     m  |more-prompt|
        |hl-ModeMsg|     M  Mode (e.g., "-- INSERT --")
        |hl-LineNr|      n  line number for ":number" and ":#" commands, and
                            when 'number' or 'relativenumber' option is set.
        |hl-CursorLineNr|  N like n for when 'cursorline' or 'relativenumber' is
                            set.
        |hl-Question|    r  |hit-enter| prompt and yes/no questions
        |hl-StatusLine|  s  status line of current window |status-line|
        |hl-StatusLineNC| S  status lines of not-current windows
        |hl-Title|       t  Titles for output from ":set all", ":autocmd" etc.
        |hl-VertSplit|   c  column used to separate vertically split windows
        |hl-Visual|      v  Visual mode
        |hl-VisualNOS|   V  Visual mode when Vim does is "Not Owning the
                            Selection" Only X11 Gui's |gui-x11| and
                            |xterm-clipboard|.
        |hl-WarningMsg|  w  warning messages
        |hl-WildMenu|    W  wildcard matches displayed for 'wildmenu'
```

```
        |hl-Folded|      f  line used for closed folds
        |hl-FoldColumn|  F  'foldcolumn'
        |hl-DiffAdd|     A  added line in diff mode
        |hl-DiffChange|  C  changed line in diff mode
        |hl-DiffDelete|  D  deleted line in diff mode
        |hl-DiffText|    T  inserted text in diff mode
        |hl-SignColumn|  >  column used for |signs|
        |hl-SpellBad|    B  misspelled word |spell|
        |hl-SpellCap|    P  word that should start with capital |spell|
        |hl-SpellRare|   R  rare word |spell|
        |hl-SpellLocal|  L  word from other region |spell|
        |hl-Conceal|     -  the placeholders used for concealed characters
                            (see 'conceallevel')
        |hl-Pmenu|       +  popup menu normal line
        |hl-PmenuSel|    =  popup menu normal line
        |hl-PmenuSbar|   x  popup menu scrollbar
        |hl-PmenuThumb|  X  popup menu scrollbar thumb

        The display modes are:
                r       reverse         (termcap entry "mr" and "me")
                i       italic          (termcap entry "ZH" and "ZR")
                b       bold            (termcap entry "md" and "me")
                s       standout        (termcap entry "so" and "se")
                u       underline       (termcap entry "us" and "ue")
                c       undercurl       (termcap entry "Cs" and "Ce")
                t       strikethrough   (termcap entry "Ts" and "Te")
                n       no highlighting
                -       no highlighting
                :       use a highlight group
        The default is used for occasions that are not included.
        If you want to change what the display modes do, see |dos-colors|
        for an example.
        When using the ':' display mode, this must be followed by the name of
        a highlight group.  A highlight group can be used to define any type
        of highlighting, including using color.  See |:highlight| on how to
        define one.  The default uses a different group for each occasion.
        See |highlight-default| for the default highlight groups.

                                                *'history'* *'hi'*
'history' 'hi'          number  (Vim default: 50, Vi default: 0,
                                                 set to 200 in |defaults.vim|)
                        global
                        {not in Vi}
        A history of ":" commands, and a history of previous search patterns
        is remembered.  This option decides how many entries may be stored in
        each of these histories (see |cmdline-editing|).
        The maximum value is 10000.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                            *'hkmap'* *'hk'* *'nohkmap'* *'nohk'*
'hkmap' 'hk'            boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        When on, the keyboard is mapped for the Hebrew character set.
        Normally you would set 'allowrevins' and use CTRL-_ in insert mode to
        toggle this option.  See |rileft.txt|.
        NOTE: This option is reset when 'compatible' is set.

                                *'hkmapp'* *'hkp'* *'nohkmapp'* *'nohkp'*
```

```
'hkmapp' 'hkp'          boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        When on, phonetic keyboard mapping is used.  'hkmap' must also be on.
        This is useful if you have a non-Hebrew keyboard.
        See |rileft.txt|.
        NOTE: This option is reset when 'compatible' is set.

                                *'hlsearch'* *'hls'* *'nohlsearch'* *'nohls'*
'hlsearch' 'hls'        boolean (default off)
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+extra_search| feature}
        When there is a previous search pattern, highlight all its matches.
        The type of highlighting used can be set with the 'l' occasion in the
        'highlight' option.  This uses the "Search" highlight group by
        default.  Note that only the matching text is highlighted, any offsets
        are not applied.
        See also: 'incsearch' and |:match|.
        When you get bored looking at the highlighted matches, you can turn it
        off with |:nohlsearch|.  This does not change the option value, as
        soon as you use a search command, the highlighting comes back.
        'redrawtime' specifies the maximum time spent on finding matches.
        When the search pattern can match an end-of-line, Vim will try to
        highlight all of the matched text.  However, this depends on where the
        search starts.  This will be the first line in the window or the first
        line below a closed fold.  A match in a previous line which is not
        drawn may not continue in a newly drawn line.
        You can specify whether the highlight status is restored on startup
        with the 'h' flag in 'viminfo' |viminfo-h|.
        NOTE: This option is reset when 'compatible' is set.

                                                *'icon'* *'noicon'*
'icon'                  boolean (default off, on when title can be restored)
                        global
                        {not in Vi}
                        {not available when compiled without the |+title|
                        feature}
        When on, the icon text of the window will be set to the value of
        'iconstring' (if it is not empty), or to the name of the file
        currently being edited.  Only the last part of the name is used.
        Overridden by the 'iconstring' option.
        Only works if the terminal supports setting window icons (currently
        only X11 GUI and terminals with a non-empty 't_IS' option - these are
        Unix xterm and iris-ansi by default, where 't_IS' is taken from the
        builtin termcap).
        When Vim was compiled with HAVE_X11 defined, the original icon will be
        restored if possible |X11|.  See |X11-icon| for changing the icon on
        X11.
        For MS-Windows the icon can be changed, see |windows-icon|.

                                                *'iconstring'*
'iconstring'            string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+title|
                        feature}
        When this option is not empty, it will be used for the icon text of
        the window.  This happens only when the 'icon' option is on.
```

```
        Only works if the terminal supports setting window icon text
        (currently only X11 GUI and terminals with a non-empty 't_IS' option).
        Does not work for MS Windows.
        When Vim was compiled with HAVE_X11 defined, the original icon will be
        restored if possible |X11|.
        When this option contains printf-style '%' items, they will be
        expanded according to the rules used for 'statusline'.  See
        'titlestring' for example settings.
        {not available when compiled without the |+statusline| feature}

                        *'ignorecase'* *'ic'* *'noignorecase'* *'noic'*
'ignorecase' 'ic'       boolean (default off)
                        global
        Ignore case in search patterns.  Also used when searching in the tags
        file.
        Also see 'smartcase' and 'tagcase'.
        Can be overruled by using "\c" or "\C" in the pattern, see
        |/ignorecase|.

                                                *'imactivatefunc'* *'imaf'*
'imactivatefunc' 'imaf' string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with |+xim| and
                        |+GUI_GTK|}
        This option specifies a function that will be called to
        activate/inactivate Input Method.

        Example: >
                function ImActivateFunc(active)
                  if a:active
                    ... do something
                  else
                    ... do something
                  endif
                  " return value is not used
                endfunction
                set imactivatefunc=ImActivateFunc
<
                                                *'imactivatekey'* *'imak'*
'imactivatekey' 'imak'  string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with |+xim| and
                        |+GUI_GTK|}                             *E599*
        Specifies the key that your Input Method in X-Windows uses for
        activation.  When this is specified correctly, vim can fully control
        IM with 'imcmdline', 'iminsert' and 'imsearch'.
        You can't use this option to change the activation key, the option
        tells Vim what the key is.
        Format:
                [MODIFIER_FLAG-]KEY_STRING

        These characters can be used for MODIFIER_FLAG (case is ignored):
                S           Shift key
                L           Lock key
                C           Control key
                1           Mod1 key
                2           Mod2 key
                3           Mod3 key
                4           Mod4 key
                5           Mod5 key
```

```
        Combinations are allowed, for example "S-C-space" or "SC-space" are
        both shift+ctrl+space.
        See <X11/keysymdef.h> and XStringToKeysym for KEY_STRING.

        Example: >
                :set imactivatekey=S-space
<       "S-space" means shift+space.  This is the activation key for kinput2 +
        canna (Japanese), and ami (Korean).

                                *'imcmdline'* *'imc'* *'noimcmdline'* *'noimc'*
'imcmdline' 'imc'       boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+xim|,
                        |+multi_byte_ime| or |global-ime| features}
        When set the Input Method is always on when starting to edit a command
        line, unless entering a search pattern (see 'imsearch' for that).
        Setting this option is useful when your input method allows entering
        English characters directly, e.g., when it's used to type accented
        characters with dead keys.

                                *'imdisable'* *'imd'* *'noimdisable'* *'noimd'*
'imdisable' 'imd'       boolean (default off, on for some systems (SGI))
                        global
                        {not in Vi}
                        {only available when compiled with the |+xim|,
                        |+multi_byte_ime| or |global-ime| features}
        When set the Input Method is never used.  This is useful to disable
        the IM when it doesn't work properly.
        Currently this option is on by default for SGI/IRIX machines.  This
        may change in later releases.

                                                *'iminsert'* *'imi'*
'iminsert' 'imi'        number (default 0)
                        local to buffer
                        {not in Vi}
        Specifies whether :lmap or an Input Method (IM) is to be used in
        Insert mode.  Valid values:
                0       :lmap is off and IM is off
                1       :lmap is ON and IM is off
                2       :lmap is off and IM is ON
        2 is available only when compiled with the |+multi_byte_ime|, |+xim|
        or |global-ime|.
        To always reset the option to zero when leaving Insert mode with <Esc>
        this can be used: >
                :inoremap <ESC> <ESC>:set iminsert=0<CR>
<       This makes :lmap and IM turn off automatically when leaving Insert
        mode.
        Note that this option changes when using CTRL-^ in Insert mode
        |i_CTRL-^|.
        The value is set to 1 when setting 'keymap' to a valid keymap name.
        It is also used for the argument of commands like "r" and "f".
        The value 0 may not work correctly with Athena and Motif with some XIM
        methods.  Use 'imdisable' to disable XIM then.

                                                *'imsearch'* *'ims'*
'imsearch' 'ims'        number (default -1)
                        local to buffer
                        {not in Vi}
        Specifies whether :lmap or an Input Method (IM) is to be used when
        entering a search pattern.  Valid values:
                -1      the value of 'iminsert' is used, makes it look like
```

```
                        'iminsert' is also used when typing a search pattern
                0       :lmap is off and IM is off
                1       :lmap is ON and IM is off
                2       :lmap is off and IM is ON
        Note that this option changes when using CTRL-^ in Command-line mode
        |c_CTRL-^|.
        The value is set to 1 when it is not -1 and setting the 'keymap'
        option to a valid keymap name.
        The value 0 may not work correctly with Athena and Motif with some XIM
        methods.  Use 'imdisable' to disable XIM then.

                                                *'imstatusfunc'* *'imsf'*
'imstatusfunc' 'imsf'   string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with |+xim| and
                        |+GUI_GTK|}
        This option specifies a function that is called to obtain the status
        of Input Method.  It must return a positive number when IME is active.

        Example: >
                function ImStatusFunc()
                  let is_active = ...do something
                  return is_active ? 1 : 0
                endfunction
                set imstatusfunc=ImStatusFunc
<
        NOTE: This function is invoked very often.  Keep it fast.

                                                *'imstyle'* *'imst'*
'imstyle' 'imst'        number (default 1)
                        global
                        {not in Vi}
                        {only available when compiled with |+xim| and
                        |+GUI_GTK|}
        This option specifies the input style of Input Method:
        0   use on-the-spot style
        1   over-the-spot style
        See: |xim-input-style|

        For a long time on-the-spot style had been used in the GTK version of
        vim, however, it is known that it causes troubles when using mappings,
        |single-repeat|, etc.  Therefore over-the-spot style becomes the
        default now.  This should work fine for most people, however if you
        have any problem with it, try using on-the-spot style.

                                                *'include'* *'inc'*
'include' 'inc'         string  (default "^\s*#\s*include")
                        global or local to buffer |global-local|
                        {not in Vi}
                        {not available when compiled without the
                        |+find_in_path| feature}
        Pattern to be used to find an include command.  It is a search
        pattern, just like for the "/" command (See |pattern|).  The default
        value is for C programs.  This option is used for the commands "[i",
        "]I", "[d", etc.
        Normally the 'isfname' option is used to recognize the file name that
        comes after the matched pattern.  But if "\zs" appears in the pattern
        then the text matched from "\zs" to the end, or until "\ze" if it
        appears, is used as the file name.  Use this to include characters
        that are not in 'isfname', such as a space.  You can then use
        'includeexpr' to process the matched text.
```

```
        See |option-backslash| about including spaces and backslashes.

                                                *'includeexpr'* *'inex'*
'includeexpr' 'inex'    string  (default "")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the
                        |+find_in_path| or |+eval| features}
        Expression to be used to transform the string found with the 'include'
        option to a file name.  Mostly useful to change "." to "/" for Java: >
                :set includeexpr=substitute(v:fname,'\\.','/','g')
<       The "v:fname" variable will be set to the file name that was detected.

        Also used for the |gf| command if an unmodified file name can't be
        found.  Allows doing "gf" on the name after an 'include' statement.
        Also used for |<cfile>|.

        The expression will be evaluated in the |sandbox| when set from a
        modeline, see |sandbox-option|.

        It is not allowed to change text or jump to another window while
        evaluating 'includeexpr' |textlock|.

                                *'incsearch'* *'is'* *'noincsearch'* *'nois'*
'incsearch' 'is'        boolean (default off, set in |defaults.vim| if the
                                                +reltime feature is supported)
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+extra_search| features}
        While typing a search command, show where the pattern, as it was typed
        so far, matches.  The matched string is highlighted.  If the pattern
        is invalid or not found, nothing is shown.  The screen will be updated
        often, this is only useful on fast terminals.
        Note that the match will be shown, but the cursor will return to its
        original position when no match is found and when pressing <Esc>.  You
        still need to finish the search command with <Enter> to move the
        cursor to the match.
        You can use the CTRL-G and CTRL-T keys to move to the next and
        previous match. |c_CTRL-G| |c_CTRL-T|
        When compiled with the |+reltime| feature Vim only searches for about
        half a second.  With a complicated pattern and/or a lot of text the
        match may not be found.  This is to avoid that Vim hangs while you
        are typing the pattern.
        The highlighting can be set with the 'i' flag in 'highlight'.
        See also: 'hlsearch'.
        CTRL-L can be used to add one character from after the current match
        to the command line.  If 'ignorecase' and 'smartcase' are set and the
        command line has no uppercase characters, the added character is
        converted to lowercase.
        CTRL-R CTRL-W can be used to add the word at the end of the current
        match, excluding the characters that were already typed.
        NOTE: This option is reset when 'compatible' is set.

                                                *'indentexpr'* *'inde'*
'indentexpr' 'inde'     string  (default "")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+cindent|
                        or |+eval| features}
        Expression which is evaluated to obtain the proper indent for a line.
        It is used when a new line is created, for the |=| operator and
```

```
        in Insert mode as specified with the 'indentkeys' option.
        When this option is not empty, it overrules the 'cindent' and
        'smartindent' indenting.  When 'lisp' is set, this option is
        overridden by the Lisp indentation algorithm.
        When 'paste' is set this option is not used for indenting.
        The expression is evaluated with |v:lnum| set to the line number for
        which the indent is to be computed.  The cursor is also in this line
        when the expression is evaluated (but it may be moved around).
        The expression must return the number of spaces worth of indent.  It
        can return "-1" to keep the current indent (this means 'autoindent' is
        used for the indent).
        Functions useful for computing the indent are |indent()|, |cindent()|
        and |lispindent()|.
        The evaluation of the expression must not have side effects!  It must
        not change the text, jump to another window, etc.  Afterwards the
        cursor position is always restored, thus the cursor may be moved.
        Normally this option would be set to call a function: >
                :set indentexpr=GetMyIndent()
<       Error messages will be suppressed, unless the 'debug' option contains
        "msg".
        See |indent-expression|.
        NOTE: This option is set to "" when 'compatible' is set.

        The expression will be evaluated in the |sandbox| when set from a
        modeline, see |sandbox-option|.

        It is not allowed to change text or jump to another window while
        evaluating 'indentexpr' |textlock|.


                                                *'indentkeys'* *'indk'*
'indentkeys' 'indk'     string  (default "0{,0},:,0#,!^F,o,O,e")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+cindent|
                        feature}
        A list of keys that, when typed in Insert mode, cause reindenting of
        the current line.  Only happens if 'indentexpr' isn't empty.
        The format is identical to 'cinkeys', see |indentkeys-format|.
        See |C-indenting| and |indent-expression|.

                        *'infercase'* *'inf'* *'noinfercase'* *'noinf'*
'infercase' 'inf'       boolean (default off)
                        local to buffer
                        {not in Vi}
        When doing keyword completion in insert mode |ins-completion|, and
        'ignorecase' is also on, the case of the match is adjusted depending
        on the typed text.  If the typed text contains a lowercase letter
        where the match has an upper case letter, the completed part is made
        lowercase.  If the typed text has no lowercase letters and the match
        has a lowercase letter where the typed text has an uppercase letter,
        and there is a letter before it, the completed part is made uppercase.
        With 'noinfercase' the match is used as-is.

                        *'insertmode'* *'im'* *'noinsertmode'* *'noim'*
'insertmode' 'im'       boolean (default off)
                        global
                        {not in Vi}
        Makes Vim work in a way that Insert mode is the default mode.  Useful
        if you want to use Vim as a modeless editor.  Used for |evim|.
        These Insert mode commands will be useful:
        - Use the cursor keys to move around.
```

```
        - Use CTRL-O to execute one Normal mode command |i_CTRL-O|.  When
          this is a mapping, it is executed as if 'insertmode' was off.
          Normal mode remains active until the mapping is finished.
        - Use CTRL-L to execute a number of Normal mode commands, then use
          <Esc> to get back to Insert mode.  Note that CTRL-L moves the cursor
          left, like <Esc> does when 'insertmode' isn't set.  |i_CTRL-L|

        These items change when 'insertmode' is set:
        - when starting to edit of a file, Vim goes to Insert mode.
        - <Esc> in Insert mode is a no-op and beeps.
        - <Esc> in Normal mode makes Vim go to Insert mode.
        - CTRL-L in Insert mode is a command, it is not inserted.
        - CTRL-Z in Insert mode suspends Vim, see |CTRL-Z|.     *i_CTRL-Z*
        However, when <Esc> is used inside a mapping, it behaves like
        'insertmode' was not set.  This was done to be able to use the same
        mappings with 'insertmode' set or not set.
        When executing commands with |:normal| 'insertmode' is not used.

        NOTE: This option is reset when 'compatible' is set.

                                                *'isfname'* *'isf'*
'isfname' 'isf'         string  (default for MS-DOS, Win32 and OS/2:
                                 "@,48-57,/,\,.,-,_,+,,,#,$,%,{,},[,],:,@-@,!,~,="
                                for AMIGA: "@,48-57,/,.,-,_,+,,,$,:"
                                for VMS: "@,48-57,/,.,-,_,+,,,#,$,%,<,>,[,],:,;,~"
                                for OS/390: "@,240-249,/,.,-,_,+,,,#,$,%,~,="
                                otherwise: "@,48-57,/,.,-,_,+,,,#,$,%,~,=")
                        global
                        {not in Vi}
        The characters specified by this option are included in file names and
        path names.  Filenames are used for commands like "gf", "[i" and in
        the tags file.  It is also used for "\f" in a |pattern|.
        Multi-byte characters 256 and above are always included, only the
        characters up to 255 are specified with this option.
        For UTF-8 the characters 0xa0 to 0xff are included as well.
        Think twice before adding white space to this option.  Although a
        space may appear inside a file name, the effect will be that Vim
        doesn't know where a file name starts or ends when doing completion.
        It most likely works better without a space in 'isfname'.

        Note that on systems using a backslash as path separator, Vim tries to
        do its best to make it work as you would expect.  That is a bit
        tricky, since Vi originally used the backslash to escape special
        characters.  Vim will not remove a backslash in front of a normal file
        name character on these systems, but it will on Unix and alikes.  The
        '&' and '^' are not included by default, because these are special for
        cmd.exe.

        The format of this option is a list of parts, separated with commas.
        Each part can be a single character number or a range.  A range is two
        character numbers with '-' in between.  A character number can be a
        decimal number between 0 and 255 or the ASCII character itself (does
        not work for digits).  Example:
                "_,-,128-140,#-43"      (include '_' and '-' and the range
                                        128 to 140 and '#' to 43)
        If a part starts with '^', the following character number or range
        will be excluded from the option.  The option is interpreted from left
        to right.  Put the excluded character after the range where it is
        included.  To include '^' itself use it as the last character of the
        option or the end of a range.  Example:
                "^a-z,#,^"      (exclude 'a' to 'z', include '#' and '^')
        If the character is '@', all characters where isalpha() returns TRUE
```

```
        are included.  Normally these are the characters a to z and A to Z,
        plus accented characters.  To include '@' itself use "@-@".  Examples:
                "@,^a-z"        All alphabetic characters, excluding lower
                                case ASCII letters.
                "a-z,A-Z,@-@"   All letters plus the '@' character.
        A comma can be included by using it where a character number is
        expected.  Example:
                "48-57,,,_"     Digits, comma and underscore.
        A comma can be excluded by prepending a '^'.  Example:
                " -~,^,,9"      All characters from space to '~', excluding
                                comma, plus <Tab>.
        See |option-backslash| about including spaces and backslashes.

                                                *'isident'* *'isi'*
'isident' 'isi'         string  (default for MS-DOS, Win32 and OS/2:
                                        "@,48-57,_,128-167,224-235"
                                otherwise: "@,48-57,_,192-255")
                        global
                        {not in Vi}
        The characters given by this option are included in identifiers.
        Identifiers are used in recognizing environment variables and after a
        match of the 'define' option.  It is also used for "\i" in a
        |pattern|.  See 'isfname' for a description of the format of this
        option.
        Careful: If you change this option, it might break expanding
        environment variables.  E.g., when '/' is included and Vim tries to
        expand "$HOME/.viminfo".  Maybe you should change 'iskeyword' instead.

                                                *'iskeyword'* *'isk'*
'iskeyword' 'isk'       string (Vim default for MS-DOS and Win32:
                                        "@,48-57,_,128-167,224-235"
                                   otherwise:  "@,48-57,_,192-255"
                                Vi default: "@,48-57,_")
                        local to buffer
                        {not in Vi}
        Keywords are used in searching and recognizing with many commands:
        "w", "*", "[i", etc.  It is also used for "\k" in a |pattern|.  See
        'isfname' for a description of the format of this option.  For C
        programs you could use "a-z,A-Z,48-57,_,.,-,>".
        For a help file it is set to all non-blank printable characters except
        '*', '"' and '|' (so that CTRL-] on a command finds the help for that
        command).
        When the 'lisp' option is on the '-' character is always included.
        This option also influences syntax highlighting, unless the syntax
        uses |:syn-iskeyword|.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'isprint'* *'isp'*
'isprint' 'isp' string  (default for MS-DOS, Win32, OS/2 and Macintosh:
                                "@,~-255"; otherwise: "@,161-255")
                        global
                        {not in Vi}
        The characters given by this option are displayed directly on the
        screen.  It is also used for "\p" in a |pattern|.  The characters from
        space (ASCII 32) to '~' (ASCII 126) are always displayed directly,
        even when they are not included in 'isprint' or excluded.  See
        'isfname' for a description of the format of this option.

        Non-printable characters are displayed with two characters:
                  0 -  31       "^@" - "^_"
                 32 - 126       always single characters
```

```
                  127          "^?"
                128 - 159      "~@" - "~_"
                160 - 254      "| " - "|~"
                   255         "~?"
        When 'encoding' is a Unicode one, illegal bytes from 128 to 255 are
        displayed as <xx>, with the hexadecimal value of the byte.
        When 'display' contains "uhex" all unprintable characters are
        displayed as <xx>.
        The SpecialKey highlighting will be used for unprintable characters.
        |hl-SpecialKey|

        Multi-byte characters 256 and above are always included, only the
        characters up to 255 are specified with this option.  When a character
        is printable but it is not available in the current font, a
        replacement character will be shown.
        Unprintable and zero-width Unicode characters are displayed as <xxxx>.
        There is no option to specify these characters.

                        *'joinspaces'* *'js'* *'nojoinspaces'* *'nojs'*
'joinspaces' 'js'       boolean (default on)
                        global
                        {not in Vi}
        Insert two spaces after a '.', '?' and '!' with a join command.
        When 'cpoptions' includes the 'j' flag, only do this after a '.'.
        Otherwise only one space is inserted.
        NOTE: This option is set when 'compatible' is set.

                                                            *'key'*
'key'                   string  (default "")
                        local to buffer
                        {not in Vi}
                        {only available when compiled with the |+cryptv|
                        feature}
        The key that is used for encrypting and decrypting the current buffer.
        See |encryption| and 'cryptmethod'.
        Careful: Do not set the key value by hand, someone might see the typed
        key.  Use the |:X| command.  But you can make 'key' empty: >
                :set key=
<       It is not possible to get the value of this option with ":set key" or
        "echo &key".  This is to avoid showing it to someone who shouldn't
        know.  It also means you cannot see it yourself once you have set it,
        be careful not to make a typing error!
        You can use "&key" in an expression to detect whether encryption is
        enabled.  When 'key' is set it returns "*****" (five stars).

                                        *'keymap'* *'kmp'* *E544*
'keymap' 'kmp'          string  (default "")
                        local to buffer
                        {not in Vi}
                        {only available when compiled with the |+keymap|
                        feature}
        Name of a keyboard mapping.  See |mbyte-keymap|.
        Setting this option to a valid keymap name has the side effect of
        setting 'iminsert' to one, so that the keymap becomes effective.
        'imsearch' is also set to one, unless it was -1
        Only normal file name characters can be used, "/\*?[|<>" are illegal.

                                        *'keymodel'* *'km'*
'keymodel' 'km'         string  (default "")
                        global
                        {not in Vi}
        List of comma separated words, which enable special things that keys
```

```
        can do.  These values can be used:
           startsel     Using a shifted special key starts selection (either
                        Select mode or Visual mode, depending on "key" being
                        present in 'selectmode').
           stopsel      Using a not-shifted special key stops selection.
        Special keys in this context are the cursor keys, <End>, <Home>,
        <PageUp> and <PageDown>.
        The 'keymodel' option is set by the |:behave| command.

                                        *'keywordprg'* *'kp'*
'keywordprg' 'kp'       string  (default "man" or "man -s",  DOS: ":help",
                                                                VMS: "help")
                        global or local to buffer |global-local|
                        {not in Vi}
        Program to use for the |K| command.  Environment variables are
        expanded |:set_env|.  ":help" may be used to access the Vim internal
        help.  (Note that previously setting the global option to the empty
        value did this, which is now deprecated.)
        When the first character is ":", the command is invoked as a Vim
        Ex command prefixed with [count].
        When "man", "man -s" or an Ex command is used, Vim will automatically
        translate a count for the "K" command and pass it as the first
        argument.  For "man -s" the "-s" is removed when there is no count.
        See |option-backslash| about including spaces and backslashes.
        Example: >
                :set keywordprg=man\ -s
<       This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                        *'langmap'* *'lmap'* *E357* *E358*
'langmap' 'lmap'        string  (default "")
                        global
                        {not in Vi}
                        {only available when compiled with the |+langmap|
                        feature}
        This option allows switching your keyboard into a special language
        mode.  When you are typing text in Insert mode the characters are
        inserted directly.  When in Normal mode the 'langmap' option takes
        care of translating these special characters to the original meaning
        of the key.  This means you don't have to change the keyboard mode to
        be able to execute Normal mode commands.
        This is the opposite of the 'keymap' option, where characters are
        mapped in Insert mode.
        Also consider resetting 'langremap' to avoid 'langmap' applies to
        characters resulting from a mapping.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

        Example (for Greek, in UTF-8):                          *greek*  >
            :set
langmap=AA,BB,ΨC,ΔD,EE,ΦF,ΓG,HH,II,ΞJ,KK,ΛL,MM,NN,OO,ΠP,QQ,PR,ΣS,TT,ΘU,ΩV,WW,XX,YY,ZZ,αa,βb,ψc,δd,εe
<       Example (exchanges meaning of z and y for commands): >
            :set langmap=zy,yz,ZY,YZ
<
        The 'langmap' option is a list of parts, separated with commas.  Each
        part can be in one of two forms:
        1.  A list of pairs.  Each pair is a "from" character immediately
            followed by the "to" character.  Examples: "aA", "aAbBcC".
        2.  A list of "from" characters, a semi-colon and a list of "to"
            characters.  Example: "abc;ABC"
        Example: "aA,fgh;FGH,cCdDeE"
        Special characters need to be preceded with a backslash.  These are
```

```
        ";", ',' and backslash itself.

        This will allow you to activate vim actions without having to switch
        back and forth between the languages.  Your language characters will
        be understood as normal vim English characters (according to the
        langmap mappings) in the following cases:
         o Normal/Visual mode (commands, buffer/register names, user mappings)
         o Insert/Replace Mode: Register names after CTRL-R
         o Insert/Replace Mode: Mappings
        Characters entered in Command-line mode will NOT be affected by
        this option.   Note that this option can be changed at any time
        allowing to switch between mappings for different languages/encodings.
        Use a mapping to avoid having to type it each time!

                                              *'langmenu'* *'lm'*
'langmenu' 'lm'         string  (default "")
                        global
                        {not in Vi}
                        {only available when compiled with the |+menu| and
                        |+multi_lang| features}
        Language to use for menu translation.  Tells which file is loaded
        from the "lang" directory in 'runtimepath': >
                "lang/menu_" . &langmenu . ".vim"
<       (without the spaces).  For example, to always use the Dutch menus, no
        matter what $LANG is set to: >
                :set langmenu=nl_NL.ISO_8859-1
<       When 'langmenu' is empty, |v:lang| is used.
        Only normal file name characters can be used, "/\*?[|<>" are illegal.
        If your $LANG is set to a non-English language but you do want to use
        the English menus: >
                :set langmenu=none
<       This option must be set before loading menus, switching on filetype
        detection or syntax highlighting.  Once the menus are defined setting
        this option has no effect.  But you could do this: >
                :source $VIMRUNTIME/delmenu.vim
                :set langmenu=de_DE.ISO_8859-1
                :source $VIMRUNTIME/menu.vim
<       Warning: This deletes all menus that you defined yourself!

                        *'langnoremap'* *'lnr'* *'nolangnoremap'* *'nolnr'*
'langnoremap' 'lnr'     boolean (default off, set in |defaults.vim|)
                        global
                        {not in Vi}
                        {only available when compiled with the |+langmap|
                        feature}
        This is just like 'langremap' but with the value inverted.  It only
        exists for backwards compatibility.  When setting 'langremap' then
        'langnoremap' is set to the inverted value, and the other way around.

                        *'langremap'* *'lrm'* *'nolangremap'* *'nolrm'*
'langremap' 'lrm'       boolean (default on, reset in |defaults.vim|)
                        global
                        {not in Vi}
                        {only available when compiled with the |+langmap|
                        feature}
        When off, setting 'langmap' does not apply to characters resulting from
        a mapping.  This basically means, if you noticed that setting
        'langmap' disables some of your mappings, try resetting this option.
        This option defaults to on for backwards compatibility.  Set it off if
        that works for you to avoid mappings to break.

                                              *'laststatus'* *'ls'*
```

```
'laststatus' 'ls'       number  (default 1)
                        global
                        {not in Vi}
        The value of this option influences when the last window will have a
        status line:
                0: never
                1: only if there are at least two windows
                2: always
        The screen looks nicer with a status line if you have several
        windows, but it takes another screen line. |status-line|

                        *'lazyredraw'* *'lz'* *'nolazyredraw'* *'nolz'*
'lazyredraw' 'lz'       boolean (default off)
                        global
                        {not in Vi}
        When this option is set, the screen will not be redrawn while
        executing macros, registers and other commands that have not been
        typed.  Also, updating the window title is postponed.  To force an
        update use |:redraw|.

                        *'linebreak'* *'lbr'* *'nolinebreak'* *'nolbr'*
'linebreak' 'lbr'       boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+linebreak|
                        feature}
        If on, Vim will wrap long lines at a character in 'breakat' rather
        than at the last character that fits on the screen.  Unlike
        'wrapmargin' and 'textwidth', this does not insert <EOL>s in the file,
        it only affects the way the file is displayed, not its contents.
        If 'breakindent' is set, line is visually indented. Then, the value
        of 'showbreak' is used to put in front of wrapped lines. This option
        is not used when the 'wrap' option is off.
        Note that <Tab> characters after an <EOL> are mostly not displayed
        with the right amount of white space.

                                                *'lines'* *E593*
'lines'                 number  (default 24 or terminal height)
                        global
        Number of lines of the Vim window.
        Normally you don't need to set this.  It is done automatically by the
        terminal initialization code.  Also see |posix-screen-size|.
        When Vim is running in the GUI or in a resizable window, setting this
        option will cause the window size to be changed.  When you only want
        to use the size for the GUI, put the command in your |gvimrc| file.
        Vim limits the number of lines to what fits on the screen.  You can
        use this command to get the tallest window possible: >
                :set lines=999
<       Minimum value is 2, maximum value is 1000.
        If you get fewer lines than expected, check the 'guiheadroom' option.
        When you set this option and Vim is unable to change the physical
        number of lines of the display, the display may be messed up.

                                                *'linespace'* *'lsp'*
'linespace' 'lsp'       number  (default 0, 1 for Win32 GUI)
                        global
                        {not in Vi}
                        {only in the GUI}
        Number of pixel lines inserted between characters.  Useful if the font
        uses the full character cell height, making lines touch each other.
        When non-zero there is room for underlining.
        With some fonts there can be too much room between lines (to have
```

```
        space for ascents and descents).  Then it makes sense to set
        'linespace' to a negative value.  This may cause display problems
        though!

                                                *'lisp'* *'nolisp'*
'lisp'                  boolean (default off)
                        local to buffer
                        {not available when compiled without the |+lispindent|
                        feature}
        Lisp mode: When <Enter> is typed in insert mode set the indent for
        the next line to Lisp standards (well, sort of).  Also happens with
        "cc" or "S".  'autoindent' must also be on for this to work.  The 'p'
        flag in 'cpoptions' changes the method of indenting: Vi compatible or
        better.  Also see 'lispwords'.
        The '-' character is included in keyword characters.  Redefines the
        "=" operator to use this same indentation algorithm rather than
        calling an external program if 'equalprg' is empty.
        This option is not used when 'paste' is set.
        {Vi: Does it a little bit differently}

                                                *'lispwords'* *'lw'*
'lispwords' 'lw'        string  (default is very long)
                        global or local to buffer |global-local|
                        {not in Vi}
                        {not available when compiled without the |+lispindent|
                        feature}
        Comma separated list of words that influence the Lisp indenting.
        |'lisp'|

                                                *'list'* *'nolist'*
'list'                  boolean (default off)
                        local to window
        List mode: Show tabs as CTRL-I is displayed, display $ after end of
        line.  Useful to see the difference between tabs and spaces and for
        trailing blanks.  Further changed by the 'listchars' option.

        The cursor is displayed at the start of the space a Tab character
        occupies, not at the end as usual in Normal mode.  To get this cursor
        position while displaying Tabs with spaces, use: >
                :set list lcs=tab:\ \
<
        Note that list mode will also affect formatting (set with 'textwidth'
        or 'wrapmargin') when 'cpoptions' includes 'L'.  See 'listchars' for
        changing the way tabs are displayed.

                                                *'listchars'* *'lcs'*
'listchars' 'lcs'       string  (default "eol:$")
                        global
                        {not in Vi}
        Strings to use in 'list' mode and for the |:list| command.  It is a
        comma separated list of string settings.
                                                        *lcs-eol*
          eol:c         Character to show at the end of each line.  When
                        omitted, there is no extra character at the end of the
                        line.
                                                        *lcs-tab*
          tab:xy        Two characters to be used to show a tab.  The first
                        char is used once.  The second char is repeated to
                        fill the space that the tab normally occupies.
                        "tab:>-" will show a tab that takes four spaces as
                        ">---".  When omitted, a tab is show as ^I.
                                                        *lcs-space*
```

```
          space:c       Character to show for a space.  When omitted, spaces
                        are left blank.
                                                        *lcs-trail*
          trail:c       Character to show for trailing spaces.  When omitted,
                        trailing spaces are blank.  Overrides the "space"
                        setting for trailing spaces.
                                                        *lcs-extends*
          extends:c     Character to show in the last column, when 'wrap' is
                        off and the line continues beyond the right of the
                        screen.
                                                        *lcs-precedes*
          precedes:c    Character to show in the first column, when 'wrap'
                        is off and there is text preceding the character
                        visible in the first column.
                                                        *lcs-conceal*
          conceal:c     Character to show in place of concealed text, when
                        'conceallevel' is set to 1.
                                                        *lcs-nbsp*
          nbsp:c        Character to show for a non-breakable space character
                        (0xA0 (160 decimal) and U+202F).  Left blank when
                        omitted.

        The characters ':' and ',' should not be used.  UTF-8 characters can
        be used when 'encoding' is "utf-8", otherwise only printable
        characters are allowed.  All characters must be single width.

        Examples: >
            :set lcs=tab:>-,trail:-
            :set lcs=tab:>-,eol:<,nbsp:%
            :set lcs=extends:>,precedes:<
<       The "NonText" highlighting will be used for "eol", "extends" and
        "precedes".  "SpecialKey" for "nbsp", "space", "tab" and "trail".
        |hl-NonText| |hl-SpecialKey|

                        *'lpl'* *'nolpl'* *'loadplugins'* *'noloadplugins'*
'loadplugins' 'lpl'     boolean (default on)
                        global
                        {not in Vi}
        When on the plugin scripts are loaded when starting up |load-plugins|.
        This option can be reset in your |vimrc| file to disable the loading
        of plugins.
        Note that using the "-u NONE", "-u DEFAULTS" and "--noplugin" command
        line arguments reset this option.  See |-u| and |--noplugin|.

                                                *'luadll'*
'luadll'                string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+lua/dyn|
                        feature}
        Specifies the name of the Lua shared library. The default is
        DYNAMIC_LUA_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'macatsui'* *'nomacatsui'*
'macatsui'              boolean (default on)
                        global
                        {only available in Mac GUI version}
        This is a workaround for when drawing doesn't work properly.  When set
        and compiled with multi-byte support ATSUI text drawing is used.  When
```

```
        not set ATSUI text drawing is not used.  Switch this option off when
        you experience drawing problems.  In a future version the problems may
        be solved and this option becomes obsolete.  Therefore use this method
        to unset it: >
                if exists('&macatsui')
                   set nomacatsui
                endif
<       Another option to check if you have drawing problems is
        'termencoding'.

                                                *'magic'* *'nomagic'*
'magic'                 boolean (default on)
                        global
        Changes the special characters that can be used in search patterns.
        See |pattern|.
        WARNING: Switching this option off most likely breaks plugins!  That
        is because many patterns assume it's on and will fail when it's off.
        Only switch it off when working with old Vi scripts.  In any other
        situation write patterns that work when 'magic' is on.  Include "\M"
        when you want to |/\M|.

                                                *'makeef'* *'mef'*
'makeef' 'mef'          string  (default: "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+quickfix|
                        feature}
        Name of the errorfile for the |:make| command (see |:make_makeprg|)
        and the |:grep| command.
        When it is empty, an internally generated temp file will be used.
        When "##" is included, it is replaced by a number to make the name
        unique.  This makes sure that the ":make" command doesn't overwrite an
        existing file.
        NOT used for the ":cf" command.  See 'errorfile' for that.
        Environment variables are expanded |:set_env|.
        See |option-backslash| about including spaces and backslashes.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                        *'makeencoding'* *'menc'*
'makeencoding' 'menc'   string  (default "")
                        global or local to buffer |global-local|
                        {only available when compiled with the |+multi_byte|
                        feature}
                        {not in Vi}
        Encoding used for reading the output of external commands.  When empty,
        encoding is not converted.
        This is used for `:make`, `:lmake`, `:grep`, `:lgrep`, `:grepadd`,
        `:lgrepadd`, `:cfile`, `:cgetfile`, `:caddfile`, `:lfile`, `:lgetfile`,
        and `:laddfile`.

        This would be mostly useful when you use MS-Windows and set 'encoding'
        to "utf-8".  If |+iconv| is enabled and GNU libiconv is used, setting
        'makeencoding' to "char" has the same effect as setting to the system
        locale encoding.  Example: >
                :set encoding=utf-8
                :set makeencoding=char  " system locale is used
<
                                                *'makeprg'* *'mp'*
'makeprg' 'mp'          string  (default "make", VMS: "MMS")
                        global or local to buffer |global-local|
                        {not in Vi}
```

```
        Program to use for the ":make" command.  See |:make_makeprg|.
        This option may contain '%' and '#' characters (see  |:_%| and |:_#|),
        which are expanded to the current and alternate file name.  Use |::S|
        to escape file names in case they contain special characters.
        Environment variables are expanded |:set_env|.  See |option-backslash|
        about including spaces and backslashes.
        Note that a '|' must be escaped twice: once for ":set" and once for
        the interpretation of a command.  When you use a filter called
        "myfilter" do it like this: >
            :set makeprg=gmake\ \\\|\ myfilter
<       The placeholder "$*" can be given (even multiple times) to specify
        where the arguments will be included, for example: >
            :set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
<       This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'matchpairs'* *'mps'*
'matchpairs' 'mps'      string  (default "(:),{:},[:]")
                        local to buffer
                        {not in Vi}
        Characters that form pairs.  The |%| command jumps from one to the
        other.
        Only character pairs are allowed that are different, thus you cannot
        jump between two double quotes.
        The characters must be separated by a colon.
        The pairs must be separated by a comma.  Example for including '<' and
        '>' (HTML): >
                :set mps+=<:>

<       A more exotic example, to jump between the '=' and ';' in an
        assignment, useful for languages like C and Java: >
                :au FileType c,cpp,java set mps+==:;

<       For a more advanced way of using "%", see the matchit.vim plugin in
        the $VIMRUNTIME/macros directory. |add-local-help|

                                                *'matchtime'* *'mat'*
'matchtime' 'mat'       number  (default 5)
                        global
                        {not in Vi}{in Nvi}
        Tenths of a second to show the matching paren, when 'showmatch' is
        set.  Note that this is not in milliseconds, like other options that
        set a time.  This is to be compatible with Nvi.

                                                *'maxcombine'* *'mco'*
'maxcombine' 'mco'      number (default 2)
                        global
                        {not in Vi}
                        {only available when compiled with the |+multi_byte|
                        feature}
        The maximum number of combining characters supported for displaying.
        Only used when 'encoding' is "utf-8".
        The default is OK for most languages.  Hebrew may require 4.
        Maximum value is 6.
        Even when this option is set to 2 you can still edit text with more
        combining characters, you just can't see them.  Use |g8| or |ga|.
        See |mbyte-combining|.

                                                *'maxfuncdepth'* *'mfd'*
'maxfuncdepth' 'mfd'    number  (default 100)
                        global
                        {not in Vi}
```

```
                        {not available when compiled without the |+eval|
                        feature}
        Maximum depth of function calls for user functions.  This normally
        catches endless recursion.  When using a recursive function with
        more depth, set 'maxfuncdepth' to a bigger number.  But this will use
        more memory, there is the danger of failing when memory is exhausted.
        Increasing this limit above 200 also changes the maximum for Ex
        command resursion, see |E169|.
        See also |:function|.

                                                *'maxmapdepth'* *'mmd'* *E223*
'maxmapdepth' 'mmd'     number  (default 1000)
                        global
                        {not in Vi}
        Maximum number of times a mapping is done without resulting in a
        character to be used.  This normally catches endless mappings, like
        ":map x y" with ":map y x".  It still does not catch ":map g wg",
        because the 'w' is used before the next mapping is done.  See also
        |key-mapping|.

                                                *'maxmem'* *'mm'*
'maxmem' 'mm'           number  (default between 256 to 5120 (system
                                 dependent) or half the amount of memory
                                 available)
                        global
                        {not in Vi}
        Maximum amount of memory (in Kbyte) to use for one buffer.  When this
        limit is reached allocating extra memory for a buffer will cause
        other memory to be freed.  The maximum usable value is about 2000000.
        Use this to work without a limit.  Also see 'maxmemtot'.

                                                *'maxmempattern'* *'mmp'*
'maxmempattern' 'mmp'   number  (default 1000)
                        global
                        {not in Vi}
        Maximum amount of memory (in Kbyte) to use for pattern matching.
        The maximum value is about 2000000.  Use this to work without a limit.
                                                              *E363*
        When Vim runs into the limit it gives an error message and mostly
        behaves like CTRL-C was typed.
        Running into the limit often means that the pattern is very
        inefficient or too complex.  This may already happen with the pattern
        "\(.\)*" on a very long line.  ".*" works much better.
        Vim may run out of memory before hitting the 'maxmempattern' limit.

                                                *'maxmemtot'* *'mmt'*
'maxmemtot' 'mmt'       number  (default between 2048 and 10240 (system
                                 dependent) or half the amount of memory
                                 available)
                        global
                        {not in Vi}
        Maximum amount of memory in Kbyte to use for all buffers together.
        The maximum usable value is about 2000000 (2 Gbyte).  Use this to work
        without a limit.
        On 64 bit machines higher values might work.  But hey, do you really
        need more than 2 Gbyte for text editing?  Keep in mind that text is
        stored in the swap file, one can edit files > 2 Gbyte anyway.  We do
        need the memory to store undo info.
        Also see 'maxmem'.

                                                *'menuitems'* *'mis'*
'menuitems' 'mis'       number  (default 25)
```

```
                        global
                        {not in Vi}
                        {not available when compiled without the |+menu|
                        feature}
        Maximum number of items to use in a menu.  Used for menus that are
        generated from a list of items, e.g., the Buffers menu.  Changing this
        option has no direct effect, the menu must be refreshed first.

                                                *'mkspellmem'* *'msm'*
'mkspellmem' 'msm'      string  (default "460000,2000,500")
                        global
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        Parameters for |:mkspell|.  This tunes when to start compressing the
        word tree.  Compression can be slow when there are many words, but
        it's needed to avoid running out of memory.  The amount of memory used
        per word depends very much on how similar the words are, that's why
        this tuning is complicated.

        There are three numbers, separated by commas:
                {start},{inc},{added}

        For most languages the uncompressed word tree fits in memory.  {start}
        gives the amount of memory in Kbyte that can be used before any
        compression is done.  It should be a bit smaller than the amount of
        memory that is available to Vim.

        When going over the {start} limit the {inc} number specifies the
        amount of memory in Kbyte that can be allocated before another
        compression is done.  A low number means compression is done after
        less words are added, which is slow.  A high number means more memory
        will be allocated.

        After doing compression, {added} times 1024 words can be added before
        the {inc} limit is ignored and compression is done when any extra
        amount of memory is needed.  A low number means there is a smaller
        chance of hitting the {inc} limit, less memory is used but it's
        slower.

        The languages for which these numbers are important are Italian and
        Hungarian.  The default works for when you have about 512 Mbyte.  If
        you have 1 Gbyte you could use: >
                :set mkspellmem=900000,3000,800
<       If you have less than 512 Mbyte |:mkspell| may fail for some
        languages, no matter what you set 'mkspellmem' to.

                                    *'modeline'* *'ml'* *'nomodeline'* *'noml'*
'modeline' 'ml'         boolean (Vim default: on (off for root),
                                 Vi default: off)
                        local to buffer
                                                *'modelines'* *'mls'*
'modelines' 'mls'       number  (default 5)
                        global
                        {not in Vi}
        If 'modeline' is on 'modelines' gives the number of lines that is
        checked for set commands.  If 'modeline' is off or 'modelines' is zero
        no lines are checked.  See |modeline|.
        NOTE: 'modeline' is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                *'modifiable'* *'ma'* *'nomodifiable'* *'noma'*
```

```
'modifiable' 'ma'       boolean (default on)
                        local to buffer
                        {not in Vi}             *E21*
        When off the buffer contents cannot be changed.  The 'fileformat' and
        'fileencoding' options also can't be changed.
        Can be reset on startup with the |-M| command line argument.

                                *'modified'* *'mod'* *'nomodified'* *'nomod'*
'modified' 'mod'        boolean (default off)
                        local to buffer
                        {not in Vi}
        When on, the buffer is considered to be modified.  This option is set
        when:
        1. A change was made to the text since it was last written.  Using the
           |undo| command to go back to the original text will reset the
           option.  But undoing changes that were made before writing the
           buffer will set the option again, since the text is different from
           when it was written.
        2. 'fileformat' or 'fileencoding' is different from its original
           value.  The original value is set when the buffer is read or
           written.  A ":set nomodified" command also resets the original
           values to the current values and the 'modified' option will be
           reset.
           Similarly for 'eol' and 'bomb'.
        This option is not set when a change is made to the buffer as the
        result of a BufNewFile, BufRead/BufReadPost, BufWritePost,
        FileAppendPost or VimLeave autocommand event.  See |gzip-example| for
        an explanation.
        When 'buftype' is "nowrite" or "nofile" this option may be set, but
        will be ignored.

                                                *'more'* *'nomore'*
'more'                  boolean (Vim default: on, Vi default: off)
                        global
                        {not in Vi}
        When on, listings pause when the whole screen is filled.  You will get
        the |more-prompt|.  When this option is off there are no pauses, the
        listing continues until finished.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'mouse'* *E538*
'mouse'                 string  (default "", "a" for GUI, MS-DOS and Win32,
                                                 set to "a" in |defaults.vim|)
                        global
                        {not in Vi}
        Enable the use of the mouse.  Only works for certain terminals
        (xterm, MS-DOS, Win32 |win32-mouse|, QNX pterm, *BSD console with
        sysmouse and Linux console with gpm).  For using the mouse in the
        GUI, see |gui-mouse|.
        The mouse can be enabled for different modes:
                n       Normal mode
                v       Visual mode
                i       Insert mode
                c       Command-line mode
                h       all previous modes when editing a help file
                a       all previous modes
                r       for |hit-enter| and |more-prompt| prompt
        Normally you would enable the mouse in all four modes with: >
                :set mouse=a
<       When the mouse is not enabled, the GUI will still use the mouse for
        modeless selection.  This doesn't move the text cursor.
```

```
        See |mouse-using|.  Also see |'clipboard'|.

        Note: When enabling the mouse in a terminal, copy/paste will use the
        "* register if there is access to an X-server.  The xterm handling of
        the mouse buttons can still be used by keeping the shift key pressed.
        Also see the 'clipboard' option.

                        *'mousefocus'* *'mousef'* *'nomousefocus'* *'nomousef'*
'mousefocus' 'mousef'   boolean (default off)
                        global
                        {not in Vi}
                        {only works in the GUI}
        The window that the mouse pointer is on is automatically activated.
        When changing the window layout or window focus in another way, the
        mouse pointer is moved to the window with keyboard focus.  Off is the
        default because it makes using the pull down menus a little goofy, as
        a pointer transit may activate a window unintentionally.

                        *'mousehide'* *'mh'* *'nomousehide'* *'nomh'*
'mousehide' 'mh'        boolean (default on)
                        global
                        {not in Vi}
                        {only works in the GUI}
        When on, the mouse pointer is hidden when characters are typed.
        The mouse pointer is restored when the mouse is moved.

                                                *'mousemodel'* *'mousem'*
'mousemodel' 'mousem'   string  (default "extend", "popup" for MS-DOS and Win32)
                        global
                        {not in Vi}
        Sets the model to use for the mouse.  The name mostly specifies what
        the right mouse button is used for:
           extend       Right mouse button extends a selection.  This works
                        like in an xterm.
           popup        Right mouse button pops up a menu.  The shifted left
                        mouse button extends a selection.  This works like
                        with Microsoft Windows.
           popup_setpos Like "popup", but the cursor will be moved to the
                        position where the mouse was clicked, and thus the
                        selected operation will act upon the clicked object.
                        If clicking inside a selection, that selection will
                        be acted upon, i.e. no cursor move.  This implies of
                        course, that right clicking outside a selection will
                        end Visual mode.
        Overview of what button does what for each model:
        mouse               extend              popup(_setpos) ~
        left click          place cursor        place cursor
        left drag           start selection     start selection
        shift-left          search word         extend selection
        right click         extend selection    popup menu (place cursor)
        right drag          extend selection    -
        middle click        paste               paste

        In the "popup" model the right mouse button produces a pop-up menu.
        You need to define this first, see |popup-menu|.

        Note that you can further refine the meaning of buttons with mappings.
        See |gui-mouse-mapping|.  But mappings are NOT used for modeless
        selection (because that's handled in the GUI code directly).

        The 'mousemodel' option is set by the |:behave| command.
```

```
                                        *'mouseshape'* *'mouses'* *E547*
'mouseshape' 'mouses'   string  (default "i:beam,r:beam,s:updown,sd:cross,
                                        m:no,ml:up-arrow,v:rightup-arrow")
                        global
                        {not in Vi}
                        {only available when compiled with the |+mouseshape|
                        feature}
        This option tells Vim what the mouse pointer should look like in
        different modes.  The option is a comma separated list of parts, much
        like used for 'guicursor'.  Each part consist of a mode/location-list
        and an argument-list:
                mode-list:shape,mode-list:shape,..
        The mode-list is a dash separated list of these modes/locations:
                        In a normal window: ~
                n       Normal mode
                v       Visual mode
                ve      Visual mode with 'selection' "exclusive" (same as 'v',
                        if not specified)
                o       Operator-pending mode
                i       Insert mode
                r       Replace mode

                        Others: ~
                c       appending to the command-line
                ci      inserting in the command-line
                cr      replacing in the command-line
                m       at the 'Hit ENTER' or 'More' prompts
                ml      idem, but cursor in the last line
                e       any mode, pointer below last window
                s       any mode, pointer on a status line
                sd      any mode, while dragging a status line
                vs      any mode, pointer on a vertical separator line
                vd      any mode, while dragging a vertical separator line
                a       everywhere

        The shape is one of the following:
        avail   name            looks like ~
        w x     arrow           Normal mouse pointer
        w x     blank           no pointer at all (use with care!)
        w x     beam            I-beam
        w x     updown          up-down sizing arrows
        w x     leftright       left-right sizing arrows
        w x     busy            The system's usual busy pointer
        w x     no              The system's usual 'no input' pointer
          x     udsizing        indicates up-down resizing
          x     lrsizing        indicates left-right resizing
          x     crosshair       like a big thin +
          x     hand1           black hand
          x     hand2           white hand
          x     pencil          what you write with
          x     question        big ?
          x     rightup-arrow   arrow pointing right-up
        w x     up-arrow        arrow pointing up
          x     <number>        any X11 pointer number (see X11/cursorfont.h)

        The "avail" column contains a 'w' if the shape is available for Win32,
        x for X11.
        Any modes not specified or shapes not available use the normal mouse
        pointer.

        Example: >
```

```
                :set mouseshape=s:udsizing,m:no
<       will make the mouse turn to a sizing arrow over the status lines and
        indicate no input when the hit-enter prompt is displayed (since
        clicking the mouse has no effect in this state.)

                                                *'mousetime'* *'mouset'*
'mousetime' 'mouset'    number  (default 500)
                        global
                        {not in Vi}
        Only for GUI, MS-DOS, Win32 and Unix with xterm.  Defines the maximum
        time in msec between two mouse clicks for the second click to be
        recognized as a multi click.

                                                *'mzschemedll'*
'mzschemedll'           string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+mzscheme/dyn|
                        feature}
        Specifies the name of the MzScheme shared library. The default is
        DYNAMIC_MZSCH_DLL which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'mzschemegcdll'*
'mzschemegcdll'         string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+mzscheme/dyn|
                        feature}
        Specifies the name of the MzScheme GC shared library. The default is
        DYNAMIC_MZGC_DLL which was specified at compile time.
        The value can be equal to 'mzschemedll' if it includes the GC code.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                    *'mzquantum'* *'mzq'*
'mzquantum' 'mzq'       number  (default 100)
                        global
                        {not in Vi}
                        {not available when compiled without the |+mzscheme|
                        feature}
        The number of milliseconds between polls for MzScheme threads.
        Negative or zero value means no thread scheduling.
        NOTE: This option is set to the Vim default value when 'compatible'
        is reset.

                                                        *'nrformats'* *'nf'*
'nrformats' 'nf'        string  (default "bin,octal,hex",
                                        set to "bin,hex" in |defaults.vim|)
                        local to buffer
                        {not in Vi}
        This defines what bases Vim will consider for numbers when using the
        CTRL-A and CTRL-X commands for adding to and subtracting from a number
        respectively; see |CTRL-A| for more info on these commands.
        alpha   If included, single alphabetical characters will be
                incremented or decremented.  This is useful for a list with a
                letter index a), b), etc.               *octal-nrformats*
        octal   If included, numbers that start with a zero will be considered
                to be octal.  Example: Using CTRL-A on "007" results in "010".
```

```
        hex     If included, numbers starting with "0x" or "0X" will be
                considered to be hexadecimal.  Example: Using CTRL-X on
                "0x100" results in "0x0ff".
        bin     If included, numbers starting with "0b" or "0B" will be
                considered to be binary.  Example: Using CTRL-X on
                "0b1000" subtracts one, resulting in "0b0111".
        Numbers which simply begin with a digit in the range 1-9 are always
        considered decimal.  This also happens for numbers that are not
        recognized as octal or hex.

                                *'number'* *'nu'* *'nonumber'* *'nonu'*
'number' 'nu'           boolean (default off)
                        local to window
        Print the line number in front of each line.  When the 'n' option is
        excluded from 'cpoptions' a wrapped line will not use the column of
        line numbers (this is the default when 'compatible' isn't set).
        The 'numberwidth' option can be used to set the room used for the line
        number.
        When a long, wrapped line doesn't start with the first character, '-'
        characters are put before the number.
        See |hl-LineNr|  and |hl-CursorLineNr| for the highlighting used for
        the number.
                                                *number_relativenumber*
        The 'relativenumber' option changes the displayed number to be
        relative to the cursor.  Together with 'number' there are these
        four combinations (cursor in line 3):

                'nonu'          'nu'            'nonu'          'nu'
                'nornu'         'nornu'         'rnu'           'rnu'

            |apple          |  1 apple      |  2 apple      |  2 apple
            |pear           |  2 pear       |  1 pear       |  1 pear
            |nobody         |  3 nobody     |  0 nobody     |3   nobody
            |there          |  4 there      |  1 there      |  1 there

                                                *'numberwidth'* *'nuw'*
'numberwidth' 'nuw'     number  (Vim default: 4  Vi default: 8)
                        local to window
                        {not in Vi}
                        {only available when compiled with the |+linebreak|
                        feature}
        Minimal number of columns to use for the line number.  Only relevant
        when the 'number' or 'relativenumber' option is set or printing lines
        with a line number. Since one space is always between the number and
        the text, there is one less character for the number itself.
        The value is the minimum width.  A bigger width is used when needed to
        fit the highest line number in the buffer respectively the number of
        rows in the window, depending on whether 'number' or 'relativenumber'
        is set. Thus with the Vim default of 4 there is room for a line number
        up to 999. When the buffer has 1000 lines five columns will be used.
        The minimum value is 1, the maximum value is 10.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'omnifunc'* *'ofu'*
'omnifunc' 'ofu'        string  (default: empty)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+eval|
                        or |+insert_expand| features}
        This option specifies a function to be used for Insert mode omni
        completion with CTRL-X CTRL-O. |i_CTRL-X_CTRL-O|
```

See |complete-functions| for an explanation of how the function is
invoked and what it should return.
This option is usually set by a filetype plugin:
|:filetype-plugin-on|
This option cannot be set from a |modeline| or in the |sandbox|, for
security reasons.

'opendevice' *'odev'* *'noopendevice'* *'noodev'*
'opendevice' 'odev' boolean (default off)
global
{not in Vi}
{only for MS-DOS, MS-Windows and OS/2}
Enable reading and writing from devices. This may get Vim stuck on a
device that can be opened but doesn't actually do the I/O. Therefore
it is off by default.
Note that on MS-Windows editing "aux.h", "lpt1.txt" and the like also
result in editing a device.

'operatorfunc' *'opfunc'*
'operatorfunc' 'opfunc' string (default: empty)
global
{not in Vi}
This option specifies a function to be called by the |g@| operator.
See |:map-operator| for more info and an example.

This option cannot be set from a |modeline| or in the |sandbox|, for
security reasons.

'osfiletype' *'oft'*
'osfiletype' 'oft' string (default: "")
local to buffer
{not in Vi}
This option was supported on RISC OS, which has been removed.

'packpath' *'pp'*
'packpath' 'pp' string (default: see 'runtimepath')
{not in Vi}
Directories used to find packages. See |packages|.

'paragraphs' *'para'*
'paragraphs' 'para' string (default "IPLPPPQPP TPHPLIPpLpItpplpipbp")
global
Specifies the nroff macros that separate paragraphs. These are pairs
of two letters (see |object-motions|).

'paste' *'nopaste'*
'paste' boolean (default off)
global
{not in Vi}
Put Vim in Paste mode. This is useful if you want to cut or copy
some text from one window and paste it in Vim. This will avoid
unexpected effects.
Setting this option is useful when using Vim in a terminal, where Vim
cannot distinguish between typed text and pasted text. In the GUI, Vim
knows about pasting and will mostly do the right thing without 'paste'
being set. The same is true for a terminal where Vim handles the
mouse clicks itself.

```
        This option is reset when starting the GUI.  Thus if you set it in
        your .vimrc it will work in a terminal, but not in the GUI.  Setting
        'paste' in the GUI has side effects: e.g., the Paste toolbar button
        will no longer work in Insert mode, because it uses a mapping.
        When the 'paste' option is switched on (also when it was already on):
                - mapping in Insert mode and Command-line mode is disabled
                - abbreviations are disabled
                - 'autoindent' is reset
                - 'expandtab' is reset
                - 'formatoptions' is used like it is empty
                - 'revins' is reset
                - 'ruler' is reset
                - 'showmatch' is reset
                - 'smartindent' is reset
                - 'smarttab' is reset
                - 'softtabstop' is set to 0
                - 'textwidth' is set to 0
                - 'wrapmargin' is set to 0
        These options keep their value, but their effect is disabled:
                - 'cindent'
                - 'indentexpr'
                - 'lisp'
        NOTE: When you start editing another file while the 'paste' option is
        on, settings from the modelines or autocommands may change the
        settings again, causing trouble when pasting text.  You might want to
        set the 'paste' option again.
        When the 'paste' option is reset the mentioned options are restored to
        the value before the moment 'paste' was switched from off to on.
        Resetting 'paste' before ever setting it does not have any effect.
        Since mapping doesn't work while 'paste' is active, you need to use
        the 'pastetoggle' option to toggle the 'paste' option with some key.

                                                *'pastetoggle'* *'pt'*
'pastetoggle' 'pt'      string  (default "")
                        global
                        {not in Vi}
        When non-empty, specifies the key sequence that toggles the 'paste'
        option.  This is like specifying a mapping: >
            :map {keys} :set invpaste<CR>
<       Where {keys} is the value of 'pastetoggle'.
        The difference is that it will work even when 'paste' is set.
        'pastetoggle' works in Insert mode and Normal mode, but not in
        Command-line mode.
        Mappings are checked first, thus overrule 'pastetoggle'.  However,
        when 'paste' is on mappings are ignored in Insert mode, thus you can do
        this: >
            :map <F10> :set paste<CR>
            :map <F11> :set nopaste<CR>
            :imap <F10> <C-O>:set paste<CR>
            :imap <F11> <nop>
            :set pastetoggle=<F11>
<       This will make <F10> start paste mode and <F11> stop paste mode.
        Note that typing <F10> in paste mode inserts "<F10>", since in paste
        mode everything is inserted literally, except the 'pastetoggle' key
        sequence.
        When the value has several bytes 'ttimeoutlen' applies.

                                                *'pex'* *'patchexpr'*
'patchexpr' 'pex'       string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+diff|
```

```
                        feature}
        Expression which is evaluated to apply a patch to a file and generate
        the resulting new version of the file.  See |diff-patchexpr|.

                                          *'patchmode'* *'pm'* *E205* *E206*
'patchmode' 'pm'        string  (default "")
                        global
                        {not in Vi}
        When non-empty the oldest version of a file is kept.  This can be used
        to keep the original version of a file if you are changing files in a
        source distribution.  Only the first time that a file is written a
        copy of the original file will be kept.  The name of the copy is the
        name of the original file with the string in the 'patchmode' option
        appended.  This option should start with a dot.  Use a string like
        ".org".  'backupdir' must not be empty for this to work (Detail: The
        backup file is renamed to the patchmode file after the new file has
        been successfully written, that's why it must be possible to write a
        backup file).  If there was no file to be backed up, an empty file is
        created.
        When the 'backupskip' pattern matches, a patchmode file is not made.
        Using 'patchmode' for compressed files appends the extension at the
        end (e.g., "file.gz.orig"), thus the resulting name isn't always
        recognized as a compressed file.
        Only normal file name characters can be used, "/\*?[|<>" are illegal.

                                  *'path'* *'pa'* *E343* *E345* *E347* *E854*
'path' 'pa'             string  (default on Unix: ".,/usr/include,,"
                                    on OS/2:       ".,/emx/include,,"
                                    other systems: ".,,")
                        global or local to buffer |global-local|
                        {not in Vi}
        This is a list of directories which will be searched when using the
        |gf|, [f, ]f, ^Wf, |:find|, |:sfind|, |:tabfind| and other commands,
        provided that the file being searched for has a relative path (not
        starting with "/", "./" or "../").  The directories in the 'path'
        option may be relative or absolute.
        - Use commas to separate directory names: >
                :set path=.,/usr/local/include,/usr/include
<       - Spaces can also be used to separate directory names (for backwards
          compatibility with version 3.0).  To have a space in a directory
          name, precede it with an extra backslash, and escape the space: >
                :set path=.,/dir/with\\\ space
<       - To include a comma in a directory name precede it with an extra
          backslash: >
                :set path=.,/dir/with\\,comma
<       - To search relative to the directory of the current file, use: >
                :set path=.
<       - To search in the current directory use an empty string between two
          commas: >
                :set path=,,
<       - A directory name may end in a ':' or '/'.
        - Environment variables are expanded |:set_env|.
        - When using |netrw.vim| URLs can be used.  For example, adding
          "http://www.vim.org" will make ":find index.html" work.
        - Search upwards and downwards in a directory tree using "*", "**" and
          ";".  See |file-searching| for info and syntax.
          {not available when compiled without the |+path_extra| feature}
        - Careful with '\' characters, type two to get one in the option: >
                :set path=.,c:\\include
<         Or just use '/' instead: >
                :set path=.,c:/include
<       Don't forget "." or files won't even be found in the same directory as
```

```
        the file!
        The maximum length is limited.  How much depends on the system, mostly
        it is something like 256 or 1024 characters.
        You can check if all the include files are found, using the value of
        'path', see |:checkpath|.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        directories from the list.  This avoids problems when a future version
        uses another default.  To remove the current directory use: >
                :set path-=
<       To add the current directory use: >
                :set path+=
<       To use an environment variable, you probably need to replace the
        separator.  Here is an example to append $INCL, in which directory
        names are separated with a semi-colon: >
                :let &path = &path . "," . substitute($INCL, ';', ',', 'g')
<       Replace the ';' with a ':' or whatever separator is used.  Note that
        this doesn't work when $INCL contains a comma or white space.

                                                *'perldll'*
'perldll'               string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+perl/dyn|
                        feature}
        Specifies the name of the Perl shared library. The default is
        DYNAMIC_PERL_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                        *'preserveindent'* *'pi'* *'nopreserveindent'* *'nopi'*
'preserveindent' 'pi'   boolean (default off)
                        local to buffer
                        {not in Vi}
        When changing the indent of the current line, preserve as much of the
        indent structure as possible.  Normally the indent is replaced by a
        series of tabs followed by spaces as required (unless |'expandtab'| is
        enabled, in which case only spaces are used).  Enabling this option
        means the indent will preserve as many existing characters as possible
        for indenting, and only add additional tabs or spaces as required.
        'expandtab' does not apply to the preserved white space, a Tab remains
        a Tab.
        NOTE: When using ">>" multiple times the resulting indent is a mix of
        tabs and spaces.  You might not like this.
        NOTE: This option is reset when 'compatible' is set.
        Also see 'copyindent'.
        Use |:retab| to clean up white space.

                                        *'previewheight'* *'pvh'*
'previewheight' 'pvh'   number (default 12)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows| or
                        |+quickfix| features}
        Default height for a preview window.  Used for |:ptag| and associated
        commands.  Used for |CTRL-W_}| when no count is given.

                                        *'previewwindow'* *'nopreviewwindow'*
                                        *'pvw'* *'nopvw'* *E590*
'previewwindow' 'pvw'   boolean (default off)
                        local to window
                        {not in Vi}
```

```
                        {not available when compiled without the |+windows| or
                        |+quickfix| features}
        Identifies the preview window.  Only one window can have this option
        set.  It's normally not set directly, but by using one of the commands
        |:ptag|, |:pedit|, etc.

                                                *'printdevice'* *'pdev'*
'printdevice' 'pdev'    string  (default empty)
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|
                        feature}
        The name of the printer to be used for |:hardcopy|.
        See |pdev-option|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'printencoding'* *'penc'*
'printencoding' 'penc'  String  (default empty, except for some systems)
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|
                        and |+postscript| features}
        Sets the character encoding used when printing.
        See |penc-option|.

                                                *'printexpr'* *'pexpr'*
'printexpr' 'pexpr'     String  (default: see below)
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|
                        and |+postscript| features}
        Expression used to print the PostScript produced with |:hardcopy|.
        See |pexpr-option|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'printfont'* *'pfn'*
'printfont' 'pfn'       string  (default "courier")
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|
                        feature}
        The name of the font that will be used for |:hardcopy|.
        See |pfn-option|.

                                                *'printheader'* *'pheader'*
'printheader' 'pheader'  string  (default "%<%f%h%m%=Page %N")
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|
                        feature}
        The format of the header produced in |:hardcopy| output.
        See |pheader-option|.

                                                *'printmbcharset'* *'pmbcs'*
'printmbcharset' 'pmbcs'  string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|,
                        |+postscript| and |+multi_byte| features}
        The CJK character set to be used for CJK output from |:hardcopy|.
```

```
        See |pmbcs-option|.

                                                *'printmbfont'* *'pmbfn'*
'printmbfont' 'pmbfn'   string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with the |+printer|,
                        |+postscript| and |+multi_byte| features}
        List of font names to be used for CJK output from |:hardcopy|.
        See |pmbfn-option|.

                                                *'printoptions'* *'popt'*
'printoptions' 'popt' string (default "")
                        global
                        {not in Vi}
                        {only available when compiled with |+printer| feature}
        List of items that control the format of the output of |:hardcopy|.
        See |popt-option|.

                                                *'prompt'* *'noprompt'*
'prompt'                boolean (default on)
                        global
        When on a ":" prompt is used in Ex mode.

                                                *'pumheight'* *'ph'*
'pumheight' 'ph'        number  (default 0)
                        global
                        {not available when compiled without the
                        |+insert_expand| feature}
                        {not in Vi}
        Determines the maximum number of items to show in the popup menu for
        Insert mode completion.  When zero as much space as available is used.
        |ins-completion-menu|.

                                                *'pythondll'*
'pythondll'             string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+python/dyn|
                        feature}
        Specifies the name of the Python 2.x shared library. The default is
        DYNAMIC_PYTHON_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'pythonthreedll'*
'pythonthreedll'        string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+python3/dyn|
                        feature}
        Specifies the name of the Python 3 shared library. The default is
        DYNAMIC_PYTHON3_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'pyxversion'* *'pyx'*
'pyxversion' 'pyx'      number  (default depends on the build)
                        global
                        {not in Vi}
```

```
                        {only available when compiled with the |+python| or
                        the |+python3| feature}
        Specifies the python version used for pyx* functions and commands
        |python_x|.  The default value is as follows:

                Compiled with               Default ~
                |+python| and |+python3|        0
                only |+python|                  2
                only |+python3|                 3

        Available values are 0, 2 and 3.
        If 'pyxversion' is 0, it is set to 2 or 3 after the first execution of
        any python2/3 commands or functions.  E.g. `:py` sets to 2, and `:py3`
        sets to 3. `:pyx` sets it to 3 if Python 3 is available, otherwise sets
        to 2 if Python 2 is available.
        See also: |has-pythonx|

        If Vim is compiled with only |+python| or |+python3| setting
        'pyxversion' has no effect.  The pyx* functions and commands are
        always the same as the compiled version.

        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'quoteescape'* *'qe'*
'quoteescape' 'qe'      string  (default "\")
                        local to buffer
                        {not in Vi}
        The characters that are used to escape quotes in a string.  Used for
        objects like a', a" and a` |a'|.
        When one of the characters in this option is found inside a string,
        the following character will be skipped.  The default value makes the
        text "foo\"bar\\" considered to be one string.

                                    *'readonly'* *'ro'* *'noreadonly'* *'noro'*
'readonly' 'ro'         boolean (default off)
                        local to buffer
        If on, writes fail unless you use a '!'.  Protects you from
        accidentally overwriting a file.  Default on when Vim is started
        in read-only mode ("vim -R") or when the executable is called "view".
        When using ":w!" the 'readonly' option is reset for the current
        buffer, unless the 'Z' flag is in 'cpoptions'.
        {not in Vi:}  When using the ":view" command the 'readonly' option is
        set for the newly edited buffer.
        See 'modifiable' for disallowing changes to the buffer.

                                                *'redrawtime'* *'rdt'*
'redrawtime' 'rdt'      number  (default 2000)
                        global
                        {not in Vi}
                        {only available when compiled with the |+reltime|
                        feature}
        The time in milliseconds for redrawing the display.  This applies to
        searching for patterns for 'hlsearch', |:match| highlighting an syntax
        highlighting.
        When redrawing takes more than this many milliseconds no further
        matches will be highlighted.
        For syntax highlighting the time applies per window.  When over the
        limit syntax highlighting is disabled until |CTRL-L| is used.
        This is used to avoid that Vim hangs when using a very complicated
        pattern.
```

'regexpengine' *'re'*
'regexpengine' 're'	number (default 0)
			global
			{not in Vi}
	This selects the default regexp engine. |two-engines|
	The possible values are:
		0	automatic selection
		1	old engine
		2	NFA engine
	Note that when using the NFA engine and the pattern contains something
	that is not supported the pattern will not match. This is only useful
	for debugging the regexp engine.
	Using automatic selection enables Vim to switch the engine, if the
	default engine becomes too costly. E.g., when the NFA engine uses too
	many states. This should prevent Vim from hanging on a combination of
	a complex pattern with long text.

		'relativenumber' *'rnu'* *'norelativenumber'* *'nornu'*
'relativenumber' 'rnu'	boolean (default off)
			local to window
			{not in Vi}
	Show the line number relative to the line with the cursor in front of
	each line. Relative line numbers help you use the |count| you can
	precede some vertical motion commands (e.g. j k + -) with, without
	having to calculate it yourself. Especially useful in combination with
	other commands (e.g. y d c < > gq gw =).
	When the 'n' option is excluded from 'cpoptions' a wrapped
	line will not use the column of line numbers (this is the default when
	'compatible' isn't set).
	The 'numberwidth' option can be used to set the room used for the line
	number.
	When a long, wrapped line doesn't start with the first character, '-'
	characters are put before the number.
	See |hl-LineNr| and |hl-CursorLineNr| for the highlighting used for
	the number.

	The number in front of the cursor line also depends on the value of
	'number', see |number_relativenumber| for all combinations of the two
	options.

'remap' *'noremap'*
'remap'			boolean (default on)
			global
	Allows for mappings to work recursively. If you do not want this for
	a single entry, use the :noremap[!] command.
	NOTE: To avoid portability problems with Vim scripts, always keep
	this option at the default "on". Only switch it off when working with
	old Vi scripts.

'renderoptions' *'rop'*
'renderoptions' 'rop'	string (default: empty)
			global
			{not in Vi}
			{only available when compiled with GUI and DIRECTX on
			MS-Windows}
	Select a text renderer and set its options. The options depend on the
	renderer.

	Syntax: >
		set rop=type:{renderer}(,{name}:{value})*
<
	Currently, only one optional renderer is available.

```
        render  behavior    ~
        directx Vim will draw text using DirectX (DirectWrite).  It makes
                drawn glyphs more beautiful than default GDI.
                It requires 'encoding' is "utf-8", and only works on
                MS-Windows Vista or newer version.

                Options:
                  name      meaning             type    value       ~
                  gamma     gamma               float   1.0 - 2.2 (maybe)
                  contrast  enhancedContrast    float   (unknown)
                  level     clearTypeLevel      float   (unknown)
                  geom      pixelGeometry       int     0 - 2 (see below)
                  renmode   renderingMode       int     0 - 6 (see below)
                  taamode   textAntialiasMode   int     0 - 3 (see below)

                See this URL for detail:
                  http://msdn.microsoft.com/en-us/library/dd368190.aspx

                For geom: structure of a device pixel.
                  0 - DWRITE_PIXEL_GEOMETRY_FLAT
                  1 - DWRITE_PIXEL_GEOMETRY_RGB
                  2 - DWRITE_PIXEL_GEOMETRY_BGR

                See this URL for detail:
                  http://msdn.microsoft.com/en-us/library/dd368114.aspx

                For renmode: method of rendering glyphs.
                  0 - DWRITE_RENDERING_MODE_DEFAULT
                  1 - DWRITE_RENDERING_MODE_ALIASED
                  2 - DWRITE_RENDERING_MODE_GDI_CLASSIC
                  3 - DWRITE_RENDERING_MODE_GDI_NATURAL
                  4 - DWRITE_RENDERING_MODE_NATURAL
                  5 - DWRITE_RENDERING_MODE_NATURAL_SYMMETRIC
                  6 - DWRITE_RENDERING_MODE_OUTLINE

                See this URL for detail:
                  http://msdn.microsoft.com/en-us/library/dd368118.aspx

                For taamode: antialiasing mode used for drawing text.
                  0 - D2D1_TEXT_ANTIALIAS_MODE_DEFAULT
                  1 - D2D1_TEXT_ANTIALIAS_MODE_CLEARTYPE
                  2 - D2D1_TEXT_ANTIALIAS_MODE_GRAYSCALE
                  3 - D2D1_TEXT_ANTIALIAS_MODE_ALIASED

                See this URL for detail:
                  http://msdn.microsoft.com/en-us/library/dd368170.aspx

                Example: >
                  set encoding=utf-8
                  set gfn=Ricty_Diminished:h12:cSHIFTJIS
                  set rop=type:directx
<
                If select a raster font (Courier, Terminal or FixedSys) to
                'guifont', it fallbacks to be drawn by GDI automatically.

        Other render types are currently not supported.

                                                *'report'*
'report'                number  (default 2)
                        global
        Threshold for reporting number of lines changed.  When the number of
```

```
        changed lines is more than 'report' a message will be given for most
        ":" commands.  If you want it always, set 'report' to 0.
        For the ":substitute" command the number of substitutions is used
        instead of the number of lines.

                         *'restorescreen'* *'rs'* *'norestorescreen'* *'nors'*
'restorescreen' 'rs'    boolean (default on)
                        global
                        {not in Vi}  {only in Windows 95/NT console version}
        When set, the screen contents is restored when exiting Vim.  This also
        happens when executing external commands.

        For non-Windows Vim: You can set or reset the 't_ti' and 't_te'
        options in your .vimrc.  To disable restoring:
                set t_ti= t_te=
        To enable restoring (for an xterm):
                set t_ti=^[7^[[r^[[?47h t_te=^[[?47l^[8
        (Where ^[ is an <Esc>, type CTRL-V <Esc> to insert it)

                                *'revins'* *'ri'* *'norevins'* *'nori'*
'revins' 'ri'           boolean (default off)
                        global
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        Inserting characters in Insert mode will work backwards.  See "typing
        backwards" |ins-reverse|.  This option can be toggled with the CTRL-_
        command in Insert mode, when 'allowrevins' is set.
        NOTE: This option is reset when 'compatible' is set.
        This option is reset when 'paste' is set and restored when 'paste' is
        reset.

                                 *'rightleft'* *'rl'* *'norightleft'* *'norl'*
'rightleft' 'rl'        boolean (default off)
                        local to window
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        When on, display orientation becomes right-to-left, i.e., characters
        that are stored in the file appear from the right to the left.
        Using this option, it is possible to edit files for languages that
        are written from the right to the left such as Hebrew and Arabic.
        This option is per window, so it is possible to edit mixed files
        simultaneously, or to view the same file in both ways (this is
        useful whenever you have a mixed text file with both right-to-left
        and left-to-right strings so that both sets are displayed properly
        in different windows).  Also see |rileft.txt|.

                        *'rightleftcmd'* *'rlc'*
'rightleftcmd' 'rlc'    string  (default "search")
                        local to window
                        {not in Vi}
                        {only available when compiled with the |+rightleft|
                        feature}
        Each word in this option enables the command line editing to work in
        right-to-left mode for a group of commands:

                search          "/" and "?" commands

        This is useful for languages such as Hebrew, Arabic and Farsi.
        The 'rightleft' option must be set for 'rightleftcmd' to take effect.
```

```
                                                *'rubydll'*
'rubydll'               string  (default: depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+ruby/dyn|
                        feature}
        Specifies the name of the Ruby shared library. The default is
        DYNAMIC_RUBY_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                        *'ruler'* *'ru'* *'noruler'* *'noru'*
'ruler' 'ru'            boolean (default off, set in |defaults.vim|)
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+cmdline_info| feature}
        Show the line and column number of the cursor position, separated by a
        comma.  When there is room, the relative position of the displayed
        text in the file is shown on the far right:
                Top     first line is visible
                Bot     last line is visible
                All     first and last line are visible
                45%     relative position in the file
        If 'rulerformat' is set, it will determine the contents of the ruler.
        Each window has its own ruler.  If a window has a status line, the
        ruler is shown there.  Otherwise it is shown in the last line of the
        screen.  If the statusline is given by 'statusline' (i.e. not empty),
        this option takes precedence over 'ruler' and 'rulerformat'
        If the number of characters displayed is different from the number of
        bytes in the text (e.g., for a TAB or a multi-byte character), both
        the text column (byte number) and the screen column are shown,
        separated with a dash.
        For an empty line "0-1" is shown.
        For an empty buffer the line number will also be zero: "0,0-1".
        This option is reset when 'paste' is set and restored when 'paste' is
        reset.
        If you don't want to see the ruler all the time but want to know where
        you are, use "g CTRL-G" |g_CTRL-G|.
        NOTE: This option is reset when 'compatible' is set.

                                                *'rulerformat'* *'ruf'*
'rulerformat' 'ruf'     string  (default empty)
                        global
                        {not in Vi}
                        {not available when compiled without the |+statusline|
                        feature}
        When this option is not empty, it determines the content of the ruler
        string, as displayed for the 'ruler' option.
        The format of this option is like that of 'statusline'.
        The default ruler width is 17 characters.  To make the ruler 15
        characters wide, put "%15(" at the start and "%)" at the end.
        Example: >
                :set rulerformat=%15(%c%V\ %p%%)
<
                                *'runtimepath'* *'rtp'* *vimfiles*
'runtimepath' 'rtp'     string  (default:
                                        Unix: "$HOME/.vim,
                                                $VIM/vimfiles,
                                                $VIMRUNTIME,
                                                $VIM/vimfiles/after,
```

```
                                        $HOME/.vim/after"
                                Amiga: "home:vimfiles,
                                        $VIM/vimfiles,
                                        $VIMRUNTIME,
                                        $VIM/vimfiles/after,
                                        home:vimfiles/after"
                                PC, OS/2: "$HOME/vimfiles,
                                        $VIM/vimfiles,
                                        $VIMRUNTIME,
                                        $VIM/vimfiles/after,
                                        $HOME/vimfiles/after"
                                Macintosh: "$VIM:vimfiles,
                                        $VIMRUNTIME,
                                        $VIM:vimfiles:after"
                                RISC-OS: "Choices:vimfiles,
                                        $VIMRUNTIME,
                                        Choices:vimfiles/after"
                                VMS: "sys$login:vimfiles,
                                        $VIM/vimfiles,
                                        $VIMRUNTIME,
                                        $VIM/vimfiles/after,
                                        sys$login:vimfiles/after")
                global
                {not in Vi}
This is a list of directories which will be searched for runtime
files:
  filetype.vim  filetypes by file name |new-filetype|
  scripts.vim   filetypes by file contents |new-filetype-scripts|
  autoload/     automatically loaded scripts |autoload-functions|
  colors/       color scheme files |:colorscheme|
  compiler/     compiler files |:compiler|
  doc/          documentation |write-local-help|
  ftplugin/     filetype plugins |write-filetype-plugin|
  indent/       indent scripts |indent-expression|
  keymap/       key mapping files |mbyte-keymap|
  lang/         menu translations |:menutrans|
  menu.vim      GUI menus |menu.vim|
  pack/         packages |:packadd|
  plugin/       plugin scripts |write-plugin|
  print/        files for printing |postscript-print-encoding|
  spell/        spell checking files |spell|
  syntax/       syntax files |mysyntaxfile|
  tutor/        files for vimtutor |tutor|

And any other file searched for with the |:runtime| command.

The defaults for most systems are setup to search five locations:
1. In your home directory, for your personal preferences.
2. In a system-wide Vim directory, for preferences from the system
   administrator.
3. In $VIMRUNTIME, for files distributed with Vim.
                                                *after-directory*
4. In the "after" directory in the system-wide Vim directory.  This is
   for the system administrator to overrule or add to the distributed
   defaults (rarely needed)
5. In the "after" directory in your home directory.  This is for
   personal preferences to overrule or add to the distributed defaults
   or system-wide settings (rarely needed).

More entries are added when using |packages|.  If it gets very long
then `:set rtp` will be truncated, use `:echo &rtp` to see the full
string.
```

```
        Note that, unlike 'path', no wildcards like "**" are allowed.  Normal
        wildcards are allowed, but can significantly slow down searching for
        runtime files.  For speed, use as few items as possible and avoid
        wildcards.
        See |:runtime|.
        Example: >
                :set runtimepath=~/vimruntime,/mygroup/vim,$VIMRUNTIME
<       This will use the directory "~/vimruntime" first (containing your
        personal Vim runtime files), then "/mygroup/vim" (shared between a
        group of people) and finally "$VIMRUNTIME" (the distributed runtime
        files).
        You probably should always include $VIMRUNTIME somewhere, to use the
        distributed runtime files.  You can put a directory before $VIMRUNTIME
        to find files which replace a distributed runtime files.  You can put
        a directory after $VIMRUNTIME to find files which add to distributed
        runtime files.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'scroll'* *'scr'*
'scroll' 'scr'          number  (default: half the window height)
                        local to window
        Number of lines to scroll with CTRL-U and CTRL-D commands.  Will be
        set to half the number of lines in the window when the window size
        changes.  If you give a count to the CTRL-U or CTRL-D command it will
        be used as the new value for 'scroll'.  Reset to half the window
        height with ":set scroll=0".   {Vi is a bit different: 'scroll' gives
        the number of screen lines instead of file lines, makes a difference
        when lines wrap}

                        *'scrollbind'* *'scb'* *'noscrollbind'* *'noscb'*
'scrollbind' 'scb'      boolean  (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+scrollbind|
                        feature}
        See also |scroll-binding|.  When this option is set, the current
        window scrolls as other scrollbind windows (windows that also have
        this option set) scroll.  This option is useful for viewing the
        differences between two versions of a file, see 'diff'.
        See |'scrollopt'| for options that determine how this option should be
        interpreted.
        This option is mostly reset when splitting a window to edit another
        file.  This means that ":split | edit file" results in two windows
        with scroll-binding, but ":split file" does not.

                                                *'scrolljump'* *'sj'*
'scrolljump' 'sj'       number  (default 1)
                        global
                        {not in Vi}
        Minimal number of lines to scroll when the cursor gets off the
        screen (e.g., with "j").  Not used for scroll commands (e.g., CTRL-E,
        CTRL-D).  Useful if your terminal scrolls very slowly.
        When set to a negative number from -1 to -100 this is used as the
        percentage of the window height.  Thus -50 scrolls half the window
        height.
        NOTE: This option is set to 1 when 'compatible' is set.

                                                *'scrolloff'* *'so'*
'scrolloff' 'so'        number  (default 0, set to 5 in |defaults.vim|)
                        global
```

```
                        {not in Vi}
        Minimal number of screen lines to keep above and below the cursor.
        This will make some context visible around where you are working.  If
        you set it to a very large value (999) the cursor line will always be
        in the middle of the window (except at the start or end of the file or
        when long lines wrap).
        For scrolling horizontally see 'sidescrolloff'.
        NOTE: This option is set to 0 when 'compatible' is set.

                                                *'scrollopt'* *'sbo'*
'scrollopt' 'sbo'       string  (default "ver,jump")
                        global
                        {not available when compiled without the |+scrollbind|
                        feature}
                        {not in Vi}
        This is a comma-separated list of words that specifies how
        'scrollbind' windows should behave.  'sbo' stands for ScrollBind
        Options.
        The following words are available:
            ver         Bind vertical scrolling for 'scrollbind' windows
            hor         Bind horizontal scrolling for 'scrollbind' windows
            jump        Applies to the offset between two windows for vertical
                        scrolling.  This offset is the difference in the first
                        displayed line of the bound windows.  When moving
                        around in a window, another 'scrollbind' window may
                        reach a position before the start or after the end of
                        the buffer.  The offset is not changed though, when
                        moving back the 'scrollbind' window will try to scroll
                        to the desired position when possible.
                        When now making that window the current one, two
                        things can be done with the relative offset:
                        1. When "jump" is not included, the relative offset is
                           adjusted for the scroll position in the new current
                           window.  When going back to the other window, the
                           new relative offset will be used.
                        2. When "jump" is included, the other windows are
                           scrolled to keep the same relative offset.  When
                           going back to the other window, it still uses the
                           same relative offset.
        Also see |scroll-binding|.
        When 'diff' mode is active there always is vertical scroll binding,
        even when "ver" isn't there.

                                                *'sections'* *'sect'*
'sections' 'sect'       string  (default "SHNHH HUnhsh")
                        global
        Specifies the nroff macros that separate sections.  These are pairs of
        two letters (See |object-motions|).  The default makes a section start
        at the nroff macros ".SH", ".NH", ".H", ".HU", ".nh" and ".sh".

                                                *'secure'* *'nosecure'* *E523*
'secure'                boolean (default off)
                        global
                        {not in Vi}
        When on, ":autocmd", shell and write commands are not allowed in
        ".vimrc" and ".exrc" in the current directory and map commands are
        displayed.  Switch it off only if you know that you will not run into
        problems, or when the 'exrc' option is off.  On Unix this option is
        only used if the ".vimrc" or ".exrc" is not owned by you.  This can be
        dangerous if the systems allows users to do a "chown".  You better set
        'secure' at the end of your ~/.vimrc then.
        This option cannot be set from a |modeline| or in the |sandbox|, for
```

```
        security reasons.

                                                *'selection'* *'sel'*
'selection' 'sel'       string  (default "inclusive")
                        global
                        {not in Vi}
        This option defines the behavior of the selection.  It is only used
        in Visual and Select mode.
        Possible values:
           value        past line     inclusive ~
           old             no           yes
           inclusive       yes          yes
           exclusive       yes          no
        "past line" means that the cursor is allowed to be positioned one
        character past the line.
        "inclusive" means that the last character of the selection is included
        in an operation.  For example, when "x" is used to delete the
        selection.
        When "old" is used and 'virtualedit' allows the cursor to move past
        the end of line the line break still isn't included.
        Note that when "exclusive" is used and selecting from the end
        backwards, you cannot include the last character of a line, when
        starting in Normal mode and 'virtualedit' empty.

        The 'selection' option is set by the |:behave| command.

                                                *'selectmode'* *'slm'*
'selectmode' 'slm'      string  (default "")
                        global
                        {not in Vi}
        This is a comma separated list of words, which specifies when to start
        Select mode instead of Visual mode, when a selection is started.
        Possible values:
           mouse        when using the mouse
           key          when using shifted special keys
           cmd          when using "v", "V" or CTRL-V
        See |Select-mode|.
        The 'selectmode' option is set by the |:behave| command.

                                                *'sessionoptions'* *'ssop'*
'sessionoptions' 'ssop' string  (default: "blank,buffers,curdir,folds,
                                                help,options,tabpages,winsize")
                        global
                        {not in Vi}
                        {not available when compiled without the |+mksession|
                        feature}
        Changes the effect of the |:mksession| command.  It is a comma
        separated list of words.  Each word enables saving and restoring
        something:
           word         save and restore ~
           blank        empty windows
           buffers      hidden and unloaded buffers, not just those in windows
           curdir       the current directory
           folds        manually created folds, opened/closed folds and local
                        fold options
           globals      global variables that start with an uppercase letter
                        and contain at least one lowercase letter.  Only
                        String and Number types are stored.
           help         the help window
           localoptions options and mappings local to a window or buffer (not
                        global values for local options)
           options      all options and mappings (also global values for local
```

```
                        options)
            resize      size of the Vim window: 'lines' and 'columns'
            sesdir      the directory in which the session file is located
                        will become the current directory (useful with
                        projects accessed over a network from different
                        systems)
            slash       backslashes in file names replaced with forward
                        slashes
            tabpages    all tab pages; without this only the current tab page
                        is restored, so that you can make a session for each
                        tab page separately
            unix        with Unix end-of-line format (single <NL>), even when
                        on Windows or DOS
            winpos      position of the whole Vim window
            winsize     window sizes

        Don't include both "curdir" and "sesdir".
        When neither "curdir" nor "sesdir" is included, file names are stored
        with absolute paths.
        "slash" and "unix" are useful on Windows when sharing session files
        with Unix.  The Unix version of Vim cannot source dos format scripts,
        but the Windows version of Vim can source unix format scripts.

                                                *'shell'* *'sh'* *E91*
'shell' 'sh'            string  (default $SHELL or "sh",
                                 MS-DOS and Win32: "command.com" or
                                 "cmd.exe", OS/2: "cmd")
                        global
        Name of the shell to use for ! and :! commands.  When changing the
        value also check these options: 'shelltype', 'shellpipe', 'shellslash'
        'shellredir', 'shellquote', 'shellxquote' and 'shellcmdflag'.
        It is allowed to give an argument to the command, e.g.  "csh -f".
        See |option-backslash| about including spaces and backslashes.
        Environment variables are expanded |:set_env|.
        If the name of the shell contains a space, you might need to enclose
        it in quotes.  Example: >
                :set shell=\"c:\program\ files\unix\sh.exe\"\ -f
<       Note the backslash before each quote (to avoid starting a comment) and
        each space (to avoid ending the option value).  Also note that the
        "-f" is not inside the quotes, because it is not part of the command
        name.  And Vim automagically recognizes the backslashes that are path
        separators.
        Under MS-Windows, when the executable ends in ".com" it must be
        included.  Thus setting the shell to "command.com" or "4dos.com"
        works, but "command" and "4dos" do not work for all commands (e.g.,
        filtering).
        For unknown reasons, when using "4dos.com" the current directory is
        changed to "C:\".  To avoid this set 'shell' like this: >
                :set shell=command.com\ /c\ 4dos
<       This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'shellcmdflag'* *'shcf'*
'shellcmdflag' 'shcf'   string  (default: "-c";
                                 MS-DOS and Win32, when 'shell' does not
                                 contain "sh" somewhere: "/c")
                        global
                        {not in Vi}
        Flag passed to the shell to execute "!" and ":!" commands; e.g.,
        "bash.exe -c ls" or "command.com /c dir".  For the MS-DOS-like
        systems, the default is set according to the value of 'shell', to
        reduce the need to set this option by the user.
```

On Unix it can have more than one flag. Each white space separated part is passed as an argument to the shell command.
See |option-backslash| about including spaces and backslashes.
Also see |dos-shell| for MS-DOS and MS-Windows.
This option cannot be set from a |modeline| or in the |sandbox|, for security reasons.

'shellpipe' *'sp'*
'shellpipe' 'sp' string (default ">", "| tee", "|& tee" or "2>&1| tee")
global
{not in Vi}
{not available when compiled without the |+quickfix| feature}
String to be used to put the output of the ":make" command in the error file. See also |:make_makeprg|. See |option-backslash| about including spaces and backslashes.
The name of the temporary file can be represented by "%s" if necessary (the file name is appended automatically if no %s appears in the value of this option).
For the Amiga and MS-DOS the default is ">". The output is directly saved in a file and not echoed to the screen.
For Unix the default it "| tee". The stdout of the compiler is saved in a file and echoed to the screen. If the 'shell' option is "csh" or "tcsh" after initializations, the default becomes "|& tee". If the 'shell' option is "sh", "ksh", "mksh", "pdksh", "zsh" or "bash" the default becomes "2>&1| tee". This means that stderr is also included.
Before using the 'shell' option a path is removed, thus "/bin/sh" uses "sh".
The initialization of this option is done after reading the ".vimrc" and the other initializations, so that when the 'shell' option is set there, the 'shellpipe' option changes automatically, unless it was explicitly set before.
When 'shellpipe' is set to an empty string, no redirection of the ":make" output will be done. This is useful if you use a 'makeprg' that writes to 'makeef' by itself. If you want no piping, but do want to include the 'makeef', set 'shellpipe' to a single space.
Don't forget to precede the space with a backslash: ":set sp=\ ".
In the future pipes may be used for filtering and this option will become obsolete (at least for Unix).
This option cannot be set from a |modeline| or in the |sandbox|, for security reasons.

'shellquote' *'shq'*
'shellquote' 'shq' string (default: ""; MS-DOS and Win32, when 'shell' contains "sh" somewhere: "\"")
global
{not in Vi}
Quoting character(s), put around the command passed to the shell, for the "!" and ":!" commands. The redirection is kept outside of the quoting. See 'shellxquote' to include the redirection. It's probably not useful to set both options.
This is an empty string by default. Only known to be useful for third-party shells on MS-DOS-like systems, such as the MKS Korn Shell or bash, where it should be "\"". The default is adjusted according the value of 'shell', to reduce the need to set this option by the user. See |dos-shell|.
This option cannot be set from a |modeline| or in the |sandbox|, for security reasons.

'shellredir' *'srr'*
'shellredir' 'srr' string (default ">", ">&" or ">%s 2>&1")
global

```
                        {not in Vi}
        String to be used to put the output of a filter command in a temporary
        file.  See also |:!|.  See |option-backslash| about including spaces
        and backslashes.
        The name of the temporary file can be represented by "%s" if necessary
        (the file name is appended automatically if no %s appears in the value
        of this option).
        The default is ">".  For Unix, if the 'shell' option is "csh", "tcsh"
        or "zsh" during initializations, the default becomes ">&".  If the
        'shell' option is "sh", "ksh" or "bash" the default becomes
        ">%s 2>&1".  This means that stderr is also included.
        For Win32, the Unix checks are done and additionally "cmd" is checked
        for, which makes the default ">%s 2>&1".  Also, the same names with
        ".exe" appended are checked for.
        The initialization of this option is done after reading the ".vimrc"
        and the other initializations, so that when the 'shell' option is set
        there, the 'shellredir' option changes automatically unless it was
        explicitly set before.
        In the future pipes may be used for filtering and this option will
        become obsolete (at least for Unix).
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                        *'shellslash'* *'ssl'* *'noshellslash'* *'nossl'*
'shellslash' 'ssl'      boolean (default off)
                        global
                        {not in Vi} {only for MSDOS, MS-Windows and OS/2}
        When set, a forward slash is used when expanding file names.  This is
        useful when a Unix-like shell is used instead of command.com or
        cmd.exe.  Backward slashes can still be typed, but they are changed to
        forward slashes by Vim.
        Note that setting or resetting this option has no effect for some
        existing file names, thus this option needs to be set before opening
        any file for best results.  This might change in the future.
        'shellslash' only works when a backslash can be used as a path
        separator.  To test if this is so use: >
                if exists('+shellslash')
<
                        *'shelltemp'* *'stmp'* *'noshelltemp'* *'nostmp'*
'shelltemp' 'stmp'      boolean (Vi default off, Vim default on)
                        global
                        {not in Vi}
        When on, use temp files for shell commands.  When off use a pipe.
        When using a pipe is not possible temp files are used anyway.
        Currently a pipe is only supported on Unix and MS-Windows 2K and
        later.  You can check it with: >
                :if has("filterpipe")
<       The advantage of using a pipe is that nobody can read the temp file
        and the 'shell' command does not need to support redirection.
        The advantage of using a temp file is that the file type and encoding
        can be detected.
        The |FilterReadPre|, |FilterReadPost| and |FilterWritePre|,
        |FilterWritePost| autocommands event are not triggered when
        'shelltemp' is off.
        The `system()` function does not respect this option and always uses
        temp files.
        NOTE: This option is set to the Vim default value when 'compatible'
        is reset.

                                                *'shelltype'* *'st'*
'shelltype' 'st'        number  (default 0)
                        global
```

```
                        {not in Vi} {only for the Amiga}
        On the Amiga this option influences the way how the commands work
        which use a shell.
        0 and 1: always use the shell
        2 and 3: use the shell only to filter lines
        4 and 5: use shell only for ':sh' command
        When not using the shell, the command is executed directly.

        0 and 2: use "shell 'shellcmdflag' cmd" to start external commands
        1 and 3: use "shell cmd" to start external commands

                                                *'shellxescape'* *'sxe'*
'shellxescape' 'sxe'    string  (default: "";
                                 for MS-DOS and MS-Windows: "\"&|<>()@^")
                        global
                        {not in Vi}
        When 'shellxquote' is set to "(" then the characters listed in this
        option will be escaped with a '^' character.  This makes it possible
        to execute most external commands with cmd.exe.

                                                *'shellxquote'* *'sxq'*
'shellxquote' 'sxq'     string  (default: "";
                                        for Win32, when 'shell' is cmd.exe: "("
                                        for Win32, when 'shell' contains "sh"
                                        somewhere: "\""
                                        for Unix, when using system(): "\"")
                        global
                        {not in Vi}
        Quoting character(s), put around the command passed to the shell, for
        the "!" and ":!" commands.  Includes the redirection.  See
        'shellquote' to exclude the redirection.  It's probably not useful
        to set both options.
        When the value is '(' then ')' is appended. When the value is '"('
        then ')"' is appended.
        When the value is '(' then also see 'shellxescape'.
        This is an empty string by default on most systems, but is known to be
        useful for on Win32 version, either for cmd.exe which automatically
        strips off the first and last quote on a command, or 3rd-party shells
        such as the MKS Korn Shell or bash, where it should be "\"".  The
        default is adjusted according the value of 'shell', to reduce the need
        to set this option by the user.  See |dos-shell|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                        *'shiftround'* *'sr'* *'noshiftround'* *'nosr'*
'shiftround' 'sr'       boolean (default off)
                        global
                        {not in Vi}
        Round indent to multiple of 'shiftwidth'.  Applies to > and <
        commands.  CTRL-T and CTRL-D in Insert mode always round the indent to
        a multiple of 'shiftwidth' (this is Vi compatible).
        NOTE: This option is reset when 'compatible' is set.

                                                *'shiftwidth'* *'sw'*
'shiftwidth' 'sw'       number  (default 8)
                        local to buffer
        Number of spaces to use for each step of (auto)indent.  Used for
        |'cindent'|, |>>|, |<<|, etc.
        When zero the 'ts' value will be used.  Use the |shiftwidth()|
        function to get the effective shiftwidth value.

                                                *'shortmess'* *'shm'*
```

```
'shortmess' 'shm'       string  (Vim default "filnxtToO", Vi default: "",
                                                        POSIX default: "A")
                        global
                        {not in Vi}
        This option helps to avoid all the |hit-enter| prompts caused by file
        messages, for example  with CTRL-G, and to avoid some other messages.
        It is a list of flags:
         flag   meaning when present    ~
          f     use "(3 of 5)" instead of "(file 3 of 5)"
          i     use "[noeol]" instead of "[Incomplete last line]"
          l     use "999L, 888C" instead of "999 lines, 888 characters"
          m     use "[+]" instead of "[Modified]"
          n     use "[New]" instead of "[New File]"
          r     use "[RO]" instead of "[readonly]"
          w     use "[w]" instead of "written" for file write message
                and "[a]" instead of "appended" for ':w >> file' command
          x     use "[dos]" instead of "[dos format]", "[unix]" instead of
                "[unix format]" and "[mac]" instead of "[mac format]".
          a     all of the above abbreviations

          o     overwrite message for writing a file with subsequent message
                for reading a file (useful for ":wn" or when 'autowrite' on)
          O     message for reading a file overwrites any previous message.
                Also for quickfix message (e.g., ":cn").
          s     don't give "search hit BOTTOM, continuing at TOP" or "search
                hit TOP, continuing at BOTTOM" messages
          t     truncate file message at the start if it is too long to fit
                on the command-line, "<" will appear in the left most column.
                Ignored in Ex mode.
          T     truncate other messages in the middle if they are too long to
                fit on the command line.  "..." will appear in the middle.
                Ignored in Ex mode.
          W     don't give "written" or "[w]" when writing a file
          A     don't give the "ATTENTION" message when an existing swap file
                is found.
          I     don't give the intro message when starting Vim |:intro|.
          c     don't give |ins-completion-menu| messages.  For example,
                "-- XXX completion (YYY)", "match 1 of 2", "The only match",
                "Pattern not found", "Back at original", etc.
          q     use "recording" instead of "recording @a"
          F     don't give the file info when editing a file, like `:silent`
                was used for the command

        This gives you the opportunity to avoid that a change between buffers
        requires you to hit <Enter>, but still gives as useful a message as
        possible for the space available.  To get the whole message that you
        would have got with 'shm' empty, use ":file!"
        Useful values:
            shm=        No abbreviation of message.
            shm=a       Abbreviation, but no loss of information.
            shm=at      Abbreviation, and truncate message when necessary.

        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                *'shortname'* *'sn'* *'noshortname'* *'nosn'*
'shortname' 'sn'        boolean (default off)
                        local to buffer
                        {not in Vi, not in MS-DOS versions}
        Filenames are assumed to be 8 characters plus one extension of 3
        characters.  Multiple dots in file names are not allowed.  When this
        option is on, dots in file names are replaced with underscores when
```

```
        adding an extension (".~" or ".swp").  This option is not available
        for MS-DOS, because then it would always be on.  This option is useful
        when editing files on an MS-DOS compatible filesystem, e.g., messydos
        or crossdos.  When running the Win32 GUI version under Win32s, this
        option is always on by default.

                                                *'showbreak'* *'sbr'* *E595*
'showbreak' 'sbr'       string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+linebreak|
                        feature}
        String to put at the start of lines that have been wrapped.  Useful
        values are "> " or "+++ ": >
                :set showbreak=>\
<       Note the backslash to escape the trailing space.  It's easier like
        this: >
                :let &showbreak = '+++ '
<       Only printable single-cell characters are allowed, excluding <Tab> and
        comma (in a future version the comma might be used to separate the
        part that is shown at the end and at the start of a line).
        The characters are highlighted according to the '@' flag in
        'highlight'.
        Note that tabs after the showbreak will be displayed differently.
        If you want the 'showbreak' to appear in between line numbers, add the
        "n" flag to 'cpoptions'.

                                    *'showcmd'* *'sc'* *'noshowcmd'* *'nosc'*
'showcmd' 'sc'          boolean (Vim default: on, off for Unix,
                                     Vi default: off, set in |defaults.vim|)
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+cmdline_info| feature}
        Show (partial) command in the last line of the screen.  Set this
        option off if your terminal is slow.
        In Visual mode the size of the selected area is shown:
        - When selecting characters within a line, the number of characters.
          If the number of bytes is different it is also displayed: "2-6"
          means two characters and six bytes.
        - When selecting more than one line, the number of lines.
        - When selecting a block, the size in screen characters:
          {lines}x{columns}.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                        *'showfulltag'* *'sft'* *'noshowfulltag'* *'nosft'*
'showfulltag' 'sft'     boolean (default off)
                        global
                        {not in Vi}
        When completing a word in insert mode (see |ins-completion|) from the
        tags file, show both the tag name and a tidied-up form of the search
        pattern (if there is one) as possible matches.  Thus, if you have
        matched a C function, you can see a template for what arguments are
        required (coding style permitting).
        Note that this doesn't work well together with having "longest" in
        'completeopt', because the completion from the search pattern may not
        match the typed text.

                                    *'showmatch'* *'sm'* *'noshowmatch'* *'nosm'*
'showmatch' 'sm'        boolean (default off)
                        global
```

When a bracket is inserted, briefly jump to the matching one. The
jump is only done if the match can be seen on the screen. The time to
show the match can be set with 'matchtime'.
A Beep is given if there is no match (no matter if the match can be
seen or not).
This option is reset when 'paste' is set and restored when 'paste' is
reset.
When the 'm' flag is not included in 'cpoptions', typing a character
will immediately move the cursor back to where it belongs.
See the "sm" field in 'guicursor' for setting the cursor shape and
blinking when showing the match.
The 'matchpairs' option can be used to specify the characters to show
matches for. 'rightleft' and 'revins' are used to look for opposite
matches.
Also see the matchparen plugin for highlighting the match when moving
around |pi_paren.txt|.
Note: Use of the short form is rated PG.

'showmode' *'smd'* *'noshowmode'* *'nosmd'*
'showmode' 'smd' boolean (Vim default: on, Vi default: off)
global
If in Insert, Replace or Visual mode put a message on the last line.
Use the 'M' flag in 'highlight' to set the type of highlighting for
this message.
When |XIM| may be used the message will include "XIM". But this
doesn't mean XIM is really active, especially when 'imactivatekey' is
not set.
NOTE: This option is set to the Vi default value when 'compatible' is
set and to the Vim default value when 'compatible' is reset.

'showtabline' *'stal'*
'showtabline' 'stal' number (default 1)
global
{not in Vi}
{not available when compiled without the |+windows|
feature}
The value of this option specifies when the line with tab page labels
will be displayed:
0: never
1: only if there are at least two tab pages
2: always
This is both for the GUI and non-GUI implementation of the tab pages
line.
See |tab-page| for more information about tab pages.

'sidescroll' *'ss'*
'sidescroll' 'ss' number (default 0)
global
{not in Vi}
The minimal number of columns to scroll horizontally. Used only when
the 'wrap' option is off and the cursor is moved off of the screen.
When it is zero the cursor will be put in the middle of the screen.
When using a slow terminal set it to a large number or 0. When using
a fast terminal use a small number or 1. Not used for "zh" and "zl"
commands.

'sidescrolloff' *'siso'*
'sidescrolloff' 'siso' number (default 0)
global
{not in Vi}
The minimal number of screen columns to keep to the left and to the
right of the cursor if 'nowrap' is set. Setting this option to a

```
        value greater than 0 while having |'sidescroll'| also at a non-zero
        value makes some context visible in the line you are scrolling in
        horizontally (except at beginning of the line).  Setting this option
        to a large value (like 999) has the effect of keeping the cursor
        horizontally centered in the window, as long as one does not come too
        close to the beginning of the line.
        NOTE: This option is set to 0 when 'compatible' is set.

        Example: Try this together with 'sidescroll' and 'listchars' as
                 in the following example to never allow the cursor to move
                 onto the "extends" character: >

                 :set nowrap sidescroll=1 listchars=extends:>,precedes:<
                 :set sidescrolloff=1
<
                                                  *'signcolumn'* *'scl'*
'signcolumn' 'scl'      string  (default "auto")
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+signs|
                        feature}
        Whether or not to draw the signcolumn. Valid values are:
           "auto"       only when there is a sign to display
           "no"         never
           "yes"        always


                        *'smartcase'* *'scs'* *'nosmartcase'* *'noscs'*
'smartcase' 'scs'       boolean (default off)
                        global
                        {not in Vi}
        Override the 'ignorecase' option if the search pattern contains upper
        case characters.  Only used when the search pattern is typed and
        'ignorecase' option is on.  Used for the commands "/", "?", "n", "N",
        ":g" and ":s".  Not used for "*", "#", "gd", tag search, etc.  After
        "*" and "#" you can make 'smartcase' used by doing a "/" command,
        recalling the search pattern from history and hitting <Enter>.
        NOTE: This option is reset when 'compatible' is set.

                             *'smartindent'* *'si'* *'nosmartindent'* *'nosi'*
'smartindent' 'si'      boolean (default off)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the
                        |+smartindent| feature}
        Do smart autoindenting when starting a new line.  Works for C-like
        programs, but can also be used for other languages.  'cindent' does
        something like this, works better in most cases, but is more strict,
        see |C-indenting|.  When 'cindent' is on or 'indentexpr' is set,
        setting 'si' has no effect.  'indentexpr' is a more advanced
        alternative.
        Normally 'autoindent' should also be on when using 'smartindent'.
        An indent is automatically inserted:
        - After a line ending in '{'.
        - After a line starting with a keyword from 'cinwords'.
        - Before a line starting with '}' (only with the "O" command).
        When typing '}' as the first character in a new line, that line is
        given the same indent as the matching '{'.
        When typing '#' as the first character in a new line, the indent for
        that line is removed, the '#' is put in the first column.  The indent
        is restored for the next line.  If you don't want this, use this
        mapping: ":inoremap # X^H#", where ^H is entered with CTRL-V CTRL-H.
```

When using the ">>" command, lines starting with '#' are not shifted right.
NOTE: This option is reset when 'compatible' is set.
This option is reset when 'paste' is set and restored when 'paste' is reset.

'smarttab' *'sta'* *'nosmarttab'* *'nosta'*
'smarttab' 'sta' boolean (default off)
global
{not in Vi}
When on, a <Tab> in front of a line inserts blanks according to 'shiftwidth'. 'tabstop' or 'softtabstop' is used in other places. A <BS> will delete a 'shiftwidth' worth of space at the start of the line.
When off, a <Tab> always inserts blanks according to 'tabstop' or 'softtabstop'. 'shiftwidth' is only used for shifting text left or right |shift-left-right|.
What gets inserted (a <Tab> or spaces) depends on the 'expandtab' option. Also see |ins-expandtab|. When 'expandtab' is not set, the number of spaces is minimized by using <Tab>s.
This option is reset when 'paste' is set and restored when 'paste' is reset.
NOTE: This option is reset when 'compatible' is set.

'softtabstop' *'sts'*
'softtabstop' 'sts' number (default 0)
local to buffer
{not in Vi}
Number of spaces that a <Tab> counts for while performing editing operations, like inserting a <Tab> or using <BS>. It "feels" like <Tab>s are being inserted, while in fact a mix of spaces and <Tab>s is used. This is useful to keep the 'ts' setting at its standard value of 8, while being able to edit like it is set to 'sts'. However, commands like "x" still work on the actual characters.
When 'sts' is zero, this feature is off.
When 'sts' is negative, the value of 'shiftwidth' is used.
'softtabstop' is set to 0 when the 'paste' option is set and restored when 'paste' is reset.
See also |ins-expandtab|. When 'expandtab' is not set, the number of spaces is minimized by using <Tab>s.
The 'L' flag in 'cpoptions' changes how tabs are used when 'list' is set.
NOTE: This option is set to 0 when 'compatible' is set.

'spell' *'nospell'*
'spell' boolean (default off)
local to window
{not in Vi}
{not available when compiled without the |+syntax| feature}
When on spell checking will be done. See |spell|.
The languages are specified with 'spelllang'.

'spellcapcheck' *'spc'*
'spellcapcheck' 'spc' string (default "[.?!]_[\])'" \t]\+")
local to buffer
{not in Vi}
{not available when compiled without the |+syntax| feature}
Pattern to locate the end of a sentence. The following word will be checked to start with a capital letter. If not then it is highlighted with SpellCap |hl-SpellCap| (unless the word is also badly spelled).

When this check is not wanted make this option empty.
Only used when 'spell' is set.
Be careful with special characters, see |option-backslash| about
including spaces and backslashes.
To set this option automatically depending on the language, see
|set-spc-auto|.

'spellfile' *'spf'*
'spellfile' 'spf' string (default empty)
local to buffer
{not in Vi}
{not available when compiled without the |+syntax|
feature}
Name of the word list file where words are added for the |zg| and |zw|
commands. It must end in ".{encoding}.add". You need to include the
path, otherwise the file is placed in the current directory.
E765
It may also be a comma separated list of names. A count before the
|zg| and |zw| commands can be used to access each. This allows using
a personal word list file and a project word list file.
When a word is added while this option is empty Vim will set it for
you: Using the first directory in 'runtimepath' that is writable. If
there is no "spell" directory yet it will be created. For the file
name the first language name that appears in 'spelllang' is used,
ignoring the region.
The resulting ".spl" file will be used for spell checking, it does not
have to appear in 'spelllang'.
Normally one file is used for all regions, but you can add the region
name if you want to. However, it will then only be used when
'spellfile' is set to it, for entries in 'spelllang' only files
without region name will be found.
This option cannot be set from a |modeline| or in the |sandbox|, for
security reasons.

'spelllang' *'spl'*
'spelllang' 'spl' string (default "en")
local to buffer
{not in Vi}
{not available when compiled without the |+syntax|
feature}
A comma separated list of word list names. When the 'spell' option is
on spellchecking will be done for these languages. Example: >
set spelllang=en_us,nl,medical
< This means US English, Dutch and medical words are recognized. Words
that are not recognized will be highlighted.
The word list name must not include a comma or dot. Using a dash is
recommended to separate the two letter language name from a
specification. Thus "en-rare" is used for rare English words.
A region name must come last and have the form "_xx", where "xx" is
the two-letter, lower case region name. You can use more than one
region by listing them: "en_us,en_ca" supports both US and Canadian
English, but not words specific for Australia, New Zealand or Great
Britain. (Note: currently en_au and en_nz dictionaries are older than
en_ca, en_gb and en_us).
If the name "cjk" is included East Asian characters are excluded from
spell checking. This is useful when editing text that also has Asian
words.
E757
As a special case the name of a .spl file can be given as-is. The
first "_xx" in the name is removed and used as the region name
(_xx is an underscore, two letters and followed by a non-letter).
This is mainly for testing purposes. You must make sure the correct

```
        encoding is used, Vim doesn't check it.
        When 'encoding' is set the word lists are reloaded.  Thus it's a good
        idea to set 'spelllang' after setting 'encoding' to avoid loading the
        files twice.
        How the related spell files are found is explained here: |spell-load|.

        If the |spellfile.vim| plugin is active and you use a language name
        for which Vim cannot find the .spl file in 'runtimepath' the plugin
        will ask you if you want to download the file.

        After this option has been set successfully, Vim will source the files
        "spell/LANG.vim" in 'runtimepath'.  "LANG" is the value of 'spelllang'
        up to the first comma, dot or underscore.
        Also see |set-spc-auto|.


                                                *'spellsuggest'* *'sps'*
'spellsuggest' 'sps'    string  (default "best")
                        global
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        Methods used for spelling suggestions.  Both for the |z=| command and
        the |spellsuggest()| function.  This is a comma-separated list of
        items:

        best            Internal method that works best for English.  Finds
                        changes like "fast" and uses a bit of sound-a-like
                        scoring to improve the ordering.

        double          Internal method that uses two methods and mixes the
                        results.  The first method is "fast", the other method
                        computes how much the suggestion sounds like the bad
                        word.  That only works when the language specifies
                        sound folding.  Can be slow and doesn't always give
                        better results.

        fast            Internal method that only checks for simple changes:
                        character inserts/deletes/swaps.  Works well for
                        simple typing mistakes.

        {number}        The maximum number of suggestions listed for |z=|.
                        Not used for |spellsuggest()|.  The number of
                        suggestions is never more than the value of 'lines'
                        minus two.

        file:{filename} Read file {filename}, which must have two columns,
                        separated by a slash.  The first column contains the
                        bad word, the second column the suggested good word.
                        Example:
                                theribal/terrible ~
                        Use this for common mistakes that do not appear at the
                        top of the suggestion list with the internal methods.
                        Lines without a slash are ignored, use this for
                        comments.
                        The word in the second column must be correct,
                        otherwise it will not be used.  Add the word to an
                        ".add" file if it is currently flagged as a spelling
                        mistake.
                        The file is used for all languages.

        expr:{expr}     Evaluate expression {expr}.  Use a function to avoid
```

```
                        trouble with spaces.  |v:val| holds the badly spelled
                        word.  The expression must evaluate to a List of
                        Lists, each with a suggestion and a score.
                        Example:
                                [['the', 33], ['that', 44]] ~
                        Set 'verbose' and use |z=| to see the scores that the
                        internal methods use.  A lower score is better.
                        This may invoke |spellsuggest()| if you temporarily
                        set 'spellsuggest' to exclude the "expr:" part.
                        Errors are silently ignored, unless you set the
                        'verbose' option to a non-zero value.

        Only one of "best", "double" or "fast" may be used.  The others may
        appear several times in any order.  Example: >
                :set sps=file:~/.vim/sugg,best,expr:MySuggest()
<
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.


                        *'splitbelow'* *'sb'* *'nosplitbelow'* *'nosb'*
'splitbelow' 'sb'       boolean (default off)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        When on, splitting a window will put the new window below the current
        one. |:split|

                        *'splitright'* *'spr'* *'nosplitright'* *'nospr'*
'splitright' 'spr'      boolean (default off)
                        global
                        {not in Vi}
                        {not available when compiled without the |+vertsplit|
                        feature}
        When on, splitting a window will put the new window right of the
        current one. |:vsplit|

                           *'startofline'* *'sol'* *'nostartofline'* *'nosol'*
'startofline' 'sol'     boolean (default on)
                        global
                        {not in Vi}
        When "on" the commands listed below move the cursor to the first
        non-blank of the line.  When off the cursor is kept in the same column
        (if possible).  This applies to the commands: CTRL-D, CTRL-U, CTRL-B,
        CTRL-F, "G", "H", "M", "L", gg, and to the commands "d", "<<" and ">>"
        with a linewise operator, with "%" with a count and to buffer changing
        commands (CTRL-^, :bnext, :bNext, etc.).  Also for an Ex command that
        only has a line number, e.g., ":25" or ":+".
        In case of buffer changing commands the cursor is placed at the column
        where it was the last time the buffer was edited.
        NOTE: This option is set when 'compatible' is set.

                           *'statusline'* *'stl'* *E540* *E542*
'statusline' 'stl'      string  (default empty)
                        global or local to window |global-local|
                        {not in Vi}
                        {not available when compiled without the |+statusline|
                        feature}
        When nonempty, this option determines the content of the status line.
        Also see |status-line|.
```

```
        The option consists of printf style '%' items interspersed with
        normal text.  Each status line item is of the form:
          %-0{minwid}.{maxwid}{item}
        All fields except the {item} are optional.  A single percent sign can
        be given as "%%".  Up to 80 items can be specified.  *E541*

        When the option starts with "%!" then it is used as an expression,
        evaluated and the result is used as the option value.  Example: >
                :set statusline=%!MyStatusLine()
<       The result can contain %{} items that will be evaluated too.
        Note that the "%!" expression is evaluated in the context of the
        current window and buffer, while %{} items are evaluated in the
        context of the window that the statusline belongs to.

        When there is error while evaluating the option then it will be made
        empty to avoid further errors.  Otherwise screen updating would loop.

        Note that the only effect of 'ruler' when this option is set (and
        'laststatus' is 2) is controlling the output of |CTRL-G|.

        field       meaning ~
        -           Left justify the item.  The default is right justified
                    when minwid is larger than the length of the item.
        0           Leading zeroes in numeric items.  Overridden by '-'.
        minwid      Minimum width of the item, padding as set by '-' & '0'.
                    Value must be 50 or less.
        maxwid      Maximum width of the item.  Truncation occurs with a '<'
                    on the left for text items.  Numeric items will be
                    shifted down to maxwid-2 digits followed by '>'number
                    where number is the amount of missing digits, much like
                    an exponential notation.
        item        A one letter code as described below.

        Following is a description of the possible statusline items.  The
        second character in "item" is the type:
                N for number
                S for string
                F for flags as described below
                - not applicable

        item  meaning ~
        f S   Path to the file in the buffer, as typed or relative to current
              directory.
        F S   Full path to the file in the buffer.
        t S   File name (tail) of file in the buffer.
        m F   Modified flag, text is "[+]"; "[-]" if 'modifiable' is off.
        M F   Modified flag, text is ",+" or ",-".
        r F   Readonly flag, text is "[RO]".
        R F   Readonly flag, text is ",RO".
        h F   Help buffer flag, text is "[help]".
        H F   Help buffer flag, text is ",HLP".
        w F   Preview window flag, text is "[Preview]".
        W F   Preview window flag, text is ",PRV".
        y F   Type of file in the buffer, e.g., "[vim]".  See 'filetype'.
        Y F   Type of file in the buffer, e.g., ",VIM".  See 'filetype'.
              {not available when compiled without |+autocmd| feature}
        q S   "[Quickfix List]", "[Location List]" or empty.
        k S   Value of "b:keymap_name" or 'keymap' when |:lmap| mappings are
              being used: "<keymap>"
        n N   Buffer number.
        b N   Value of character under cursor.
        B N   As above, in hexadecimal.
```

```
        o N   Byte number in file of byte under cursor, first byte is 1.
              Mnemonic: Offset from start of file (with one added)
              {not available when compiled without |+byte_offset| feature}
        O N   As above, in hexadecimal.
        N N   Printer page number.  (Only works in the 'printheader' option.)
        l N   Line number.
        L N   Number of lines in buffer.
        c N   Column number.
        v N   Virtual column number.
        V N   Virtual column number as -{num}.  Not displayed if equal to 'c'.
        p N   Percentage through file in lines as in |CTRL-G|.
        P S   Percentage through file of displayed window.  This is like the
              percentage described for 'ruler'.  Always 3 in length, unless
              translated.
        a S   Argument list status as in default title.  ({current} of {max})
              Empty if the argument file count is zero or one.
        { NF  Evaluate expression between '%{' and '}' and substitute result.
              Note that there is no '%' before the closing '}'.
        ( -   Start of item group.  Can be used for setting the width and
              alignment of a section.  Must be followed by %) somewhere.
        ) -   End of item group.  No width fields allowed.
        T N   For 'tabline': start of tab page N label.  Use %T after the last
              label.  This information is used for mouse clicks.
        X N   For 'tabline': start of close tab N label.  Use %X after the
              label, e.g.: %3Xclose%X.  Use %999X for a "close current tab"
              mark.  This information is used for mouse clicks.
        < -   Where to truncate line if too long.  Default is at the start.
              No width fields allowed.
        = -   Separation point between left and right aligned items.
              No width fields allowed.
        # -   Set highlight group.  The name must follow and then a # again.
              Thus use %#HLname# for highlight group HLname.  The same
              highlighting is used, also for the statusline of non-current
              windows.
        * -   Set highlight group to User{N}, where {N} is taken from the
              minwid field, e.g. %1*.  Restore normal highlight with %* or %0*.
              The difference between User{N} and StatusLine  will be applied
              to StatusLineNC for the statusline of non-current windows.
              The number N must be between 1 and 9.  See |hl-User1..9|

        When displaying a flag, Vim removes the leading comma, if any, when
        that flag comes right after plaintext.  This will make a nice display
        when flags are used like in the examples below.

        When all items in a group becomes an empty string (i.e. flags that are
        not set) and a minwid is not set for the group, the whole group will
        become empty.  This will make a group like the following disappear
        completely from the statusline when none of the flags are set. >
                :set statusline=...%(\ [%M%R%H]%)...
<                                                       *g:actual_curbuf*
        Beware that an expression is evaluated each and every time the status
        line is displayed.  The current buffer and current window will be set
        temporarily to that of the window (and buffer) whose statusline is
        currently being drawn.  The expression will evaluate in this context.
        The variable "actual_curbuf" is set to the 'bufnr()' number of the
        real current buffer.

        The 'statusline' option will be evaluated in the |sandbox| if set from
        a modeline, see |sandbox-option|.

        It is not allowed to change text or jump to another window while
        evaluating 'statusline' |textlock|.
```

```
        If the statusline is not updated when you want it (e.g., after setting
        a variable that's used in an expression), you can force an update by
        setting an option without changing its value.  Example: >
                :let &ro = &ro

<       A result of all digits is regarded a number for display purposes.
        Otherwise the result is taken as flag text and applied to the rules
        described above.

        Watch out for errors in expressions.  They may render Vim unusable!
        If you are stuck, hold down ':' or 'Q' to get a prompt, then quit and
        edit your .vimrc or whatever with "vim --clean" to get it right.

        Examples:
        Emulate standard status line with 'ruler' set >
          :set statusline=%<%f\ %h%m%r%=%-14.(%l,%c%V%)\ %P
<       Similar, but add ASCII value of char under the cursor (like "ga") >
          :set statusline=%<%f%h%m%r%=%b\ 0x%B\ \ %l,%c%V\ %P
<       Display byte count and byte value, modified flag in red. >
          :set statusline=%<%f%=\ [%1*%M%*%n%R%H]\ %-19(%3l,%02c%03V%)%O'%02b'
          :hi User1 term=inverse,bold cterm=inverse,bold ctermfg=red
<       Display a ,GZ flag if a compressed file is loaded >
          :set statusline=...%r%{VarExists('b:gzflag','\ [GZ]')}%h...
<       In the |:autocmd|'s: >
          :let b:gzflag = 1
<       And: >
          :unlet b:gzflag
<       And define this function: >
          :function VarExists(var, val)
          :    if exists(a:var) | return a:val | else | return '' | endif
          :endfunction
<
                                                *'suffixes'* *'su'*
'suffixes' 'su'         string  (default ".bak,~,.o,.h,.info,.swp,.obj")
                        global
                        {not in Vi}
        Files with these suffixes get a lower priority when multiple files
        match a wildcard.  See |suffixes|.  Commas can be used to separate the
        suffixes.  Spaces after the comma are ignored.  A dot is also seen as
        the start of a suffix.  To avoid a dot or comma being recognized as a
        separator, precede it with a backslash (see |option-backslash| about
        including spaces and backslashes).
        See 'wildignore' for completely ignoring files.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        suffixes from the list.  This avoids problems when a future version
        uses another default.

                                                *'suffixesadd'* *'sua'*
'suffixesadd' 'sua'     string  (default "")
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the
                        |+file_in_path| feature}
        Comma separated list of suffixes, which are used when searching for a
        file for the "gf", "[I", etc. commands.  Example: >
                :set suffixesadd=.java
<
                                *'swapfile'* *'swf'* *'noswapfile'* *'noswf'*
'swapfile' 'swf'        boolean (default on)
                        local to buffer
                        {not in Vi}
```

```
        Use a swapfile for the buffer.  This option can be reset when a
        swapfile is not wanted for a specific buffer.  For example, with
        confidential information that even root must not be able to access.
        Careful: All text will be in memory:
                - Don't use this for big files.
                - Recovery will be impossible!
        A swapfile will only be present when |'updatecount'| is non-zero and
        'swapfile' is set.
        When 'swapfile' is reset, the swap file for the current buffer is
        immediately deleted.  When 'swapfile' is set, and 'updatecount' is
        non-zero, a swap file is immediately created.
        Also see |swap-file| and |'swapsync'|.
        If you want to open a new buffer without creating a swap file for it,
        use the |:noswapfile| modifier.

        This option is used together with 'bufhidden' and 'buftype' to
        specify special kinds of buffers.   See |special-buffers|.

                                                *'swapsync'* *'sws'*
'swapsync' 'sws'        string  (default "fsync")
                        global
                        {not in Vi}
        When this option is not empty a swap file is synced to disk after
        writing to it.  This takes some time, especially on busy unix systems.
        When this option is empty parts of the swap file may be in memory and
        not written to disk.  When the system crashes you may lose more work.
        On Unix the system does a sync now and then without Vim asking for it,
        so the disadvantage of setting this option off is small.  On some
        systems the swap file will not be written at all.  For a unix system
        setting it to "sync" will use the sync() call instead of the default
        fsync(), which may work better on some systems.
        The 'fsync' option is used for the actual file.

                                                *'switchbuf'* *'swb'*
'switchbuf' 'swb'       string  (default "")
                        global
                        {not in Vi}
        This option controls the behavior when switching between buffers.
        Possible values (comma separated list):
           useopen      If included, jump to the first open window that
                        contains the specified buffer (if there is one).
                        Otherwise: Do not examine other windows.
                        This setting is checked with |quickfix| commands, when
                        jumping to errors (":cc", ":cn", "cp", etc.).  It is
                        also used in all buffer related split commands, for
                        example ":sbuffer", ":sbnext", or ":sbrewind".
           usetab       Like "useopen", but also consider windows in other tab
                        pages.
           split        If included, split the current window before loading
                        a buffer for a |quickfix| command that display errors.
                        Otherwise: do not split, use current window.
           vsplit       Just like "split" but split vertically.
           newtab       Like "split", but open a new tab page.  Overrules
                        "split" when both are present.

                                                *'synmaxcol'* *'smc'*
'synmaxcol' 'smc'       number  (default 3000)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        Maximum column in which to search for syntax items.  In long lines the
```

```
        text after this column is not highlighted and following lines may not
        be highlighted correctly, because the syntax state is cleared.
        This helps to avoid very slow redrawing for an XML file that is one
        long line.
        Set to zero to remove the limit.

                                                *'syntax'* *'syn'*
'syntax' 'syn'          string  (default empty)
                        local to buffer
                        {not in Vi}
                        {not available when compiled without the |+syntax|
                        feature}
        When this option is set, the syntax with this name is loaded, unless
        syntax highlighting has been switched off with ":syntax off".
        Otherwise this option does not always reflect the current syntax (the
        b:current_syntax variable does).
        This option is most useful in a modeline, for a file which syntax is
        not automatically recognized.  Example, in an IDL file:
                /* vim: set syntax=idl : */ ~
        When a dot appears in the value then this separates two filetype
        names.  Example:
                /* vim: set syntax=c.doxygen : */ ~
        This will use the "c" syntax first, then the "doxygen" syntax.
        Note that the second one must be prepared to be loaded as an addition,
        otherwise it will be skipped.  More than one dot may appear.
        To switch off syntax highlighting for the current file, use: >
                :set syntax=OFF
<       To switch syntax highlighting on according to the current value of the
        'filetype' option: >
                :set syntax=ON
<       What actually happens when setting the 'syntax' option is that the
        Syntax autocommand event is triggered with the value as argument.
        This option is not copied to another buffer, independent of the 's' or
        'S' flag in 'cpoptions'.
        Only normal file name characters can be used, "/\*?[|<>" are illegal.

                                                *'tabline'* *'tal'*
'tabline' 'tal'         string  (default empty)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        When nonempty, this option determines the content of the tab pages
        line at the top of the Vim window.  When empty Vim will use a default
        tab pages line.  See |setting-tabline| for more info.

        The tab pages line only appears as specified with the 'showtabline'
        option and only when there is no GUI tab line.  When 'e' is in
        'guioptions' and the GUI supports a tab line 'guitablabel' is used
        instead.  Note that the two tab pages lines are very different.

        The value is evaluated like with 'statusline'.  You can use
        |tabpagenr()|, |tabpagewinnr()| and |tabpagebuflist()| to figure out
        the text to be displayed.  Use "%1T" for the first label, "%2T" for
        the second one, etc.  Use "%X" items for closing labels.

        Keep in mind that only one of the tab pages is the current one, others
        are invisible and you can't jump to their windows.


                                                *'tabpagemax'* *'tpm'*
'tabpagemax' 'tpm'      number  (default 10)
```

```
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        Maximum number of tab pages to be opened by the |-p| command line
        argument or the ":tab all" command. |tabpage|


                                                *'tabstop'* *'ts'*
'tabstop' 'ts'          number  (default 8)
                        local to buffer
        Number of spaces that a <Tab> in the file counts for.  Also see
        |:retab| command, and 'softtabstop' option.

        Note: Setting 'tabstop' to any other value than 8 can make your file
        appear wrong in many places (e.g., when printing it).

        There are four main ways to use tabs in Vim:
        1. Always keep 'tabstop' at 8, set 'softtabstop' and 'shiftwidth' to 4
           (or 3 or whatever you prefer) and use 'noexpandtab'.  Then Vim
           will use a mix of tabs and spaces, but typing <Tab> and <BS> will
           behave like a tab appears every 4 (or 3) characters.
        2. Set 'tabstop' and 'shiftwidth' to whatever you prefer and use
           'expandtab'.  This way you will always insert spaces.  The
           formatting will never be messed up when 'tabstop' is changed.
        3. Set 'tabstop' and 'shiftwidth' to whatever you prefer and use a
           |modeline| to set these values when editing the file again.  Only
           works when using Vim to edit the file.
        4. Always set 'tabstop' and 'shiftwidth' to the same value, and
           'noexpandtab'.  This should then work (for initial indents only)
           for any tabstop setting that people use.  It might be nice to have
           tabs after the first non-blank inserted as spaces if you do this
           though.  Otherwise aligned comments will be wrong when 'tabstop' is
           changed.

                        *'tagbsearch'* *'tbs'* *'notagbsearch'* *'notbs'*
'tagbsearch' 'tbs'      boolean (default on)
                        global
                        {not in Vi}
        When searching for a tag (e.g., for the |:ta| command), Vim can either
        use a binary search or a linear search in a tags file.  Binary
        searching makes searching for a tag a LOT faster, but a linear search
        will find more tags if the tags file wasn't properly sorted.
        Vim normally assumes that your tags files are sorted, or indicate that
        they are not sorted.  Only when this is not the case does the
        'tagbsearch' option need to be switched off.

        When 'tagbsearch' is on, binary searching is first used in the tags
        files.  In certain situations, Vim will do a linear search instead for
        certain files, or retry all files with a linear search.  When
        'tagbsearch' is off, only a linear search is done.

        Linear searching is done anyway, for one file, when Vim finds a line
        at the start of the file indicating that it's not sorted: >
   !_TAG_FILE_SORTED    0       /some comment/
<       [The whitespace before and after the '0' must be a single <Tab>]

        When a binary search was done and no match was found in any of the
        files listed in 'tags', and case is ignored or a pattern is used
        instead of a normal tag name, a retry is done with a linear search.
        Tags in unsorted tags files, and matches with different case will only
        be found in the retry.
```

```
        If a tag file indicates that it is case-fold sorted, the second,
        linear search can be avoided when case is ignored.  Use a value of '2'
        in the "!_TAG_FILE_SORTED" line for this.  A tag file can be case-fold
        sorted with the -f switch to "sort" in most unices, as in the command:
        "sort -f -o tags tags".  For "Exuberant ctags" version 5.x or higher
        (at least 5.5) the --sort=foldcase switch can be used for this as
        well.  Note that case must be folded to uppercase for this to work.

        By default, tag searches are case-sensitive.  Case is ignored when
        'ignorecase' is set and 'tagcase' is "followic", or when 'tagcase' is
        "ignore".
        Also when 'tagcase' is "followscs" and 'smartcase' is set, or
        'tagcase' is "smart", and the pattern contains only lowercase
        characters.

        When 'tagbsearch' is off, tags searching is slower when a full match
        exists, but faster when no full match exists.  Tags in unsorted tags
        files may only be found with 'tagbsearch' off.
        When the tags file is not sorted, or sorted in a wrong way (not on
        ASCII byte value), 'tagbsearch' should be off, or the line given above
        must be included in the tags file.
        This option doesn't affect commands that find all matching tags (e.g.,
        command-line completion and ":help").
        {Vi: always uses binary search in some versions}

                                                          *'tagcase'* *'tc'*
'tagcase' 'tc'          string  (default "followic")
                        global or local to buffer |global-local|
                        {not in Vi}
        This option specifies how case is handled when searching the tags
        file:
           followic     Follow the 'ignorecase' option
           followscs    Follow the 'smartcase' and 'ignorecase' options
           ignore       Ignore case
           match        Match case
           smart        Ignore case unless an upper case letter is used
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                  *'taglength'* *'tl'*
'taglength' 'tl'        number  (default 0)
                        global
        If non-zero, tags are significant up to this number of characters.

                        *'tagrelative'* *'tr'* *'notagrelative'* *'notr'*
'tagrelative' 'tr'      boolean (Vim default: on, Vi default: off)
                        global
                        {not in Vi}
        If on and using a tags file in another directory, file names in that
        tags file are relative to the directory where the tags file is.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                  *'tags'* *'tag'* *E433*
'tags' 'tag'            string  (default "./tags,tags", when compiled with
                                |+emacs_tags|: "./tags,./TAGS,tags,TAGS")
                        global or local to buffer |global-local|
        Filenames for the tag command, separated by spaces or commas.  To
        include a space or comma in a file name, precede it with a backslash
        (see |option-backslash| about including spaces and backslashes).
        When a file name starts with "./", the '.' is replaced with the path
```

```
        of the current file.  But only when the 'd' flag is not included in
        'cpoptions'.  Environment variables are expanded |:set_env|.  Also see
        |tags-option|.
        "*", "**" and other wildcards can be used to search for tags files in
        a directory tree.  See |file-searching|.  E.g., "/lib/**/tags" will
        find all files named "tags" below "/lib".  The filename itself cannot
        contain wildcards, it is used as-is.  E.g., "/lib/**/tags?" will find
        files called "tags?".  {not available when compiled without the
        |+path_extra| feature}
        The |tagfiles()| function can be used to get a list of the file names
        actually used.
        If Vim was compiled with the |+emacs_tags| feature, Emacs-style tag
        files are also supported.  They are automatically recognized.  The
        default value becomes "./tags,./TAGS,tags,TAGS", unless case
        differences are ignored (MS-Windows).  |emacs-tags|
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        file names from the list.  This avoids problems when a future version
        uses another default.
        {Vi: default is "tags /usr/lib/tags"}

                                *'tagstack'* *'tgst'* *'notagstack'* *'notgst'*
'tagstack' 'tgst'       boolean (default on)
                        global
                        {not in all versions of Vi}
        When on, the |tagstack| is used normally.  When off, a ":tag" or
        ":tselect" command with an argument will not push the tag onto the
        tagstack.  A following ":tag" without an argument, a ":pop" command or
        any other command that uses the tagstack will use the unmodified
        tagstack, but does change the pointer to the active entry.
        Resetting this option is useful when using a ":tag" command in a
        mapping which should not change the tagstack.

                                                *'tcldll'*
'tcldll'                string  (default depends on the build)
                        global
                        {not in Vi}
                        {only available when compiled with the |+tcl/dyn|
                        feature}
        Specifies the name of the Tcl shared library. The default is
        DYNAMIC_TCL_DLL, which was specified at compile time.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'term'* *E529* *E530* *E531*
'term'                  string  (default is $TERM, if that fails:
                                     in the GUI: "builtin_gui"
                                       on Amiga: "amiga"
                                        on BeOS: "beos-ansi"
                                         on Mac: "mac-ansi"
                                       on MiNT: "vt52"
                                     on MS-DOS: "pcterm"
                                       on OS/2: "os2ansi"
                                       on Unix: "ansi"
                                        on VMS: "ansi"
                                     on Win 32: "win32")
                        global
        Name of the terminal.  Used for choosing the terminal control
        characters.  Environment variables are expanded |:set_env|.
        For example: >
                :set term=$TERM
<       See |termcap|.
```

```
                                                *'termbidi'* *'tbidi'*
                                                *'notermbidi'* *'notbidi'*
'termbidi' 'tbidi'      boolean (default off, on for "mlterm")
                        global
                        {not in Vi}
                        {only available when compiled with the |+arabic|
                        feature}
        The terminal is in charge of Bi-directionality of text (as specified
        by Unicode).  The terminal is also expected to do the required shaping
        that some languages (such as Arabic) require.
        Setting this option implies that 'rightleft' will not be set when
        'arabic' is set and the value of 'arabicshape' will be ignored.
        Note that setting 'termbidi' has the immediate effect that
        'arabicshape' is ignored, but 'rightleft' isn't changed automatically.
        This option is reset when the GUI is started.
        For further details see |arabic.txt|.

                                          *'termencoding'* *'tenc'*
'termencoding' 'tenc'   string  (default ""; with GTK+ GUI: "utf-8"; with
                                                  Macintosh GUI: "macroman")
                        global
                        {only available when compiled with the |+multi_byte|
                        feature}
                        {not in Vi}
        Encoding used for the terminal.  This specifies what character
        encoding the keyboard produces and the display will understand.  For
        the GUI it only applies to the keyboard ('encoding' is used for the
        display).  Except for the Mac when 'macatsui' is off, then
        'termencoding' should be "macroman".
                                                                  *E617*
        Note: This does not apply to the GTK+ GUI.  After the GUI has been
        successfully initialized, 'termencoding' is forcibly set to "utf-8".
        Any attempts to set a different value will be rejected, and an error
        message is shown.
        For the Win32 GUI and console versions 'termencoding' is not used,
        because the Win32 system always passes Unicode characters.
        When empty, the same encoding is used as for the 'encoding' option.
        This is the normal value.
        Not all combinations for 'termencoding' and 'encoding' are valid.  See
        |encoding-table|.
        The value for this option must be supported by internal conversions or
        iconv().  When this is not possible no conversion will be done and you
        will probably experience problems with non-ASCII characters.
        Example: You are working with the locale set to euc-jp (Japanese) and
        want to edit a UTF-8 file: >
                :let &termencoding = &encoding
                :set encoding=utf-8
<       You need to do this when your system has no locale support for UTF-8.

                                                *'termguicolors'* *'tgc'*
'termguicolors' 'tgc'   boolean (default off)
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+termguicolors| feature}
        When on, uses |highlight-guifg| and |highlight-guibg| attributes in
        the terminal (thus using 24-bit color). Requires a ISO-8613-3
        compatible terminal.
        If setting this option does not work (produces a colorless UI)
        reading |xterm-true-color| might help.
        Note that the "cterm" attributes are still used, not the "gui" ones.
```

```
        NOTE: This option is reset when 'compatible' is set.

                                                *'termkey'* *'tk'*
'termkey' 'tk'          string  (default "")
                        local to window
                        {not in Vi}
        The key that starts a CTRL-W command in a terminal window.  Other keys
        are sent to the job running in the window.
        The <> notation can be used, e.g.: >
                :set termkey=<C-L>
<       The string must be one key stroke but can be multiple bytes.
        When not set CTRL-W is used, so that CTRL-W : gets you to the command
        line.  If 'termkey' is set to CTRL-L then CTRL-L : gets you to the
        command line.

                                                *'termsize'* *'tms'*
'termsize' 'tms'        string  (default "")
                        local to window
                        {not in Vi}
        Size of the |terminal| window.  Format: {rows}x{columns}.
        - When empty the terminal gets the size from the window.
        - When set (e.g., "24x80") the terminal size is not adjusted to the
          window size.  If the window is smaller only the top-left part is
          displayed.
          When rows is zero then use the height of the window.
          When columns is zero then use the width of the window.
          For example: "30x0" uses 30 rows with the current window width.
          Using "0x0" is the same as empty.
        Note that the command running in the terminal window may still change
        the size of the terminal.  In that case the Vim window will be
        adjusted to that size, if possible.

                                                *'terse'* *'noterse'*
'terse'                 boolean (default off)
                        global
        When set: Add 's' flag to 'shortmess' option (this makes the message
        for a search that hits the start or end of the file not being
        displayed).  When reset: Remove 's' flag from 'shortmess' option.  {Vi
        shortens a lot of messages}

                                    *'textauto'* *'ta'* *'notextauto'* *'nota'*
'textauto' 'ta'         boolean (Vim default: on, Vi default: off)
                        global
                        {not in Vi}
        This option is obsolete.  Use 'fileformats'.
        For backwards compatibility, when 'textauto' is set, 'fileformats' is
        set to the default value for the current system.  When 'textauto' is
        reset, 'fileformats' is made empty.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                    *'textmode'* *'tx'* *'notextmode'* *'notx'*
'textmode' 'tx'         boolean (MS-DOS, Win32 and OS/2: default on,
                                 others: default off)
                        local to buffer
                        {not in Vi}
        This option is obsolete.  Use 'fileformat'.
        For backwards compatibility, when 'textmode' is set, 'fileformat' is
        set to "dos".  When 'textmode' is reset, 'fileformat' is set to
        "unix".

                                                *'textwidth'* *'tw'*
```

```
'textwidth' 'tw'        number  (default 0)
                        local to buffer
                        {not in Vi}
        Maximum width of text that is being inserted.  A longer line will be
        broken after white space to get this width.  A zero value disables
        this.
        'textwidth' is set to 0 when the 'paste' option is set and restored
        when 'paste' is reset.
        When 'textwidth' is zero, 'wrapmargin' may be used.  See also
        'formatoptions' and |ins-textwidth|.
        When 'formatexpr' is set it will be used to break the line.
        NOTE: This option is set to 0 when 'compatible' is set.

                                                *'thesaurus'* *'tsr'*
'thesaurus' 'tsr'       string  (default "")
                        global or local to buffer |global-local|
                        {not in Vi}
        List of file names, separated by commas, that are used to lookup words
        for thesaurus completion commands |i_CTRL-X_CTRL-T|.  Each line in
        the file should contain words with similar meaning, separated by
        non-keyword characters (white space is preferred).  Maximum line
        length is 510 bytes.
        To obtain a file to be used here, check out this ftp site:
        [Sorry this link doesn't work anymore, do you know the right one?]
        ftp://ftp.ox.ac.uk/pub/wordlists/  First get the README file.
        To include a comma in a file name precede it with a backslash.  Spaces
        after a comma are ignored, otherwise spaces are included in the file
        name.  See |option-backslash| about using backslashes.
        The use of |:set+=| and |:set-=| is preferred when adding or removing
        directories from the list.  This avoids problems when a future version
        uses another default.
        Backticks cannot be used in this option for security reasons.

                             *'tildeop'* *'top'* *'notildeop'* *'notop'*
'tildeop' 'top'         boolean (default off)
                        global
                        {not in Vi}
        When on: The tilde command "~" behaves like an operator.
        NOTE: This option is reset when 'compatible' is set.

                                *'timeout'* *'to'* *'notimeout'* *'noto'*
'timeout' 'to'          boolean (default on)
                        global
                                                *'ttimeout'* *'nottimeout'*
'ttimeout'              boolean (default off, set in |defaults.vim|)
                        global
                        {not in Vi}
        These two options together determine the behavior when part of a
        mapped key sequence or keyboard code has been received:

        'timeout'    'ttimeout'         action  ~
           off          off             do not time out
           on           on or off       time out on :mappings and key codes
           off          on              time out on key codes

        If both options are off, Vim will wait until either the complete
        mapping or key sequence has been received, or it is clear that there
        is no mapping or key sequence for the received characters.  For
        example: if you have mapped "vl" and Vim has received 'v', the next
        character is needed to see if the 'v' is followed by an 'l'.
        When one of the options is on, Vim will wait for about 1 second for
        the next character to arrive.  After that the already received
```

```
        characters are interpreted as single characters.  The waiting time can
        be changed with the 'timeoutlen' option.
        On slow terminals or very busy systems timing out may cause
        malfunctioning cursor keys.  If both options are off, Vim waits
        forever after an entered <Esc> if there are key codes that start
        with <Esc>.  You will have to type <Esc> twice.  If you do not have
        problems with key codes, but would like to have :mapped key
        sequences not timing out in 1 second, set the 'ttimeout' option and
        reset the 'timeout' option.

        NOTE: 'ttimeout' is reset when 'compatible' is set.

                                                *'timeoutlen'* *'tm'*
'timeoutlen' 'tm'       number  (default 1000)
                        global
                        {not in all versions of Vi}
                                                *'ttimeoutlen'* *'ttm'*
'ttimeoutlen' 'ttm'     number  (default -1, set to 100 in |defaults.vim|)
                        global
                        {not in Vi}
        The time in milliseconds that is waited for a key code or mapped key
        sequence to complete.  Also used for CTRL-\ CTRL-N and CTRL-\ CTRL-G
        when part of a command has been typed.
        Normally only 'timeoutlen' is used and 'ttimeoutlen' is -1.  When a
        different timeout value for key codes is desired set 'ttimeoutlen' to
        a non-negative number.

                ttimeoutlen     mapping delay      key code delay       ~
                   < 0          'timeoutlen'       'timeoutlen'
                  >= 0          'timeoutlen'       'ttimeoutlen'

        The timeout only happens when the 'timeout' and 'ttimeout' options
        tell so.  A useful setting would be >
                :set timeout timeoutlen=3000 ttimeoutlen=100
<       (time out on mapping after three seconds, time out on key codes after
        a tenth of a second).

                                                *'title'* *'notitle'*
'title'                 boolean (default off, on when title can be restored)
                        global
                        {not in Vi}
                        {not available when compiled without the |+title|
                        feature}
        When on, the title of the window will be set to the value of
        'titlestring' (if it is not empty), or to:
                filename [+=-] (path) - VIM
        Where:
                filename        the name of the file being edited
                -               indicates the file cannot be modified, 'ma' off
                +               indicates the file was modified
                =               indicates the file is read-only
                =+              indicates the file is read-only and modified
                (path)          is the path of the file being edited
                - VIM           the server name |v:servername| or "VIM"
        Only works if the terminal supports setting window titles
        (currently Amiga console, Win32 console, all GUI versions and
        terminals with a non- empty 't_ts' option - these are Unix xterm and
        iris-ansi by default, where 't_ts' is taken from the builtin termcap).
                                                                    *X11*
        When Vim was compiled with HAVE_X11 defined, the original title will
        be restored if possible.  The output of ":version" will include "+X11"
        when HAVE_X11 was defined, otherwise it will be "-X11".  This also
```

```
        works for the icon name |'icon'|.
        But: When Vim was started with the |-X| argument, restoring the title
        will not work (except in the GUI).
        If the title cannot be restored, it is set to the value of 'titleold'.
        You might want to restore the title outside of Vim then.
        When using an xterm from a remote machine you can use this command:
            rsh machine_name xterm -display $DISPLAY &
        then the WINDOWID environment variable should be inherited and the
        title of the window should change back to what it should be after
        exiting Vim.

                                                                *'titlelen'*
'titlelen'              number  (default 85)
                        global
                        {not in Vi}
                        {not available when compiled without the |+title|
                        feature}
        Gives the percentage of 'columns' to use for the length of the window
        title.  When the title is longer, only the end of the path name is
        shown.  A '<' character before the path name is used to indicate this.
        Using a percentage makes this adapt to the width of the window.  But
        it won't work perfectly, because the actual number of characters
        available also depends on the font used and other things in the title
        bar.  When 'titlelen' is zero the full path is used.  Otherwise,
        values from 1 to 30000 percent can be used.
        'titlelen' is also used for the 'titlestring' option.

                                                *'titleold'*
'titleold'              string  (default "Thanks for flying Vim")
                        global
                        {not in Vi}
                        {only available when compiled with the |+title|
                        feature}
        This option will be used for the window title when exiting Vim if the
        original title cannot be restored.  Only happens if 'title' is on or
        'titlestring' is not empty.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.
                                                *'titlestring'*
'titlestring'           string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+title|
                        feature}
        When this option is not empty, it will be used for the title of the
        window.  This happens only when the 'title' option is on.
        Only works if the terminal supports setting window titles (currently
        Amiga console, Win32 console, all GUI versions and terminals with a
        non-empty 't_ts' option).
        When Vim was compiled with HAVE_X11 defined, the original title will
        be restored if possible, see |X11|.
        When this option contains printf-style '%' items, they will be
        expanded according to the rules used for 'statusline'.
        Example: >
    :auto BufEnter * let &titlestring = hostname() . "/" . expand("%:p")
    :set title titlestring=%<%F%=%l/%L-%P titlelen=70
<       The value of 'titlelen' is used to align items in the middle or right
        of the available space.
        Some people prefer to have the file name first: >
    :set titlestring=%t%(\ %M%)%(\ (%{expand(\"%:~:.:h\")})%)%(\ %a%)
<       Note the use of "%{ }" and an expression to get the path of the file,
        without the file name.  The "%( %)" constructs are used to add a
```

```
        separating space only when needed.
        NOTE: Use of special characters in 'titlestring' may cause the display
        to be garbled (e.g., when it contains a CR or NL character).
        {not available when compiled without the |+statusline| feature}

                                *'toolbar'* *'tb'*
'toolbar' 'tb'          string  (default "icons,tooltips")
                        global
                        {only for |+GUI_GTK|, |+GUI_Athena|, |+GUI_Motif| and
                        |+GUI_Photon|}
        The contents of this option controls various toolbar settings.  The
        possible values are:
                icons           Toolbar buttons are shown with icons.
                text            Toolbar buttons shown with text.
                horiz           Icon and text of a toolbar button are
                                horizontally arranged.  {only in GTK+ 2 GUI}
                tooltips        Tooltips are active for toolbar buttons.
        Tooltips refer to the popup help text which appears after the mouse
        cursor is placed over a toolbar button for a brief moment.

        If you want the toolbar to be shown with icons as well as text, do the
        following: >
                :set tb=icons,text
<       Motif and Athena cannot display icons and text at the same time.  They
        will show icons if both are requested.

        If none of the strings specified in 'toolbar' are valid or if
        'toolbar' is empty, this option is ignored.  If you want to disable
        the toolbar, you need to set the 'guioptions' option.  For example: >
                :set guioptions-=T
<       Also see |gui-toolbar|.

                                              *'toolbariconsize'* *'tbis'*
'toolbariconsize' 'tbis'        string  (default "small")
                                global
                                {not in Vi}
                                {only in the GTK+ GUI}
        Controls the size of toolbar icons.  The possible values are:
                tiny            Use tiny icons.
                small           Use small icons (default).
                medium          Use medium-sized icons.
                large           Use large icons.
                huge            Use even larger icons.
                giant           Use very big icons.
        The exact dimensions in pixels of the various icon sizes depend on
        the current theme.  Common dimensions are giant=48x48, huge=32x32,
        large=24x24, medium=24x24, small=20x20 and tiny=16x16.

        If 'toolbariconsize' is empty, the global default size as determined
        by user preferences or the current theme is used.

                             *'ttybuiltin'* *'tbi'* *'nottybuiltin'* *'notbi'*
'ttybuiltin' 'tbi'      boolean (default on)
                        global
                        {not in Vi}
        When on, the builtin termcaps are searched before the external ones.
        When off the builtin termcaps are searched after the external ones.
        When this option is changed, you should set the 'term' option next for
        the change to take effect, for example: >
                :set notbi term=$TERM
<       See also |termcap|.
        Rationale: The default for this option is "on", because the builtin
```

```
        termcap entries are generally better (many systems contain faulty
        xterm entries...).

                                        *'ttyfast'* *'tf'* *'nottyfast'* *'notf'*
'ttyfast' 'tf'          boolean (default off, on when 'term' is xterm, hpterm,
                                    sun-cmd, screen, rxvt, dtterm or
                                    iris-ansi; also on when running Vim in
                                    a DOS console)
                        global
                        {not in Vi}
        Indicates a fast terminal connection.  More characters will be sent to
        the screen for redrawing, instead of using insert/delete line
        commands.  Improves smoothness of redrawing when there are multiple
        windows and the terminal does not support a scrolling region.
        Also enables the extra writing of characters at the end of each screen
        line for lines that wrap.  This helps when using copy/paste with the
        mouse in an xterm and other terminals.

                                                *'ttymouse'* *'ttym'*
'ttymouse' 'ttym'       string  (default depends on 'term')
                        global
                        {not in Vi}
                        {only in Unix and VMS, doesn't work in the GUI; not
                        available when compiled without |+mouse|}
        Name of the terminal type for which mouse codes are to be recognized.
        Currently these strings are valid:
                                                        *xterm-mouse*
           xterm        xterm-like mouse handling.  The mouse generates
                        "<Esc>[Mscr", where "scr" is three bytes:
                                "s"  = button state
                                "c"  = column plus 33
                                "r"  = row plus 33
                        This only works up to 223 columns!  See "dec",
                        "urxvt", and "sgr" for solutions.
           xterm2       Works like "xterm", but with the xterm reporting the
                        mouse position while the mouse is dragged.  This works
                        much faster and more precise.  Your xterm must at
                        least at patchlevel 88 / XFree 3.3.3 for this to
                        work.  See below for how Vim detects this
                        automatically.
                                                        *netterm-mouse*
           netterm      NetTerm mouse handling.  The mouse generates
                        "<Esc>}r,c<CR>", where "r,c" are two decimal numbers
                        for the row and column.
                                                        *dec-mouse*
           dec          DEC terminal mouse handling.  The mouse generates a
                        rather complex sequence, starting with "<Esc>[".
                        This is also available for an Xterm, if it was
                        configured with "--enable-dec-locator".
                                                        *jsbterm-mouse*
           jsbterm      JSB term mouse handling.
                                                        *pterm-mouse*
           pterm        QNX pterm mouse handling.
                                                        *urxvt-mouse*
           urxvt        Mouse handling for the urxvt (rxvt-unicode) terminal.
                        The mouse works only if the terminal supports this
                        encoding style, but it does not have 223 columns limit
                        unlike "xterm" or "xterm2".
                                                        *sgr-mouse*
           sgr          Mouse handling for the terminal that emits SGR-styled
                        mouse reporting.  The mouse works even in columns
                        beyond 223.  This option is backward compatible with
```

```
                        "xterm2" because it can also decode "xterm2" style
                        mouse codes.

        The mouse handling must be enabled at compile time |+mouse_xterm|
        |+mouse_dec| |+mouse_netterm| |+mouse_jsbterm| |+mouse_urxvt|
        |+mouse_sgr|.
        Only "xterm"(2) is really recognized.  NetTerm mouse codes are always
        recognized, if enabled at compile time.  DEC terminal mouse codes
        are recognized if enabled at compile time, and 'ttymouse' is not
        "xterm", "xterm2", "urxvt" or "sgr" (because dec mouse codes conflict
        with them).
        This option is automatically set to "xterm", when the 'term' option is
        set to a name that starts with "xterm", "mlterm", "screen", "tmux",
        "st" (full match only), "st-" or "stterm", and 'ttymouse' is not set
        already.
        Additionally, if vim is compiled with the |+termresponse| feature and
        |t_RV| is set to the escape sequence to request the xterm version
        number, more intelligent detection process runs.
        The "xterm2" value will be set if the xterm version is reported to be
        from 95 to 276.  The "sgr" value will be set if the xterm version is
        277 or highter.
        If you do not want 'ttymouse' to be set to "xterm2" or "sgr"
        automatically, set t_RV to an empty string: >
                :set t_RV=
<
                                                *'ttyscroll'* *'tsl'*
'ttyscroll' 'tsl'       number  (default 999)
                        global
        Maximum number of lines to scroll the screen.  If there are more lines
        to scroll the window is redrawn.  For terminals where scrolling is
        very slow and redrawing is not slow this can be set to a small number,
        e.g., 3, to speed up displaying.

                                                *'ttytype'* *'tty'*
'ttytype' 'tty'         string  (default from $TERM)
                        global
        Alias for 'term', see above.

                                                *'undodir'* *'udir'*
'undodir' 'udir'        string  (default ".")
                        global
                        {not in Vi}
                        {only when compiled with the |+persistent_undo| feature}
        List of directory names for undo files, separated with commas.
        See |'backupdir'| for details of the format.
        "." means using the directory of the file.  The undo file name for
        "file.txt" is ".file.txt.un~".
        For other directories the file name is the full path of the edited
        file, with path separators replaced with "%".
        When writing: The first directory that exists is used. "." always
        works, no directories after "." will be used for writing.
        When reading all entries are tried to find an undo file.  The first
        undo file that exists is used.  When it cannot be read an error is
        given, no further entry is used.
        See |undo-persistence|.

                                *'undofile'* *'noundofile'* *'udf'* *'noudf'*
'undofile' 'udf'        boolean (default off)
                        local to buffer
                        {not in Vi}
                        {only when compiled with the |+persistent_undo| feature}
        When on, Vim automatically saves undo history to an undo file when
```

writing a buffer to a file, and restores undo history from the same
file on buffer read.
The directory where the undo file is stored is specified by 'undodir'.
For more information about this feature see |undo-persistence|.
The undo file is not read when 'undoreload' causes the buffer from
before a reload to be saved for undo.
When 'undofile' is turned off the undo file is NOT deleted.
NOTE: This option is reset when 'compatible' is set.

'undolevels' *'ul'*
'undolevels' 'ul' number (default 100, 1000 for Unix, VMS,
Win32 and OS/2)
global or local to buffer |global-local|
{not in Vi}
Maximum number of changes that can be undone. Since undo information
is kept in memory, higher numbers will cause more memory to be used
(nevertheless, a single change can use an unlimited amount of memory).
Set to 0 for Vi compatibility: One level of undo and "u" undoes
itself: >
set ul=0
< But you can also get Vi compatibility by including the 'u' flag in
'cpoptions', and still be able to use CTRL-R to repeat undo.
Also see |undo-two-ways|.
Set to -1 for no undo at all. You might want to do this only for the
current buffer: >
setlocal ul=-1
< This helps when you run out of memory for a single change.

The local value is set to -123456 when the global value is to be used.

Also see |clear-undo|.

'undoreload' *'ur'*
'undoreload' 'ur' number (default 10000)
global
{not in Vi}
Save the whole buffer for undo when reloading it. This applies to the
":e!" command and reloading for when the buffer changed outside of
Vim. |FileChangedShell|
The save only happens when this option is negative or when the number
of lines is smaller than the value of this option.
Set this option to zero to disable undo for a reload.

When saving undo for a reload, any undo file is not read.

Note that this causes the whole buffer to be stored in memory. Set
this option to a lower value if you run out of memory.

'updatecount' *'uc'*
'updatecount' 'uc' number (default: 200)
global
{not in Vi}
After typing this many characters the swap file will be written to
disk. When zero, no swap file will be created at all (see chapter on
recovery |crash-recovery|). 'updatecount' is set to zero by starting
Vim with the "-n" option, see |startup|. When editing in readonly
mode this option will be initialized to 10000.
The swapfile can be disabled per buffer with |'swapfile'|.
When 'updatecount' is set from zero to non-zero, swap files are
created for all buffers that have 'swapfile' set. When 'updatecount'
is set to zero, existing swap files are not deleted.
Also see |'swapsync'|.

```
        This option has no meaning in buffers where |'buftype'| is "nofile"
        or "nowrite".

                                                *'updatetime'* *'ut'*
'updatetime' 'ut'       number  (default 4000)
                        global
                        {not in Vi}
        If this many milliseconds nothing is typed the swap file will be
        written to disk (see |crash-recovery|).  Also used for the
        |CursorHold| autocommand event.

                                                *'verbose'* *'vbs'*
'verbose' 'vbs'         number  (default 0)
                        global
                        {not in Vi, although some versions have a boolean
                        verbose option}
        When bigger than zero, Vim will give messages about what it is doing.
        Currently, these messages are given:
        >= 1    When the viminfo file is read or written.
        >= 2    When a file is ":source"'ed.
        >= 5    Every searched tags file and include file.
        >= 8    Files for which a group of autocommands is executed.
        >= 9    Every executed autocommand.
        >= 12   Every executed function.
        >= 13   When an exception is thrown, caught, finished, or discarded.
        >= 14   Anything pending in a ":finally" clause.
        >= 15   Every executed Ex command (truncated at 200 characters).

        This option can also be set with the "-V" argument.  See |-V|.
        This option is also set by the |:verbose| command.

        When the 'verbosefile' option is set then the verbose messages are not
        displayed.

                                                *'verbosefile'* *'vfile'*
'verbosefile' 'vfile'   string  (default empty)
                        global
                        {not in Vi}
        When not empty all messages are written in a file with this name.
        When the file exists messages are appended.
        Writing to the file ends when Vim exits or when 'verbosefile' is made
        empty.  Writes are buffered, thus may not show up for some time.
        Setting 'verbosefile' to a new value is like making it empty first.
        The difference with |:redir| is that verbose messages are not
        displayed when 'verbosefile' is set.

                                                *'viewdir'* *'vdir'*
'viewdir' 'vdir'        string  (default for Amiga, MS-DOS, OS/2 and Win32:
                                                        "$VIM/vimfiles/view",
                                 for Unix: "~/.vim/view",
                                 for Macintosh: "$VIM:vimfiles:view"
                                 for VMS: "sys$login:vimfiles/view"
                                 for RiscOS: "Choices:vimfiles/view")
                        global
                        {not in Vi}
                        {not available when compiled without the |+mksession|
                        feature}
        Name of the directory where to store files for |:mkview|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'viewoptions'* *'vop'*
```

```
'viewoptions' 'vop'     string  (default: "folds,options,cursor")
                        global
                        {not in Vi}
                        {not available when compiled without the |+mksession|
                        feature}
        Changes the effect of the |:mkview| command.  It is a comma separated
        list of words.  Each word enables saving and restoring something:
           word         save and restore ~
           cursor       cursor position in file and in window
           folds        manually created folds, opened/closed folds and local
                        fold options
           options      options and mappings local to a window or buffer (not
                        global values for local options)
           localoptions same as "options"
           slash        backslashes in file names replaced with forward
                        slashes
           unix         with Unix end-of-line format (single <NL>), even when
                        on Windows or DOS

        "slash" and "unix" are useful on Windows when sharing view files
        with Unix.  The Unix version of Vim cannot source dos format scripts,
        but the Windows version of Vim can source unix format scripts.

                                *'viminfo'* *'vi'* *E526* *E527* *E528*
'viminfo' 'vi'          string  (Vi default: "", Vim default for MS-DOS,
                                   Windows and OS/2: '100,<50,s10,h,rA:,rB:,
                                   for Amiga: '100,<50,s10,h,rdf0:,rdf1:,rdf2:
                                   for others: '100,<50,s10,h)
                        global
                        {not in Vi}
                        {not available when compiled without the |+viminfo|
                        feature}
        When non-empty, the viminfo file is read upon startup and written
        when exiting Vim (see |viminfo-file|). Except when 'viminfofile' is
        "NONE".
        The string should be a comma separated list of parameters, each
        consisting of a single character identifying the particular parameter,
        followed by a number or string which specifies the value of that
        parameter.  If a particular character is left out, then the default
        value is used for that parameter.  The following is a list of the
        identifying characters and the effect of their value.
        CHAR    VALUE   ~
                                                        *viminfo-!*
        !       When included, save and restore global variables that start
                with an uppercase letter, and don't contain a lowercase
                letter.  Thus "KEEPTHIS and "K_L_M" are stored, but "KeepThis"
                and "_K_L_M" are not.  Nested List and Dict items may not be
                read back correctly, you end up with an empty item.
                                                        *viminfo-quote*
        "       Maximum number of lines saved for each register.  Old name of
                the '<' item, with the disadvantage that you need to put a
                backslash before the ", otherwise it will be recognized as the
                start of a comment!
                                                        *viminfo-%*
        %       When included, save and restore the buffer list.  If Vim is
                started with a file name argument, the buffer list is not
                restored.  If Vim is started without a file name argument, the
                buffer list is restored from the viminfo file.  Quickfix
                ('buftype'), unlisted ('buflisted'), unnamed and buffers on
                removable media (|viminfo-r|) are not saved.
                When followed by a number, the number specifies the maximum
                number of buffers that are stored.  Without a number all
```

```
                buffers are stored.
                                                        *viminfo-'*
        '       Maximum number of previously edited files for which the marks
                are remembered.  This parameter must always be included when
                'viminfo' is non-empty.
                Including this item also means that the |jumplist| and the
                |changelist| are stored in the viminfo file.
                                                        *viminfo-/*
        /       Maximum number of items in the search pattern history to be
                saved.  If non-zero, then the previous search and substitute
                patterns are also saved.  When not included, the value of
                'history' is used.
                                                        *viminfo-:*
        :       Maximum number of items in the command-line history to be
                saved.  When not included, the value of 'history' is used.
                                                        *viminfo-<*
        <       Maximum number of lines saved for each register.  If zero then
                registers are not saved.  When not included, all lines are
                saved.  '"' is the old name for this item.
                Also see the 's' item below: limit specified in Kbyte.
                                                        *viminfo-@*
        @       Maximum number of items in the input-line history to be
                saved.  When not included, the value of 'history' is used.
                                                        *viminfo-c*
        c       When included, convert the text in the viminfo file from the
                'encoding' used when writing the file to the current
                'encoding'.  See |viminfo-encoding|.
                                                        *viminfo-f*
        f       Whether file marks need to be stored.  If zero, file marks ('0
                to '9, 'A to 'Z) are not stored.  When not present or when
                non-zero, they are all stored.  '0 is used for the current
                cursor position (when exiting or when doing ":wviminfo").
                                                        *viminfo-h*
        h       Disable the effect of 'hlsearch' when loading the viminfo
                file.  When not included, it depends on whether ":nohlsearch"
                has been used since the last search command.
                                                        *viminfo-n*
        n       Name of the viminfo file.  The name must immediately follow
                the 'n'.  Must be at the end of the option!  If the
                'viminfofile' option is set, that file name overrides the one
                given here with 'viminfo'.  Environment variables are
                expanded when opening the file, not when setting the option.
                                                        *viminfo-r*
        r       Removable media.  The argument is a string (up to the next
                ',').  This parameter can be given several times.  Each
                specifies the start of a path for which no marks will be
                stored.  This is to avoid removable media.  For MS-DOS you
                could use "ra:,rb:", for Amiga "rdf0:,rdf1:,rdf2:".  You can
                also use it for temp files, e.g., for Unix: "r/tmp".  Case is
                ignored.  Maximum length of each 'r' argument is 50
                characters.
                                                        *viminfo-s*
        s       Maximum size of an item in Kbyte.  If zero then registers are
                not saved.  Currently only applies to registers.  The default
                "s10" will exclude registers with more than 10 Kbyte of text.
                Also see the '<' item above: line count limit.

        Example: >
            :set viminfo='50,<1000,s100,:0,n~/vim/viminfo
<
        '50             Marks will be remembered for the last 50 files you
                        edited.
```

```
        <1000           Contents of registers (up to 1000 lines each) will be
                        remembered.
        s100            Registers with more than 100 Kbyte text are skipped.
        :0              Command-line history will not be saved.
        n~/vim/viminfo  The name of the file to use is "~/vim/viminfo".
        no /            Since '/' is not specified, the default will be used,
                        that is, save all of the search history, and also the
                        previous search and substitute patterns.
        no %            The buffer list will not be saved nor read back.
        no h            'hlsearch' highlighting will be restored.

        When setting 'viminfo' from an empty value you can use |:rviminfo| to
        load the contents of the file, this is not done automatically.

        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.
        NOTE: This option is set to the Vim default value when 'compatible'
        is reset.

                                *'viminfofile'* *'vif'*
'viminfofile' 'vif'     string  (default: "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+viminfo|
                        feature}
        When non-empty, overrides the file name used for viminfo.
        When equal to "NONE" no viminfo file will be read or written.
        This option can be set with the |-i| command line flag.  The |--clean|
        command line flag sets it to "NONE".

                                            *'virtualedit'* *'ve'*
'virtualedit' 've'      string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the
                        |+virtualedit| feature}
        A comma separated list of these words:
            block       Allow virtual editing in Visual block mode.
            insert      Allow virtual editing in Insert mode.
            all         Allow virtual editing in all modes.
            onemore     Allow the cursor to move just past the end of the line

        Virtual editing means that the cursor can be positioned where there is
        no actual character.  This can be halfway into a tab or beyond the end
        of the line.  Useful for selecting a rectangle in Visual mode and
        editing a table.
        "onemore" is not the same, it will only allow moving the cursor just
        after the last character of the line.  This makes some commands more
        consistent.  Previously the cursor was always past the end of the line
        if the line was empty.  But it is far from Vi compatible.  It may also
        break some plugins or Vim scripts.  For example because |l| can move
        the cursor after the last character.  Use with care!
        Using the `$` command will move to the last character in the line, not
        past it.  This may actually move the cursor to the left!
        The `g$` command will move to the end of the screen line.
        It doesn't make sense to combine "all" with "onemore", but you will
        not get a warning for it.
        NOTE: This option is set to "" when 'compatible' is set.

                        *'visualbell'* *'vb'* *'novisualbell'* *'novb'* *beep*
'visualbell' 'vb'       boolean (default off)
                        global
```

```
                        {not in Vi}
        Use a visual bell instead of beeping.  The terminal code to display the
        visual bell is given with 't_vb'.  When no beep or flash is wanted,
        use: >
                :set vb t_vb=
<       If you want a short flash, you can use this on many terminals: >
                :set vb t_vb=[?5h$<100>[?5l
<       Here $<100> specifies the time, you can use a smaller or bigger value
        to get a shorter or longer flash.

        Note: Vim will limit the bell to once per half a second.  This avoids
        having to wait for the flashing to finish when there are lots of
        bells, e.g. on key repeat.  This also happens without 'visualbell'
        set.

        In the GUI, 't_vb' defaults to "<Esc>|f", which inverts the display
        for 20 msec.  If you want to use a different time, use "<Esc>|40f",
        where 40 is the time in msec.

        Note: When the GUI starts, 't_vb' is reset to its default value.  You
        might want to set it again in your |gvimrc|.

        Does not work on the Amiga, you always get a screen flash.
        Also see 'errorbells'.

                                                *'warn'* *'nowarn'*
'warn'                  boolean (default on)
                        global
        Give a warning message when a shell command is used while the buffer
        has been changed.

                    *'weirdinvert'* *'wiv'* *'noweirdinvert'* *'nowiv'*
'weirdinvert' 'wiv'     boolean (default off)
                        global
                        {not in Vi}
        This option has the same effect as the 't_xs' terminal option.
        It is provided for backwards compatibility with version 4.x.
        Setting 'weirdinvert' has the effect of making 't_xs' non-empty, and
        vice versa.  Has no effect when the GUI is running.

                                                *'whichwrap'* *'ww'*
'whichwrap' 'ww'        string  (Vim default: "b,s", Vi default: "")
                        global
                        {not in Vi}
        Allow specified keys that move the cursor left/right to move to the
        previous/next line when the cursor is on the first/last character in
        the line.  Concatenate characters to allow this for these keys:
                char   key        mode  ~
                 b    <BS>       Normal and Visual
                 s    <Space>    Normal and Visual
                 h    "h"        Normal and Visual (not recommended)
                 l    "l"        Normal and Visual (not recommended)
                 <    <Left>     Normal and Visual
                 >    <Right>    Normal and Visual
                 ~    "~"        Normal
                 [    <Left>     Insert and Replace
                 ]    <Right>    Insert and Replace
        For example: >
                :set ww=<,>,[,]
<       allows wrap only when cursor keys are used.
        When the movement keys are used in combination with a delete or change
        operator, the <EOL> also counts for a character.  This makes "3h"
```

```
        different from "3dh" when the cursor crosses the end of a line.  This
        is also true for "x" and "X", because they do the same as "dl" and
        "dh".  If you use this, you may also want to use the mapping
        ":map <BS> X" to make backspace delete the character in front of the
        cursor.
        When 'l' is included and it is used after an operator at the end of a
        line then it will not move to the next line.  This makes "dl", "cl",
        "yl" etc. work normally.
        NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'wildchar'* *'wc'*
'wildchar' 'wc'         number  (Vim default: <Tab>, Vi default: CTRL-E)
                        global
                        {not in Vi}
        Character you have to type to start wildcard expansion in the
        command-line, as specified with 'wildmode'.
        More info here: |cmdline-completion|.
        The character is not recognized when used inside a macro.  See
        'wildcharm' for that.
        Although 'wc' is a number option, you can set it to a special key: >
                :set wc=<Esc>
<       NOTE: This option is set to the Vi default value when 'compatible' is
        set and to the Vim default value when 'compatible' is reset.

                                                *'wildcharm'* *'wcm'*
'wildcharm' 'wcm'       number  (default: none (0))
                        global
                        {not in Vi}
        'wildcharm' works exactly like 'wildchar', except that it is
        recognized when used inside a macro.  You can find "spare" command-line
        keys suitable for this option by looking at |ex-edit-index|.  Normally
        you'll never actually type 'wildcharm', just use it in mappings that
        automatically invoke completion mode, e.g.: >
                :set wcm=<C-Z>
                :cnoremap ss so $vim/sessions/*.vim<C-Z>
<       Then after typing :ss you can use CTRL-P & CTRL-N.

                                                *'wildignore'* *'wig'*
'wildignore' 'wig'      string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+wildignore|
                        feature}
        A list of file patterns.  A file that matches with one of these
        patterns is ignored when expanding |wildcards|, completing file or
        directory names, and influences the result of |expand()|, |glob()| and
        |globpath()| unless a flag is passed to disable this.
        The pattern is used like with |:autocmd|, see |autocmd-patterns|.
        Also see 'suffixes'.
        Example: >
                :set wildignore=*.o,*.obj
<       The use of |:set+=| and |:set-=| is preferred when adding or removing
        a pattern from the list.  This avoids problems when a future version
        uses another default.


                        *'wildignorecase'* *'wic'* *'nowildignorecase'* *'nowic'*
'wildignorecase' 'wic'  boolean (default off)
                        global
                        {not in Vi}
        When set case is ignored when completing file names and directories.
```

```
        Has no effect when 'fileignorecase' is set.
        Does not apply when the shell is used to expand wildcards, which
        happens when there are special characters.


                                  *'wildmenu'* *'wmnu'* *'nowildmenu'* *'nowmnu'*
'wildmenu' 'wmnu'       boolean (default off, set in |defaults.vim|)
                        global
                        {not in Vi}
                        {not available if compiled without the |+wildmenu|
                        feature}
        When 'wildmenu' is on, command-line completion operates in an enhanced
        mode.  On pressing 'wildchar' (usually <Tab>) to invoke completion,
        the possible matches are shown just above the command line, with the
        first match highlighted (overwriting the status line, if there is
        one).  Keys that show the previous/next match, such as <Tab> or
        CTRL-P/CTRL-N, cause the highlight to move to the appropriate match.
        When 'wildmode' is used, "wildmenu" mode is used where "full" is
        specified.  "longest" and "list" do not start "wildmenu" mode.
        You can check the current mode with |wildmenumode()|.
        If there are more matches than can fit in the line, a ">" is shown on
        the right and/or a "<" is shown on the left.  The status line scrolls
        as needed.
        The "wildmenu" mode is abandoned when a key is hit that is not used
        for selecting a completion.
        While the "wildmenu" is active the following keys have special
        meanings:

        <Left> <Right>  - select previous/next match (like CTRL-P/CTRL-N)
        <Down>          - in filename/menu name completion: move into a
                          subdirectory or submenu.
        <CR>            - in menu completion, when the cursor is just after a
                          dot: move into a submenu.
        <Up>            - in filename/menu name completion: move up into
                          parent directory or parent menu.

        This makes the menus accessible from the console |console-menus|.

        If you prefer the <Left> and <Right> keys to move the cursor instead
        of selecting a different match, use this: >
                :cnoremap <Left> <Space><BS><Left>
                :cnoremap <Right> <Space><BS><Right>
<
        The "WildMenu" highlighting is used for displaying the current match
        |hl-WildMenu|.

                                                *'wildmode'* *'wim'*
'wildmode' 'wim'        string  (Vim default: "full")
                        global
                        {not in Vi}
        Completion mode that is used for the character specified with
        'wildchar'.  It is a comma separated list of up to four parts.  Each
        part specifies what to do for each consecutive use of 'wildchar'.  The
        first part specifies the behavior for the first use of 'wildchar',
        The second part for the second use, etc.
        These are the possible values for each part:
        ""              Complete only the first match.
        "full"          Complete the next full match.  After the last match,
                        the original string is used and then the first match
                        again.
        "longest"       Complete till longest common string.  If this doesn't
                        result in a longer string, use the next part.
```

```
        "longest:full"  Like "longest", but also start 'wildmenu' if it is
                        enabled.
        "list"          When more than one match, list all matches.
        "list:full"     When more than one match, list all matches and
                        complete first match.
        "list:longest"  When more than one match, list all matches and
                        complete till longest common string.
        When there is only a single match, it is fully completed in all cases.

        Examples: >
                :set wildmode=full
<       Complete first full match, next match, etc.  (the default) >
                :set wildmode=longest,full
<       Complete longest common string, then each full match >
                :set wildmode=list:full
<       List all matches and complete each full match >
                :set wildmode=list,full
<       List all matches without completing, then each full match >
                :set wildmode=longest,list
<       Complete longest common string, then list alternatives.
        More info here: |cmdline-completion|.

                                                *'wildoptions'* *'wop'*
'wildoptions' 'wop'     string  (default "")
                        global
                        {not in Vi}
                        {not available when compiled without the |+wildignore|
                        feature}
        A list of words that change how command line completion is done.
        Currently only one word is allowed:
          tagfile       When using CTRL-D to list matching tags, the kind of
                        tag and the file of the tag is listed.  Only one match
                        is displayed per line.  Often used tag kinds are:
                                d       #define
                                f       function
        Also see |cmdline-completion|.

                                                *'winaltkeys'* *'wak'*
'winaltkeys' 'wak'      string  (default "menu")
                        global
                        {not in Vi}
                        {only used in Win32, Motif, GTK and Photon GUI}
        Some GUI versions allow the access to menu entries by using the ALT
        key in combination with a character that appears underlined in the
        menu.  This conflicts with the use of the ALT key for mappings and
        entering special characters.  This option tells what to do:
          no    Don't use ALT keys for menus.  ALT key combinations can be
                mapped, but there is no automatic handling.  This can then be
                done with the |:simalt| command.
          yes   ALT key handling is done by the windowing system.  ALT key
                combinations cannot be mapped.
          menu  Using ALT in combination with a character that is a menu
                shortcut key, will be handled by the windowing system.  Other
                keys can be mapped.
        If the menu is disabled by excluding 'm' from 'guioptions', the ALT
        key is never used for the menu.
        This option is not used for <F10>; on Win32 and with GTK <F10> will
        select the menu, unless it has been mapped.

                                                *'window'* *'wi'*
'window' 'wi'           number  (default screen height - 1)
                        global
```

```
        Window height.  Do not confuse this with the height of the Vim window,
        use 'lines' for that.
        Used for |CTRL-F| and |CTRL-B| when there is only one window and the
        value is smaller than 'lines' minus one.  The screen will scroll
        'window' minus two lines, with a minimum of one.
        When 'window' is equal to 'lines' minus one CTRL-F and CTRL-B scroll
        in a much smarter way, taking care of wrapping lines.
        When resizing the Vim window, the value is smaller than 1 or more than
        or equal to 'lines' it will be set to 'lines' minus 1.
        {Vi also uses the option to specify the number of displayed lines}

                                                    *'winheight'* *'wh'* *E591*
'winheight' 'wh'        number  (default 1)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        Minimal number of lines for the current window.  This is not a hard
        minimum, Vim will use fewer lines if there is not enough room.  If the
        focus goes to a window that is smaller, its size is increased, at the
        cost of the height of other windows.
        Set 'winheight' to a small number for normal editing.
        Set it to 999 to make the current window fill most of the screen.
        Other windows will be only 'winminheight' high.  This has the drawback
        that ":all" will create only two windows.  To avoid "vim -o 1 2 3 4"
        to create only two windows, set the option after startup is done,
        using the |VimEnter| event: >
                au VimEnter * set winheight=999
<       Minimum value is 1.
        The height is not adjusted after one of the commands that change the
        height of the current window.
        'winheight' applies to the current window.  Use 'winminheight' to set
        the minimal height for other windows.

                        *'winfixheight'* *'wfh'* *'nowinfixheight'* *'nowfh'*
'winfixheight' 'wfh'    boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        Keep the window height when windows are opened or closed and
        'equalalways' is set.  Also for |CTRL-W_=|.  Set by default for the
        |preview-window| and |quickfix-window|.
        The height may be changed anyway when running out of room.

                        *'winfixwidth'* *'wfw'* *'nowinfixwidth'* *'nowfw'*
'winfixwidth' 'wfw'     boolean (default off)
                        local to window
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        Keep the window width when windows are opened or closed and
        'equalalways' is set.  Also for |CTRL-W_=|.
        The width may be changed anyway when running out of room.

                                                    *'winminheight'* *'wmh'*
'winminheight' 'wmh'    number  (default 1)
                        global
                        {not in Vi}
                        {not available when compiled without the |+windows|
                        feature}
        The minimal height of a window, when it's not the current window.
```

```
        This is a hard minimum, windows will never become smaller.
        When set to zero, windows may be "squashed" to zero lines (i.e. just a
        status bar) if necessary.  They will return to at least one line when
        they become active (since the cursor has to have somewhere to go.)
        Use 'winheight' to set the minimal height of the current window.
        This option is only checked when making a window smaller.  Don't use a
        large number, it will cause errors when opening more than a few
        windows.  A value of 0 to 3 is reasonable.

                                                *'winminwidth'* *'wmw'*
'winminwidth' 'wmw'     number  (default 1)
                        global
                        {not in Vi}
                        {not available when compiled without the |+vertsplit|
                        feature}
        The minimal width of a window, when it's not the current window.
        This is a hard minimum, windows will never become smaller.
        When set to zero, windows may be "squashed" to zero columns (i.e. just
        a vertical separator) if necessary.  They will return to at least one
        line when they become active (since the cursor has to have somewhere
        to go.)
        Use 'winwidth' to set the minimal width of the current window.
        This option is only checked when making a window smaller.  Don't use a
        large number, it will cause errors when opening more than a few
        windows.  A value of 0 to 12 is reasonable.

                                                *'winptydll'*
'winptydll'             string  (default "winpty32.dll" or "winpty64.dll")
                        global
                        {not in Vi}
                        {only available when compiled with the |terminal|
                        feature on MS-Windows}
        Specifies the name of the winpty shared library, used for the
        |:terminal| command. The default depends on whether was build as a
        32-bit or 64-bit executable.  If not found, "winpty.dll" is tried as
        a fallback.
        Environment variables are expanded |:set_env|.
        This option cannot be set from a |modeline| or in the |sandbox|, for
        security reasons.

                                                *'winwidth'* *'wiw'* *E592*
'winwidth' 'wiw'        number  (default 20)
                        global
                        {not in Vi}
                        {not available when compiled without the |+vertsplit|
                        feature}
        Minimal number of columns for the current window.  This is not a hard
        minimum, Vim will use fewer columns if there is not enough room.  If
        the current window is smaller, its size is increased, at the cost of
        the width of other windows.  Set it to 999 to make the current window
        always fill the screen.  Set it to a small number for normal editing.
        The width is not adjusted after one of the commands to change the
        width of the current window.
        'winwidth' applies to the current window.  Use 'winminwidth' to set
        the minimal width for other windows.

                                                *'wrap'* *'nowrap'*
'wrap'                  boolean (default on)
                        local to window
                        {not in Vi}
        This option changes how text is displayed.  It doesn't change the text
        in the buffer, see 'textwidth' for that.
```

When on, lines longer than the width of the window will wrap and
displaying continues on the next line. When off lines will not wrap
and only part of long lines will be displayed. When the cursor is
moved to a part that is not shown, the screen will scroll
horizontally.
The line will be broken in the middle of a word if necessary. See
'linebreak' to get the break at a word boundary.
To make scrolling horizontally a bit more useful, try this: >
:set sidescroll=5
:set listchars+=precedes:<,extends:>
< See 'sidescroll', 'listchars' and |wrap-off|.
This option can't be set from a |modeline| when the 'diff' option is
on.

'wrapmargin' *'wm'*
'wrapmargin' 'wm' number (default 0)
local to buffer
Number of characters from the right window border where wrapping
starts. When typing text beyond this limit, an <EOL> will be inserted
and inserting continues on the next line.
Options that add a margin, such as 'number' and 'foldcolumn', cause
the text width to be further reduced. This is Vi compatible.
When 'textwidth' is non-zero, this option is not used.
This option is set to 0 when 'paste' is set and restored when 'paste'
is reset.
See also 'formatoptions' and |ins-textwidth|. {Vi: works differently
and less usefully}

'wrapscan' *'ws'* *'nowrapscan'* *'nows'*
'wrapscan' 'ws' boolean (default on) *E384* *E385*
global
Searches wrap around the end of the file. Also applies to |]s| and
|[s|, searching for spelling mistakes.

'write' *'nowrite'*
'write' boolean (default on)
global
{not in Vi}
Allows writing files. When not set, writing a file is not allowed.
Can be used for a view-only mode, where modifications to the text are
still allowed. Can be reset with the |-m| or |-M| command line
argument. Filtering text is still possible, even though this requires
writing a temporary file.

'writeany' *'wa'* *'nowriteany'* *'nowa'*
'writeany' 'wa' boolean (default off)
global
Allows writing to any file with no need for "!" override.

'writebackup' *'wb'* *'nowritebackup'* *'nowb'*
'writebackup' 'wb' boolean (default on with |+writebackup| feature, off
otherwise)
global
{not in Vi}
Make a backup before overwriting a file. The backup is removed after
the file was successfully written, unless the 'backup' option is
also on.
WARNING: Switching this option off means that when Vim fails to write
your buffer correctly and then, for whatever reason, Vim exits, you
lose both the original file and what you were writing. Only reset
this option if your file system is almost full and it makes the write
fail (and make sure not to exit Vim until the write was successful).

```
        See |backup-table| for another explanation.
        When the 'backupskip' pattern matches, a backup is not made anyway.
        NOTE: This option is set to the default value when 'compatible' is
        set.

                                                *'writedelay'* *'wd'*
'writedelay' 'wd'       number  (default 0)
                        global
                        {not in Vi}
        The number of milliseconds to wait for each character sent to the
        screen.  When non-zero, characters are sent to the terminal one by
        one.  For MS-DOS pcterm this does not work.  For debugging purposes.

 vim:tw=78:ts=8:ft=help:norl:
*pattern.txt*   For Vim version 8.0.  Last change: 2017 Jun 05


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Patterns and search commands                              *pattern-searches*

The very basics can be found in section |03.9| of the user manual.  A few more
explanations are in chapter 27 |usr_27.txt|.

1. Search commands              |search-commands|
2. The definition of a pattern  |search-pattern|
3. Magic                        |/magic|
4. Overview of pattern items    |pattern-overview|
5. Multi items                  |pattern-multi-items|
6. Ordinary atoms               |pattern-atoms|
7. Ignoring case in a pattern   |/ignorecase|
8. Composing characters         |patterns-composing|
9. Compare with Perl patterns   |perl-patterns|
10. Highlighting matches        |match-highlight|

==============================================================================
1. Search commands                                *search-commands*

                                                          */*
/{pattern}[/]<CR>       Search forward for the [count]'th occurrence of
                        {pattern} |exclusive|.

/{pattern}/{offset}<CR> Search forward for the [count]'th occurrence of
                        {pattern} and go |{offset}| lines up or down.
                        |linewise|.

                                                          */<CR>*
/<CR>                   Search forward for the [count]'th occurrence of the
                        latest used pattern |last-pattern| with latest used
                        |{offset}|.

//{offset}<CR>          Search forward for the [count]'th occurrence of the
                        latest used pattern |last-pattern| with new
                        |{offset}|.  If {offset} is empty no offset is used.

                                                          *?*
?{pattern}[?]<CR>       Search backward for the [count]'th previous
                        occurrence of {pattern} |exclusive|.

?{pattern}?{offset}<CR> Search backward for the [count]'th previous
                        occurrence of {pattern} and go |{offset}| lines up or
```

down |linewise|.

?<CR>

?<CR> Search backward for the [count]'th occurrence of the latest used pattern |last-pattern| with latest used |{offset}|.

??{offset}<CR> Search backward for the [count]'th occurrence of the latest used pattern |last-pattern| with new |{offset}|. If {offset} is empty no offset is used.

n

n Repeat the latest "/" or "?" [count] times. If the cursor doesn't move the search is repeated with count + 1. |last-pattern| {Vi: no count}

N

N Repeat the latest "/" or "?" [count] times in opposite direction. |last-pattern| {Vi: no count}

star *E348* *E349*

* Search forward for the [count]'th occurrence of the word nearest to the cursor. The word used for the search is the first of:

1. the keyword under the cursor |'iskeyword'|
2. the first keyword after the cursor, in the current line
3. the non-blank word under the cursor
4. the first non-blank word after the cursor, in the current line

Only whole keywords are searched for, like with the command "/\<keyword\>". |exclusive| {not in Vi} 'ignorecase' is used, 'smartcase' is not.

#

Same as "*", but search backward. The pound sign (character 163) also works. If the "#" key works as backspace, try using "stty erase <BS>" before starting Vim (<BS> is CTRL-H or a real backspace). {not in Vi}

gstar

g* Like "*", but don't put "\<" and "\>" around the word. This makes the search also find matches that are not a whole word. {not in Vi}

g#

g# Like "#", but don't put "\<" and "\>" around the word. This makes the search also find matches that are not a whole word. {not in Vi}

gd

gd Goto local Declaration. When the cursor is on a local variable, this command will jump to its declaration. First Vim searches for the start of the current function, just like "[[". If it is not found the search stops in line 1. If it is found, Vim goes back until a blank line is found. From this position Vim searches for the keyword under the cursor, like with "*", but lines that look like a comment are ignored (see 'comments' option).
Note that this is not guaranteed to work, Vim does not

really check the syntax, it only searches for a match
with the keyword. If included files also need to be
searched use the commands listed in |include-search|.
After this command |n| searches forward for the next
match (not backward).
{not in Vi}

gD
gD Goto global Declaration. When the cursor is on a
global variable that is defined in the file, this
command will jump to its declaration. This works just
like "gd", except that the search for the keyword
always starts in line 1. {not in Vi}

1gd
1gd Like "gd", but ignore matches inside a {} block that
ends before the cursor position. {not in Vi}

1gD
1gD Like "gD", but ignore matches inside a {} block that
ends before the cursor position. {not in Vi}

CTRL-C
CTRL-C Interrupt current (search) command. Use CTRL-Break on
MS-DOS |dos-CTRL-Break|.
In Normal mode, any pending command is aborted.

:noh *:nohlsearch*
:noh[lsearch] Stop the highlighting for the 'hlsearch' option. It
is automatically turned back on when using a search
command, or setting the 'hlsearch' option.
This command doesn't work in an autocommand, because
the highlighting state is saved and restored when
executing autocommands |autocmd-searchpat|.
Same thing for when invoking a user function.

While typing the search pattern the current match will be shown if the
'incsearch' option is on. Remember that you still have to finish the search
command with <CR> to actually position the cursor at the displayed match. Or
use <Esc> to abandon the search.

All matches for the last used search pattern will be highlighted if you set
the 'hlsearch' option. This can be suspended with the |:nohlsearch| command.

When no match is found you get the error: *E486* Pattern not found
Note that for the |:global| command this behaves like a normal message, for Vi
compatibility. For the |:s| command the "e" flag can be used to avoid the
error message |:s_flags|.

search-offset *{offset}*
These commands search for the specified pattern. With "/" and "?" an
additional offset may be given. There are two types of offsets: line offsets
and character offsets. {the character offsets are not in Vi}

The offset gives the cursor position relative to the found match:

[num]	[num] lines downwards, in column 1
+[num]	[num] lines downwards, in column 1
-[num]	[num] lines upwards, in column 1
e[+num]	[num] characters to the right of the end of the match
e[-num]	[num] characters to the left of the end of the match
s[+num]	[num] characters to the right of the start of the match
s[-num]	[num] characters to the left of the start of the match

```
    b[+num]     [num] identical to s[+num] above (mnemonic: begin)
    b[-num]     [num] identical to s[-num] above (mnemonic: begin)
    ;{pattern}  perform another search, see |//;|

If a '-' or '+' is given but [num] is omitted, a count of one will be used.
When including an offset with 'e', the search becomes inclusive (the
character the cursor lands on is included in operations).

Examples:

pattern                 cursor position ~
/test/+1                one line below "test", in column 1
/test/e                 on the last t of "test"
/test/s+2               on the 's' of "test"
/test/b-3               three characters before "test"

If one of these commands is used after an operator, the characters between
the cursor position before and after the search is affected.  However, if a
line offset is given, the whole lines between the two cursor positions are
affected.

An example of how to search for matches with a pattern and change the match
with another word: >
        /foo<CR>        find "foo"
        c//e<CR>        change until end of match
        bar<Esc>        type replacement
        //<CR>          go to start of next match
        c//e<CR>        change until end of match
        beep<Esc>       type another replacement
                        etc.
<
                                                        *//;* *E386*
A very special offset is ';' followed by another search command.  For example: >

   /test 1/;/test
   /test.*/+1;?ing?

The first one first finds the next occurrence of "test 1", and then the first
occurrence of "test" after that.

This is like executing two search commands after each other, except that:
- It can be used as a single motion command after an operator.
- The direction for a following "n" or "N" command comes from the first
  search command.
- When an error occurs the cursor is not moved at all.

                                                        *last-pattern*
The last used pattern and offset are remembered.  They can be used to repeat
the search, possibly in another direction or with another count.  Note that
two patterns are remembered: One for 'normal' search commands and one for the
substitute command ":s".  Each time an empty pattern is given, the previously
used pattern is used.  However, if there is no previous search command, a
previous substitute pattern is used, if possible.

The 'magic' option sticks with the last used pattern.  If you change 'magic',
this will not change how the last used pattern will be interpreted.
The 'ignorecase' option does not do this.  When 'ignorecase' is changed, it
will result in the pattern to match other text.

All matches for the last used search pattern will be highlighted if you set
the 'hlsearch' option.
```

```
To clear the last used search pattern: >
        :let @/ = ""
This will not set the pattern to an empty string, because that would match
everywhere.  The pattern is really cleared, like when starting Vim.

The search usually skips matches that don't move the cursor.  Whether the next
match is found at the next character or after the skipped match depends on the
'c' flag in 'cpoptions'.  See |cpo-c|.
          with 'c' flag:   "/..." advances 1 to 3 characters
       without 'c' flag:   "/..." advances 1 character
The unpredictability with the 'c' flag is caused by starting the search in the
first column, skipping matches until one is found past the cursor position.

When searching backwards, searching starts at the start of the line, using the
'c' flag in 'cpoptions' as described above.  Then the last match before the
cursor position is used.

In Vi the ":tag" command sets the last search pattern when the tag is searched
for.  In Vim this is not done, the previous search pattern is still remembered,
unless the 't' flag is present in 'cpoptions'.  The search pattern is always
put in the search history.

If the 'wrapscan' option is on (which is the default), searches wrap around
the end of the buffer.  If 'wrapscan' is not set, the backward search stops
at the beginning and the forward search stops at the end of the buffer.  If
'wrapscan' is set and the pattern was not found the error message "pattern
not found" is given, and the cursor will not be moved.  If 'wrapscan' is not
set the message becomes "search hit BOTTOM without match" when searching
forward, or "search hit TOP without match" when searching backward.  If
wrapscan is set and the search wraps around the end of the file the message
"search hit TOP, continuing at BOTTOM" or "search hit BOTTOM, continuing at
TOP" is given when searching backwards or forwards respectively.  This can be
switched off by setting the 's' flag in the 'shortmess' option.  The highlight
method 'w' is used for this message (default: standout).

                                                        *search-range*
You can limit the search command "/" to a certain range of lines by including
\%>l items.  For example, to match the word "limit" below line 199 and above
line 300: >
        /\%>199l\%<300llimit
Also see |/\%>l|.

Another way is to use the ":substitute" command with the 'c' flag.  Example: >
   :.,300s/Pattern//gc
This command will search from the cursor position until line 300 for
"Pattern".  At the match, you will be asked to type a character.  Type 'q' to
stop at this match, type 'n' to find the next match.

The "*", "#", "g*" and "g#" commands look for a word near the cursor in this
order, the first one that is found is used:
- The keyword currently under the cursor.
- The first keyword to the right of the cursor, in the same line.
- The WORD currently under the cursor.
- The first WORD to the right of the cursor, in the same line.
The keyword may only contain letters and characters in 'iskeyword'.
The WORD may contain any non-blanks (<Tab>s and/or <Space>s).
Note that if you type with ten fingers, the characters are easy to remember:
the "#" is under your left hand middle finger (search to the left and up) and
the "*" is under your right hand middle finger (search to the right and down).
(this depends on your keyboard layout though).

==============================================================================
```

2. The definition of a pattern *search-pattern* *pattern* *[pattern]*
regular-expression *regexp* *Pattern*
E76 *E383* *E476*

For starters, read chapter 27 of the user manual |usr_27.txt|.

/bar */\bar* */pattern*
1. A pattern is one or more branches, separated by "\|". It matches anything
that matches one of the branches. Example: "foo\|beep" matches "foo" and
matches "beep". If more than one branch matches, the first one is used.

```
pattern ::=      branch
             or  branch \| branch
             or  branch \| branch \| branch
             etc.
```

/branch */\&*
2. A branch is one or more concats, separated by "\&". It matches the last
concat, but only if all the preceding concats also match at the same
position. Examples:
"foobeep\&..." matches "foo" in "foobeep".
".*Peter\&.*Bob" matches in a line containing both "Peter" and "Bob"

```
branch ::=       concat
             or  concat \& concat
             or  concat \& concat \& concat
             etc.
```

/concat
3. A concat is one or more pieces, concatenated. It matches a match for the
first piece, followed by a match for the second piece, etc. Example:
"f[0-9]b", first matches "f", then a digit and then "b".

```
concat  ::=      piece
             or  piece piece
             or  piece piece piece
             etc.
```

/piece
4. A piece is an atom, possibly followed by a multi, an indication of how many
times the atom can be matched. Example: "a*" matches any sequence of "a"
characters: "", "a", "aa", etc. See |/multi|.

```
piece   ::=      atom
             or  atom  multi
```

/atom
5. An atom can be one of a long list of items. Many atoms match one character
in the text. It is often an ordinary character or a character class.
Braces can be used to make a pattern into an atom. The "\z(\)" construct
is only for syntax highlighting.

```
atom    ::=      ordinary-atom              |/ordinary-atom|
             or  \( pattern \)              |/\(|
             or  \%( pattern \)             |/\%(|
             or  \z( pattern \)             |/\z(|
```

/\%#= *two-engines* *NFA*
Vim includes two regexp engines:
1. An old, backtracking engine that supports everything.
2. A new, NFA engine that works much faster on some patterns, possibly slower

on some patterns.

Vim will automatically select the right engine for you. However, if you run into a problem or want to specifically select one engine or the other, you can prepend one of the following to the pattern:

```
\%#=0   Force automatic selection.  Only has an effect when
        'regexpengine' has been set to a non-zero value.
\%#=1   Force using the old engine.
\%#=2   Force using the NFA engine.
```

You can also use the 'regexpengine' option to change the default.

E864 *E868* *E874* *E875* *E876* *E877* *E878*
If selecting the NFA engine and it runs into something that is not implemented the pattern will not match. This is only useful when debugging Vim.

==========
3. Magic */magic*

Some characters in the pattern are taken literally. They match with the same character in the text. When preceded with a backslash however, these characters get a special meaning.

Other characters have a special meaning without a backslash. They need to be preceded with a backslash to match literally.

If a character is taken literally or not depends on the 'magic' option and the items mentioned next.

/\m */\M*
Use of "\m" makes the pattern after it be interpreted as if 'magic' is set, ignoring the actual value of the 'magic' option.
Use of "\M" makes the pattern after it be interpreted as if 'nomagic' is used.

/\v */\V*
Use of "\v" means that in the pattern after it all ASCII characters except '0'-'9', 'a'-'z', 'A'-'Z' and '_' have a special meaning. "very magic"

Use of "\V" means that in the pattern after it only the backslash and the terminating character (/ or ?) has a special meaning. "very nomagic"

Examples:

after:	\v	\m	\M	\V	matches ~
		'magic'	'nomagic'		
	$	$	$	\$	matches end-of-line
	.	.	\.	\.	matches any character
	*	*	*	*	any number of the previous atom
	~	~	\~	\~	latest substitute string
	()	\(\)	\(\)	\(\)	grouping into an atom
	|	|	|	|	separating alternatives
	\a	\a	\a	\a	alphabetic character
	\\	\\	\\	\\	literal backslash
	\.	\.	.	.	literal dot
	\{	{	{	{	literal '{'
	a	a	a	a	literal 'a'

{only Vim supports \m, \M, \v and \V}

It is recommended to always keep the 'magic' option at the default setting, which is 'magic'. This avoids portability problems. To make a pattern immune to the 'magic' option being set or not, put "\m" or "\M" at the start of the pattern.

```
==============================================================================
4. Overview of pattern items                                *pattern-overview*
                                                  *E865* *E866* *E867* *E869*

Overview of multi items.                                    */multi* *E61* *E62*
More explanation and examples below, follow the links.          *E64* *E871*

          multi ~
     'magic' 'nomagic'  matches of the preceding atom ~
|/star| *       \*      0 or more       as many as possible
|/\+|   \+      \+      1 or more       as many as possible (*)
|/\=|   \=      \=      0 or 1          as many as possible (*)
|/\?|   \?      \?      0 or 1          as many as possible (*)

|/\{|   \{n,m}  \{n,m}  n to m          as many as possible (*)
        \{n}    \{n}    n               exactly (*)
        \{n,}   \{n,}   at least n      as many as possible (*)
        \{,m}   \{,m}   0 to m          as many as possible (*)
        \{}     \{}     0 or more       as many as possible (same as *) (*)

|/\{-|  \{-n,m} \{-n,m} n to m          as few as possible (*)
        \{-n}   \{-n}   n               exactly (*)
        \{-n,}  \{-n,}  at least n      as few as possible (*)
        \{-,m}  \{-,m}  0 to m          as few as possible (*)
        \{-}    \{-}    0 or more       as few as possible (*)

                                                        *E59*
|/\@>|  \@>     \@>     1, like matching a whole pattern (*)
|/\@=|  \@=     \@=     nothing, requires a match |/zero-width| (*)
|/\@!|  \@!     \@!     nothing, requires NO match |/zero-width| (*)
|/\@<=| \@<=    \@<=    nothing, requires a match behind |/zero-width| (*)
|/\@<!| \@<!    \@<!    nothing, requires NO match behind |/zero-width| (*)

(*) {not in Vi}


Overview of ordinary atoms.                                 */ordinary-atom*
More explanation and examples below, follow the links.

      ordinary atom ~
      magic   nomagic   matches ~
|/^|    ^       ^       start-of-line (at start of pattern) |/zero-width|
|/\^|   \^      \^      literal '^'
|/\_^|  \_^     \_^     start-of-line (used anywhere) |/zero-width|
|/$|    $       $       end-of-line (at end of pattern) |/zero-width|
|/\$|   \$      \$      literal '$'
|/\_$|  \_$     \_$     end-of-line (used anywhere) |/zero-width|
|/.|    .       \.      any single character (not an end-of-line)
|/\_.|  \_.     \_.     any single character or end-of-line
|/\<|   \<      \<      beginning of a word |/zero-width|
|/\>|   \>      \>      end of a word |/zero-width|
|/\zs|  \zs     \zs     anything, sets start of match
|/\ze|  \ze     \ze     anything, sets end of match
|/\%^|  \%^     \%^     beginning of file |/zero-width|         *E71*
|/\%$|  \%$     \%$     end of file |/zero-width|
|/\%V|  \%V     \%V     inside Visual area |/zero-width|
|/\%#|  \%#     \%#     cursor position |/zero-width|
|/\%'m| \%'m    \%'m    mark m position |/zero-width|
|/\%l|  \%23l   \%23l   in line 23 |/zero-width|
|/\%c|  \%23c   \%23c   in column 23 |/zero-width|
|/\%v|  \%23v   \%23v   in virtual column 23 |/zero-width|
```

```
Character classes {not in Vi}:                          */character-classes*
      magic   nomagic   matches ~
|/\i|   \i      \i      identifier character (see 'isident' option)
|/\I|   \I      \I      like "\i", but excluding digits
|/\k|   \k      \k      keyword character (see 'iskeyword' option)
|/\K|   \K      \K      like "\k", but excluding digits
|/\f|   \f      \f      file name character (see 'isfname' option)
|/\F|   \F      \F      like "\f", but excluding digits
|/\p|   \p      \p      printable character (see 'isprint' option)
|/\P|   \P      \P      like "\p", but excluding digits
|/\s|   \s      \s      whitespace character: <Space> and <Tab>
|/\S|   \S      \S      non-whitespace character; opposite of \s
|/\d|   \d      \d      digit:                          [0-9]
|/\D|   \D      \D      non-digit:                      [^0-9]
|/\x|   \x      \x      hex digit:                      [0-9A-Fa-f]
|/\X|   \X      \X      non-hex digit:                  [^0-9A-Fa-f]
|/\o|   \o      \o      octal digit:                    [0-7]
|/\O|   \O      \O      non-octal digit:                [^0-7]
|/\w|   \w      \w      word character:                 [0-9A-Za-z_]
|/\W|   \W      \W      non-word character:             [^0-9A-Za-z_]
|/\h|   \h      \h      head of word character:         [A-Za-z_]
|/\H|   \H      \H      non-head of word character:     [^A-Za-z_]
|/\a|   \a      \a      alphabetic character:           [A-Za-z]
|/\A|   \A      \A      non-alphabetic character:       [^A-Za-z]
|/\l|   \l      \l      lowercase character:            [a-z]
|/\L|   \L      \L      non-lowercase character:        [^a-z]
|/\u|   \u      \u      uppercase character:            [A-Z]
|/\U|   \U      \U      non-uppercase character         [^A-Z]
|/\_|   \_x     \_x     where x is any of the characters above: character
                        class with end-of-line included
(end of character classes)

      magic   nomagic   matches ~
|/\e|   \e      \e      <Esc>
|/\t|   \t      \t      <Tab>
|/\r|   \r      \r      <CR>
|/\b|   \b      \b      <BS>
|/\n|   \n      \n      end-of-line
|/~|    ~       \~      last given substitute string
|/\1|   \1      \1      same string as matched by first \(\) {not in Vi}
|/\2|   \2      \2      Like "\1", but uses second \(\)
          ...
|/\9|   \9      \9      Like "\1", but uses ninth \(\)
                                                              *E68*
|/\z1|  \z1     \z1     only for syntax highlighting, see |:syn-ext-match|
          ...
|/\z1|  \z9     \z9     only for syntax highlighting, see |:syn-ext-match|

        x       x       a character with no special meaning matches itself

|/[]|   []      \[]     any character specified inside the []
|/\%[]| \%[]    \%[]    a sequence of optionally matched atoms

|/\c|   \c      \c      ignore case, do not use the 'ignorecase' option
|/\C|   \C      \C      match case, do not use the 'ignorecase' option
|/\Z|   \Z      \Z      ignore differences in Unicode "combining characters".
                        Useful when searching voweled Hebrew or Arabic text.

      magic   nomagic   matches ~
|/\m|   \m      \m      'magic' on for the following chars in the pattern
|/\M|   \M      \M      'magic' off for the following chars in the pattern
|/\v|   \v      \v      the following chars in the pattern are "very magic"
```

```
|/\V|   \V      \V      the following chars in the pattern are "very nomagic"
|/\%#=|   \%#=1   \%#=1   select regexp engine |/zero-width|

|/\%d|  \%d     \%d     match specified decimal character (eg \%d123)
|/\%x|  \%x     \%x     match specified hex character (eg \%x2a)
|/\%o|  \%o     \%o     match specified octal character (eg \%o040)
|/\%u|  \%u     \%u     match specified multibyte character (eg \%u20ac)
|/\%U|  \%U     \%U     match specified large multibyte character (eg
                       \%U12345678)
|/\%C|  \%C     \%C     match any composing characters

Example                matches ~
\<\I\i*         or
\<\h\w*
\<[a-zA-Z_][a-zA-Z0-9_]*
                       An identifier (e.g., in a C program).

\(\.$\|\. \)           A period followed by <EOL> or a space.

[.!?][])"']*\($\|[ ]\)  A search pattern that finds the end of a sentence,
                       with almost the same definition as the ")" command.

cat\Z                  Both "cat" and "càt" ("a" followed by 0x0300)
                       Does not match "càt" (character 0x00e0), even
                       though it may look the same.


==============================================================================
5. Multi items                                          *pattern-multi-items*

An atom can be followed by an indication of how many times the atom can be
matched and in what way.  This is called a multi.  See |/multi| for an
overview.

                                                        */star* */\star*
*       (use \* when 'magic' is not set)
        Matches 0 or more of the preceding atom, as many as possible.
        Example  'nomagic'      matches ~
        a*         a\*          "", "a", "aa", "aaa", etc.
        .*         \.\*         anything, also an empty string, no end-of-line
        \_.*       \_.\*        everything up to the end of the buffer
        \_.*END    \_.\*END     everything up to and including the last "END"
                                in the buffer

        Exception: When "*" is used at the start of the pattern or just after
        "^" it matches the star character.

        Be aware that repeating "\_." can match a lot of text and take a long
        time.  For example, "\_.*END" matches all text from the current
        position to the last occurrence of "END" in the file.  Since the "*"
        will match as many as possible, this first skips over all lines until
        the end of the file and then tries matching "END", backing up one
        character at a time.

                                                        */\+*
\+      Matches 1 or more of the preceding atom, as many as possible. {not in
        Vi}
        Example         matches ~
        ^.\+$           any non-empty line
        \s\+            white space of at least one character

                                                        */\=*
```

```
\=      Matches 0 or 1 of the preceding atom, as many as possible. {not in Vi}
        Example         matches ~
        foo\=           "fo" and "foo"

                                                              */\?*
\?      Just like \=.  Cannot be used when searching backwards with the "?"
        command. {not in Vi}

                                          */\{* *E60* *E554* *E870*
\{n,m}  Matches n to m of the preceding atom, as many as possible
\{n}    Matches n of the preceding atom
\{n,}   Matches at least n of the preceding atom, as many as possible
\{,m}   Matches 0 to m of the preceding atom, as many as possible
\{}     Matches 0 or more of the preceding atom, as many as possible (like *)
                                                              */\{-*
\{-n,m} matches n to m of the preceding atom, as few as possible
\{-n}   matches n of the preceding atom
\{-n,}  matches at least n of the preceding atom, as few as possible
\{-,m}  matches 0 to m of the preceding atom, as few as possible
\{-}    matches 0 or more of the preceding atom, as few as possible
        {Vi does not have any of these}

        n and m are positive decimal numbers or zero
                                                                  *non-greedy*
        If a "-" appears immediately after the "{", then a shortest match
        first algorithm is used (see example below).  In particular, "\{-}" is
        the same as "*" but uses the shortest match first algorithm.  BUT: A
        match that starts earlier is preferred over a shorter match: "a\{-}b"
        matches "aaab" in "xaaab".

        Example                 matches ~
        ab\{2,3}c               "abbc" or "abbbc"
        a\{5}                   "aaaaa"
        ab\{2,}c                "abbc", "abbbc", "abbbbc", etc.
        ab\{,3}c                "ac", "abc", "abbc" or "abbbc"
        a[bc]\{3}d              "abbbd", "abbcd", "acbcd", "acccd", etc.
        a\(bc\)\{1,2}d          "abcd" or "abcbcd"
        a[bc]\{-}[cd]           "abc" in "abcd"
        a[bc]*[cd]              "abcd" in "abcd"

        The } may optionally be preceded with a backslash: \{n,m\}.

                                                              */\@=*
\@=     Matches the preceding atom with zero width. {not in Vi}
        Like "(?=pattern)" in Perl.
        Example                 matches ~
        foo\(bar\)\@=           "foo" in "foobar"
        foo\(bar\)\@=foo        nothing
                                                              */zero-width*
        When using "\@=" (or "^", "$", "\<", "\>") no characters are included
        in the match.  These items are only used to check if a match can be
        made.  This can be tricky, because a match with following items will
        be done in the same position.  The last example above will not match
        "foobarfoo", because it tries match "foo" in the same position where
        "bar" matched.

        Note that using "\&" works the same as using "\@=": "foo\&.." is the
        same as "\(foo\)\@=..".  But using "\&" is easier, you don't need the
        braces.


                                                              */\@!*
```

```
\@!     Matches with zero width if the preceding atom does NOT match at the
        current position. |/zero-width| {not in Vi}
        Like "(?!pattern)" in Perl.
        Example                 matches ~
        foo\(bar\)\@!           any "foo" not followed by "bar"
        a.\{-}p\@!              "a", "ap", "app", "appp", etc. not immediately
                                followed by a "p"
        if \(\(then\)\@!.\)*$   "if " not followed by "then"

        Using "\@!" is tricky, because there are many places where a pattern
        does not match.  "a.*p\@!" will match from an "a" to the end of the
        line, because ".*" can match all characters in the line and the "p"
        doesn't match at the end of the line.  "a.\{-}p\@!" will match any
        "a", "ap", "app", etc. that isn't followed by a "p", because the "."
        can match a "p" and "p\@!" doesn't match after that.

        You can't use "\@!" to look for a non-match before the matching
        position: "\(foo\)\@!bar" will match "bar" in "foobar", because at the
        position where "bar" matches, "foo" does not match.  To avoid matching
        "foobar" you could use "\(foo\)\@!...bar", but that doesn't match a
        bar at the start of a line.  Use "\(foo\)\@<!bar".

        Useful example: to find "foo" in a line that does not contain "bar": >
                /^\%(.*bar\)\@!.*\zsfoo
<       This pattern first checks that there is not a single position in the
        line where "bar" matches.  If ".*bar" matches somewhere the \@! will
        reject the pattern.  When there is no match any "foo" will be found.
        The "\zs" is to have the match start just before "foo".

                                                        */\@<=*
\@<=    Matches with zero width if the preceding atom matches just before what
        follows. |/zero-width| {not in Vi}
        Like "(?<=pattern)" in Perl, but Vim allows non-fixed-width patterns.
        Example                 matches ~
        \(an\_s\+\)\@<=file     "file" after "an" and white space or an
                                end-of-line
        For speed it's often much better to avoid this multi.  Try using "\zs"
        instead |/\zs|.  To match the same as the above example:
                an\_s\+\zsfile
        At least set a limit for the look-behind, see below.

        "\@<=" and "\@<!" check for matches just before what follows.
        Theoretically these matches could start anywhere before this position.
        But to limit the time needed, only the line where what follows matches
        is searched, and one line before that (if there is one).  This should
        be sufficient to match most things and not be too slow.

        In the old regexp engine the part of the pattern after "\@<=" and
        "\@<!" are checked for a match first, thus things like "\1" don't work
        to reference \(\) inside the preceding atom.  It does work the other
        way around:
        Bad example                     matches ~
        \%#=1\1\@<=,\([a-z]\+\)         ",abc" in "abc,abc"

        However, the new regexp engine works differently, it is better to not
        rely on this behavior, do not use \@<= if it can be avoided:
        Example                         matches ~
        \([a-z]\+\)\zs,\1               ",abc" in "abc,abc"

\@123<=
        Like "\@<=" but only look back 123 bytes. This avoids trying lots
        of matches that are known to fail and make executing the pattern very
```

slow. Example, check if there is a "<" just before "span":
/<\@1<=span
This will try matching "<" only one byte before "span", which is the
only place that works anyway.
After crossing a line boundary, the limit is relative to the end of
the line. Thus the characters at the start of the line with the match
are not counted (this is just to keep it simple).
The number zero is the same as no limit.

/\@<!
\@<! Matches with zero width if the preceding atom does NOT match just
before what follows. Thus this matches if there is no position in the
current or previous line where the atom matches such that it ends just
before what follows. |/zero-width| {not in Vi}
Like "(?<!pattern)" in Perl, but Vim allows non-fixed-width patterns.
The match with the preceding atom is made to end just before the match
with what follows, thus an atom that ends in ".*" will work.
Warning: This can be slow (because many positions need to be checked
for a match). Use a limit if you can, see below.
Example matches ~
\(foo\)\@<!bar any "bar" that's not in "foobar"
\(\/\/.*\)\@<!in "in" which is not after "//"

\@123<!
Like "\@<!" but only look back 123 bytes. This avoids trying lots of
matches that are known to fail and make executing the pattern very
slow.

/\@>
\@> Matches the preceding atom like matching a whole pattern. {not in Vi}
Like "(?>pattern)" in Perl.
Example matches ~
\(a*\)\@>a nothing (the "a*" takes all the "a"'s, there can't be
another one following)

This matches the preceding atom as if it was a pattern by itself. If
it doesn't match, there is no retry with shorter sub-matches or
anything. Observe this difference: "a*b" and "a*ab" both match
"aaab", but in the second case the "a*" matches only the first two
"a"s. "\(a*\)\@>ab" will not match "aaab", because the "a*" matches
the "aaa" (as many "a"s as possible), thus the "ab" can't match.

==
6. Ordinary atoms *pattern-atoms*

An ordinary atom can be:

*/^*
^ At beginning of pattern or after "\|", "\(", "\%(" or "\n": matches
start-of-line; at other positions, matches literal '^'. |/zero-width|
Example matches ~
^beep(the start of the C function "beep" (probably).

*/\^*
\^ Matches literal '^'. Can be used at any position in the pattern.

*/_^*
_^ Matches start-of-line. |/zero-width| Can be used at any position in
the pattern.
Example matches ~
_s*_^foo white space and blank lines and then "foo" at

```
                        start-of-line

                                                              */$*
$       At end of pattern or in front of "\|", "\)" or "\n" ('magic' on):
        matches end-of-line <EOL>; at other positions, matches literal '$'.
        |/zero-width|

                                                              */\$*
\$      Matches literal '$'.  Can be used at any position in the pattern.

                                                              */\_$*
\_$     Matches end-of-line. |/zero-width|  Can be used at any position in the
        pattern.  Note that "a\_$b" never matches, since "b" cannot match an
        end-of-line.  Use "a\nb" instead |/\n|.
        Example         matches ~
        foo\_$\_s*      "foo" at end-of-line and following white space and
                        blank lines

.       (with 'nomagic': \.)                                  */.* */\.*
        Matches any single character, but not an end-of-line.

                                                              */\_.*
\_.     Matches any single character or end-of-line.
        Careful: "\_.*" matches all text to the end of the buffer!

                                                              */\<*
\<      Matches the beginning of a word: The next char is the first char of a
        word.  The 'iskeyword' option specifies what is a word character.
        |/zero-width|

                                                              */\>*
\>      Matches the end of a word: The previous char is the last char of a
        word.  The 'iskeyword' option specifies what is a word character.
        |/zero-width|

                                                              */\zs*
\zs     Matches at any position, and sets the start of the match there: The
        next char is the first char of the whole match. |/zero-width|
        Example: >
                /^\s*\zsif
<       matches an "if" at the start of a line, ignoring white space.
        Can be used multiple times, the last one encountered in a matching
        branch is used.  Example: >
                /\(.\{-}\zsFab\)\{3}
<       Finds the third occurrence of "Fab".
        This cannot be followed by a multi. *E888*
        {not in Vi} {not available when compiled without the |+syntax| feature}
                                                              */\ze*
\ze     Matches at any position, and sets the end of the match there: The
        previous char is the last char of the whole match. |/zero-width|
        Can be used multiple times, the last one encountered in a matching
        branch is used.
        Example: "end\ze\(if\|for\)" matches the "end" in "endif" and
        "endfor".
        This cannot be followed by a multi. |E888|
        {not in Vi} {not available when compiled without the |+syntax| feature}

                                                  */\%^* *start-of-file*
\%^     Matches start of the file.  When matching with a string, matches the
        start of the string. {not in Vi}
        For example, to find the first "VIM" in a file: >
                /\%^\_.\{-}\zsVIM
```

```
<
                                                *\%$* *end-of-file*
\%$     Matches end of the file.  When matching with a string, matches the
        end of the string. {not in Vi}
        Note that this does NOT find the last "VIM" in a file: >
                /VIM\_.\{-}\%$
<       It will find the next VIM, because the part after it will always
        match.  This one will find the last "VIM" in the file: >
                /VIM\ze\(\(VIM\)\@!\_.\)*\%$
<       This uses |/\@!| to ascertain that "VIM" does NOT match in any
        position after the first "VIM".
        Searching from the end of the file backwards is easier!

                                                *\%V*
\%V     Match inside the Visual area.  When Visual mode has already been
        stopped match in the area that |gv| would reselect.
        This is a |/zero-width| match.  To make sure the whole pattern is
        inside the Visual area put it at the start and just before the end of
        the pattern, e.g.: >
                /\%Vfoo.*ba\%Vr
<       This also works if only "foo bar" was Visually selected. This: >
                /\%Vfoo.*bar\%V
<       would match "foo bar" if the Visual selection continues after the "r".
        Only works for the current buffer.

                                                *\%#* *cursor-position*
\%#     Matches with the cursor position.  Only works when matching in a
        buffer displayed in a window. {not in Vi}
        WARNING: When the cursor is moved after the pattern was used, the
        result becomes invalid.  Vim doesn't automatically update the matches.
        This is especially relevant for syntax highlighting and 'hlsearch'.
        In other words: When the cursor moves the display isn't updated for
        this change.  An update is done for lines which are changed (the whole
        line is updated) or when using the |CTRL-L| command (the whole screen
        is updated).  Example, to highlight the word under the cursor: >
                /\k*\%#\k*
<       When 'hlsearch' is set and you move the cursor around and make changes
        this will clearly show when the match is updated or not.

                                                *\%'m* *\%<'m* *\%>'m*
\%'m    Matches with the position of mark m.
\%<'m   Matches before the position of mark m.
\%>'m   Matches after the position of mark m.
        Example, to highlight the text from mark 's to 'e: >
                /.\%>'s.*\%<'e..
<       Note that two dots are required to include mark 'e in the match.  That
        is because "\%<'e" matches at the character before the 'e mark, and
        since it's a |/zero-width| match it doesn't include that character.
        {not in Vi}
        WARNING: When the mark is moved after the pattern was used, the result
        becomes invalid.  Vim doesn't automatically update the matches.
        Similar to moving the cursor for "\%#" |/\%#|.

                                                *\%l* *\%>l* *\%<l*
\%23l   Matches in a specific line.
\%<23l  Matches above a specific line (lower line number).
\%>23l  Matches below a specific line (higher line number).
        These three can be used to match specific lines in a buffer.  The "23"
        can be any line number.  The first line is 1. {not in Vi}
        WARNING: When inserting or deleting lines Vim does not automatically
        update the matches.  This means Syntax highlighting quickly becomes
        wrong.
```

```
        Example, to highlight the line where the cursor currently is: >
                :exe '/\%' . line(".") . 'l.*'
<       When 'hlsearch' is set and you move the cursor around and make changes
        this will clearly show when the match is updated or not.

                                                */\%c* */\%>c* */\%<c*
\%23c   Matches in a specific column.
\%<23c  Matches before a specific column.
\%>23c  Matches after a specific column.
        These three can be used to match specific columns in a buffer or
        string.  The "23" can be any column number.  The first column is 1.
        Actually, the column is the byte number (thus it's not exactly right
        for multi-byte characters).  {not in Vi}
        WARNING: When inserting or deleting text Vim does not automatically
        update the matches.  This means Syntax highlighting quickly becomes
        wrong.
        Example, to highlight the column where the cursor currently is: >
                :exe '/\%' . col(".") . 'c'
<       When 'hlsearch' is set and you move the cursor around and make changes
        this will clearly show when the match is updated or not.
        Example for matching a single byte in column 44: >
                /\%>43c.\%<46c
<       Note that "\%<46c" matches in column 45 when the "." matches a byte in
        column 44.
                                                */\%v* */\%>v* */\%<v*
\%23v   Matches in a specific virtual column.
\%<23v  Matches before a specific virtual column.
\%>23v  Matches after a specific virtual column.
        These three can be used to match specific virtual columns in a buffer
        or string.  When not matching with a buffer in a window, the option
        values of the current window are used (e.g., 'tabstop').
        The "23" can be any column number.  The first column is 1.
        Note that some virtual column positions will never match, because they
        are halfway through a tab or other character that occupies more than
        one screen character.  {not in Vi}
        WARNING: When inserting or deleting text Vim does not automatically
        update highlighted matches.  This means Syntax highlighting quickly
        becomes wrong.
        Example, to highlight all the characters after virtual column 72: >
                /\%>72v.*
<       When 'hlsearch' is set and you move the cursor around and make changes
        this will clearly show when the match is updated or not.
        To match the text up to column 17: >
                /^.*\%17v
<       Column 17 is not included, because this is a |/zero-width| match. To
        include the column use: >
                /^.*\%17v.
<       This command does the same thing, but also matches when there is no
        character in column 17: >
                /^.*\%<18v.
<       Note that without the "^" to anchor the match in the first column,
        this will also highlight column 17: >
                /.*\%17v
<       Column 17 is highlighted by 'hlsearch' because there is another match
        where ".*" matches zero characters.
<

Character classes: {not in Vi}
\i      identifier character (see 'isident' option)     */\i*
\I      like "\i", but excluding digits                 */\I*
\k      keyword character (see 'iskeyword' option)      */\k*
\K      like "\k", but excluding digits                 */\K*
```

```
\f      file name character (see 'isfname' option)      */\f*
\F      like "\f", but excluding digits                 */\F*
\p      printable character (see 'isprint' option)      */\p*
\P      like "\p", but excluding digits                 */\P*

NOTE: the above also work for multi-byte characters.  The ones below only
match ASCII characters, as indicated by the range.

                                                *whitespace* *white-space*
\s      whitespace character: <Space> and <Tab>         */\s*
\S      non-whitespace character; opposite of \s        */\S*
\d      digit:                          [0-9]           */\d*
\D      non-digit:                      [^0-9]          */\D*
\x      hex digit:                      [0-9A-Fa-f]     */\x*
\X      non-hex digit:                  [^0-9A-Fa-f]    */\X*
\o      octal digit:                    [0-7]           */\o*
\O      non-octal digit:                [^0-7]          */\O*
\w      word character:                 [0-9A-Za-z_]    */\w*
\W      non-word character:             [^0-9A-Za-z_]   */\W*
\h      head of word character:         [A-Za-z_]       */\h*
\H      non-head of word character:     [^A-Za-z_]      */\H*
\a      alphabetic character:           [A-Za-z]        */\a*
\A      non-alphabetic character:       [^A-Za-z]       */\A*
\l      lowercase character:            [a-z]           */\l*
\L      non-lowercase character:        [^a-z]          */\L*
\u      uppercase character:            [A-Z]           */\u*
\U      non-uppercase character:        [^A-Z]          */\U*

        NOTE: Using the atom is faster than the [] form.

        NOTE: 'ignorecase', "\c" and "\C" are not used by character classes.

                        */\_* *E63* */\_i* */\_I* */\_k* */\_K* */\_f* */\_F*
                        */\_p* */\_P* */\_s* */\_S* */\_d* */\_D* */\_x* */\_X*
                        */\_o* */\_O* */\_w* */\_W* */\_h* */\_H* */\_a* */\_A*
                        */\_l* */\_L* */\_u* */\_U*
\_x     Where "x" is any of the characters above: The character class with
        end-of-line added
(end of character classes)

\e      matches <Esc>                                   */\e*
\t      matches <Tab>                                   */\t*
\r      matches <CR>                                    */\r*
\b      matches <BS>                                    */\b*
\n      matches an end-of-line                          */\n*
        When matching in a string instead of buffer text a literal newline
        character is matched.

~       matches the last given substitute string        */~* */\~*

\(\)    A pattern enclosed by escaped parentheses.      */\(* */\(\)* */\)*
        E.g., "\(^a\)" matches 'a' at the start of a line.
        *E51* *E54* *E55* *E872* *E873*

\1      Matches the same string that was matched by     */\1* *E65*
        the first sub-expression in \( and \). {not in Vi}
        Example: "\([a-z]\).\1" matches "ata", "ehe", "tot", etc.
\2      Like "\1", but uses second sub-expression,      */\2*
   ...                                                  */\3*
\9      Like "\1", but uses ninth sub-expression.       */\9*
        Note: The numbering of groups is done based on which "\(" comes first
        in the pattern (going left to right), NOT based on what is matched
```

first.

\%(\) A pattern enclosed by escaped parentheses. */\%(\)* */\%(* *E53*
Just like \(\), but without counting it as a sub-expression. This
allows using more groups and it's a little bit faster.
{not in Vi}

x A single character, with no special meaning, matches itself

/ */*
\x A backslash followed by a single character, with no special meaning,
is reserved for future expansions

[] (with 'nomagic': \[]) */[]* */\[]* */_[]* */collection*
_[]
A collection. This is a sequence of characters enclosed in brackets.
It matches any single character in the collection.

Example	matches ~
[xyz]	any 'x', 'y' or 'z'
[a-zA-Z]$	any alphabetic character at the end of a line
\c[a-z]$	same
[A-яЁё]	Russian alphabet (with utf-8 and cp1251)

/[\n]
With "_" prepended the collection also includes the end-of-line.
The same can be done by including "\n" in the collection. The
end-of-line is also matched when the collection starts with "^"! Thus
"_[^ab]" matches the end-of-line and any character but "a" and "b".
This makes it Vi compatible: Without the "_" or "\n" the collection
does not match an end-of-line.
E769
When the ']' is not there Vim will not give an error message but
assume no collection is used. Useful to search for '['. However, you
do get E769 for internal searching. And be aware that in a
`:substitute` command the whole command becomes the pattern. E.g.
":s/[/x/" searches for "[/x" and replaces it with nothing. It does
not search for "[" and replaces it with "x"!

E944 *E945*
If the sequence begins with "^", it matches any single character NOT
in the collection: "[^xyz]" matches anything but 'x', 'y' and 'z'.
- If two characters in the sequence are separated by '-', this is
shorthand for the full list of ASCII characters between them. E.g.,
"[0-9]" matches any decimal digit. If the starting character exceeds
the ending character, e.g. [c-a], E944 occurs. Non-ASCII characters
can be used, but the character values must not be more than 256 apart
in the old regexp engine. For example, searching by [\u3000-\u4000]
after setting re=1 emits a E945 error. Prepending \%#=2 will fix it.
- A character class expression is evaluated to the set of characters
belonging to that character class. The following character classes
are supported:

	Name	Func	Contents ~
[:alnum:]	[:alnum:]	isalnum	ASCII letters and digits
[:alpha:]	[:alpha:]	isalpha	ASCII letters
[:blank:]	[:blank:]		space and tab
[:cntrl:]	[:cntrl:]	iscntrl	ASCII control characters
[:digit:]	[:digit:]		decimal digits '0' to '9'
[:graph:]	[:graph:]	isgraph	ASCII printable characters excluding space
[:lower:]	[:lower:]	(1)	lowercase letters (all letters when 'ignorecase' is used)
[:print:]	[:print:]	(2)	printable characters including space

```
*[:punct:]*       [:punct:]   ispunct   ASCII punctuation characters
*[:space:]*       [:space:]             whitespace characters: space, tab, CR,
                                        NL, vertical tab, form feed
*[:upper:]*       [:upper:]   (3)       uppercase letters (all letters when
                                        'ignorecase' is used)
*[:xdigit:]*      [:xdigit:]            hexadecimal digits: 0-9, a-f, A-F
*[:return:]*      [:return:]            the <CR> character
*[:tab:]*         [:tab:]               the <Tab> character
*[:escape:]*      [:escape:]            the <Esc> character
*[:backspace:]*   [:backspace:]         the <BS> character
          The brackets in character class expressions are additional to the
          brackets delimiting a collection.  For example, the following is a
          plausible pattern for a UNIX filename: "[-./[:alnum:]_~]\+" That is,
          a list of at least one character, each of which is either '-', '.',
          '/', alphabetic, numeric, '_' or '~'.
          These items only work for 8-bit characters, except [:lower:] and
          [:upper:] also work for multi-byte characters when using the new
          regexp engine.  See |two-engines|.  In the future these items may
          work for multi-byte characters.  For now, to get all "alpha"
          characters you can use: [[:lower:][:upper:]].

          The "Func" column shows what library function is used.  The
          implementation depends on the system.  Otherwise:
          (1) Uses islower() for ASCII and Vim builtin rules for other
          characters when built with the |+multi_byte| feature.
          (2) Uses Vim builtin rules
          (3) As with (1) but using isupper()
                                                          */[[=* *[==]*
        - An equivalence class.  This means that characters are matched that
          have almost the same meaning, e.g., when ignoring accents.  This
          only works for Unicode, latin1 and latin9.  The form is:
                [=a=]
                                                          */[[.* *[..]*
        - A collation element.  This currently simply accepts a single
          character in the form:
                [.a.]
                                                            */\]*
        - To include a literal ']', '^', '-' or '\' in the collection, put a
          backslash before it: "[xyz\]]", "[\^xyz]", "[xy\-z]" and "[xyz\\]".
          (Note: POSIX does not support the use of a backslash this way).  For
          ']' you can also make it the first character (following a possible
          "^"):  "[]xyz]" or "[^]xyz]" {not in Vi}.
          For '-' you can also make it the first or last character: "[-xyz]",
          "[^-xyz]" or "[xyz-]".  For '\' you can also let it be followed by
          any character that's not in "^]-\bdertnoUux".  "[\xyz]" matches '\',
          'x', 'y' and 'z'.  It's better to use "\\" though, future expansions
          may use other characters after '\'.
        - Omitting the trailing ] is not considered an error. "[]" works like
          "[]]", it matches the ']' character.
        - The following translations are accepted when the 'l' flag is not
          included in 'cpoptions' {not in Vi}:
                \e      <Esc>
                \t      <Tab>
                \r      <CR>    (NOT end-of-line!)
                \b      <BS>
                \n      line break, see above |/[\n]|
                \d123   decimal number of character
                \o40    octal number of character up to 0377
                \x20    hexadecimal number of character up to 0xff
                \u20AC  hex. number of multibyte character up to 0xffff
                \U1234  hex. number of multibyte character up to 0xffffffff
          NOTE: The other backslash codes mentioned above do not work inside
```

```
          []!
        - Matching with a collection can be slow, because each character in
          the text has to be compared with each character in the collection.
          Use one of the other atoms above when possible.  Example: "\d" is
          much faster than "[0-9]" and matches the same characters.

                                                */\%[]* *E69* *E70* *E369*
\%[]    A sequence of optionally matched atoms.  This always matches.
        It matches as much of the list of atoms it contains as possible.  Thus
        it stops at the first atom that doesn't match.  For example: >
                /r\%[ead]
<       matches "r", "re", "rea" or "read".  The longest that matches is used.
        To match the Ex command "function", where "fu" is required and
        "nction" is optional, this would work: >
                /\<fu\%[nction]\>
<       The end-of-word atom "\>" is used to avoid matching "fu" in "full".
        It gets more complicated when the atoms are not ordinary characters.
        You don't often have to use it, but it is possible.  Example: >
                /\<r\%[[eo]ad]\>
<       Matches the words "r", "re", "ro", "rea", "roa", "read" and "road".
        There can be no \(\), \%(\) or \z(\) items inside the [] and \%[] does
        not nest.
        To include a "[" use "[[]" and for "]" use []]", e.g.,: >
                /index\%[[[]0[]]]
<       matches "index" "index[", "index[0" and "index[0]".
        {not available when compiled without the |+syntax| feature}

                                */\%d* */\%x* */\%o* */\%u* */\%U* *E678*

\%d123  Matches the character specified with a decimal number.  Must be
        followed by a non-digit.
\%o40   Matches the character specified with an octal number up to 0377.
        Numbers below 040 must be followed by a non-octal digit or a non-digit.
\%x2a   Matches the character specified with up to two hexadecimal characters.
\%u20AC Matches the character specified with up to four hexadecimal
        characters.
\%U1234abcd     Matches the character specified with up to eight hexadecimal
        characters.

==============================================================================
7. Ignoring case in a pattern                                   */ignorecase*

If the 'ignorecase' option is on, the case of normal letters is ignored.
'smartcase' can be set to ignore case when the pattern contains lowercase
letters only.
                                                            */\c* */\C*
When "\c" appears anywhere in the pattern, the whole pattern is handled like
'ignorecase' is on.  The actual value of 'ignorecase' and 'smartcase' is
ignored.  "\C" does the opposite: Force matching case for the whole pattern.
{only Vim supports \c and \C}
Note that 'ignorecase', "\c" and "\C" are not used for the character classes.

Examples:
      pattern   'ignorecase'  'smartcase'       matches ~
        foo       off           -               foo
        foo       on            -               foo Foo FOO
        Foo       on            off             foo Foo FOO
        Foo       on            on                  Foo
        \cfoo     -             -               foo Foo FOO
        foo\C     -             -               foo

Technical detail:                               *NL-used-for-Nul*
```

<Nul> characters in the file are stored as <NL> in memory. In the display they are shown as "^@". The translation is done when reading and writing files. To match a <Nul> with a search pattern you can just enter CTRL-@ or "CTRL-V 000". This is probably just what you expect. Internally the character is replaced with a <NL> in the search pattern. What is unusual is that typing CTRL-V CTRL-J also inserts a <NL>, thus also searches for a <Nul> in the file. {Vi cannot handle <Nul> characters in the file at all}

CR-used-for-NL
When 'fileformat' is "mac", <NL> characters in the file are stored as <CR> characters internally. In the text they are shown as "^J". Otherwise this works similar to the usage of <NL> for a <Nul>.

When working with expression evaluation, a <NL> character in the pattern matches a <NL> in the string. The use of "\n" (backslash n) to match a <NL> doesn't work there, it only works to match text in the buffer.

pattern-multi-byte
Patterns will also work with multi-byte characters, mostly as you would expect. But invalid bytes may cause trouble, a pattern with an invalid byte will probably never match.

==
8. Composing characters *patterns-composing*

/\Z
When "\Z" appears anywhere in the pattern, all composing characters are ignored. Thus only the base characters need to match, the composing characters may be different and the number of composing characters may differ. Only relevant when 'encoding' is "utf-8".
Exception: If the pattern starts with one or more composing characters, these must match.

/\%C
Use "\%C" to skip any composing characters. For example, the pattern "a" does not match in "càt" (where the a has the composing character 0x0300), but "a\%C" does. Note that this does not match "cát" (where the á is character 0xe1, it does not have a compositing character). It does match "cat" (where the a is just an a).

When a composing character appears at the start of the pattern of after an item that doesn't include the composing character, a match is found at any character that includes this composing character.

When using a dot and a composing character, this works the same as the composing character by itself, except that it doesn't matter what comes before this.

The order of composing characters does not matter. Also, the text may have more composing characters than the pattern, it still matches. But all composing characters in the pattern must be found in the text.

Suppose B is a base character and x and y are composing characters:

pattern	text	match ~
Bxy	Bxy	yes (perfect match)
Bxy	Byx	yes (order ignored)
Bxy	By	no (x missing)
Bxy	Bx	no (y missing)
Bx	Bx	yes (perfect match)
Bx	By	no (x missing)
Bx	Bxy	yes (extra y ignored)
Bx	Byx	yes (extra y ignored)

==
9. Compare with Perl patterns *perl-patterns*

Vim's regexes are most similar to Perl's, in terms of what you can do. The
difference between them is mostly just notation; here's a summary of where
they differ:

Capability	in Vimspeak	in Perlspeak ~
force case insensitivity	\c	(?i)
force case sensitivity	\C	(?-i)
backref-less grouping	\%(atom\)	(?:atom)
conservative quantifiers	\{-n,m}	*?, +?, ??, {}?
0-width match	atom\@=	(?=atom)
0-width non-match	atom\@!	(?!atom)
0-width preceding match	atom\@<=	(?<=atom)
0-width preceding non-match	atom\@<!	(?<!atom)
match without retry	atom\@>	(?>atom)

Vim and Perl handle newline characters inside a string a bit differently:

In Perl, ^ and $ only match at the very beginning and end of the text,
by default, but you can set the 'm' flag, which lets them match at
embedded newlines as well. You can also set the 's' flag, which causes
a . to match newlines as well. (Both these flags can be changed inside
a pattern using the same syntax used for the i flag above, BTW.)

On the other hand, Vim's ^ and $ always match at embedded newlines, and
you get two separate atoms, \%^ and \%$, which only match at the very
start and end of the text, respectively. Vim solves the second problem
by giving you the _ "modifier": put it in front of a . or a character
class, and they will match newlines as well.

Finally, these constructs are unique to Perl:
- execution of arbitrary code in the regex: (?{perl code})
- conditional expressions: (?(condition)true-expr|false-expr)

...and these are unique to Vim:
- changing the magic-ness of a pattern: \v \V \m \M
  (very useful for avoiding backslashitis)
- sequence of optionally matching atoms: \%[atoms]
- \& (which is to \| what "and" is to "or"; it forces several branches
  to match at one spot)
- matching lines/columns by number: \%5l \%5c \%5v
- setting the start and end of the match: \zs \ze

==
10. Highlighting matches *match-highlight*

:mat *:match*
:mat[ch] {group} /{pattern}/
Define a pattern to highlight in the current window. It will
be highlighted with {group}. Example: >
:highlight MyGroup ctermbg=green guibg=green
:match MyGroup /TODO/
< Instead of // any character can be used to mark the start and
end of the {pattern}. Watch out for using special characters,
such as '"' and '|'.

{group} must exist at the moment this command is executed.

The {group} highlighting still applies when a character is

```
                to be highlighted for 'hlsearch', as the highlighting for
                matches is given higher priority than that of 'hlsearch'.
                Syntax highlighting (see 'syntax') is also overruled by
                matches.

                Note that highlighting the last used search pattern with
                'hlsearch' is used in all windows, while the pattern defined
                with ":match" only exists in the current window.  It is kept
                when switching to another buffer.

                'ignorecase' does not apply, use |/\c| in the pattern to
                ignore case.  Otherwise case is not ignored.

                'redrawtime' defines the maximum time searched for pattern
                matches.

                When matching end-of-line and Vim redraws only part of the
                display you may get unexpected results.  That is because Vim
                looks for a match in the line where redrawing starts.

                Also see |matcharg()| and |getmatches()|. The former returns
                the highlight group and pattern of a previous |:match|
                command.  The latter returns a list with highlight groups and
                patterns defined by both |matchadd()| and |:match|.

                Highlighting matches using |:match| are limited to three
                matches (aside from |:match|, |:2match| and |:3match| are
                available). |matchadd()| does not have this limitation and in
                addition makes it possible to prioritize matches.

                Another example, which highlights all characters in virtual
                column 72 and more: >
                        :highlight rightMargin term=bold ctermfg=blue guifg=blue
                        :match rightMargin /.\%>72v/
<               To highlight all character that are in virtual column 7: >
                        :highlight col8 ctermbg=grey guibg=grey
                        :match col8 /\%<8v.\%>7v/
<               Note the use of two items to also match a character that
                occupies more than one virtual column, such as a TAB.

:mat[ch]
:mat[ch] none
                Clear a previously defined match pattern.


:2mat[ch] {group} /{pattern}/                                   *:2match*
:2mat[ch]
:2mat[ch] none
:3mat[ch] {group} /{pattern}/                                   *:3match*
:3mat[ch]
:3mat[ch] none
                Just like |:match| above, but set a separate match.  Thus
                there can be three matches active at the same time.  The match
                with the lowest number has priority if several match at the
                same position.
                The ":3match" command is used by the |matchparen| plugin.  You
                are suggested to use ":match" for manual matching and
                ":2match" for another plugin.


 vim:tw=78:ts=8:ft=help:norl:
*map.txt*       For Vim version 8.0.  Last change: 2017 Sep 23
```

VIM REFERENCE MANUAL by Bram Moolenaar

Key mapping, abbreviations and user-defined commands.

This subject is introduced in sections |05.3|, |24.7| and |40.1| of the user manual.

==
1. Key mapping *key-mapping* *mapping* *macro*

Key mapping is used to change the meaning of typed keys. The most common use is to define a sequence of commands for a function key. Example: >

```
:map <F2> a<C-R>=strftime("%c")<CR><Esc>
```

This appends the current date and time after the cursor (in <> notation |<>|).

1.1 MAP COMMANDS *:map-commands*

There are commands to enter new mappings, remove mappings and list mappings. See |map-overview| for the various forms of "map" and their relationships with modes.

{lhs} means left-hand-side *{lhs}*
{rhs} means right-hand-side *{rhs}*

:map	{lhs} {rhs}	|mapmode-nvo|	*:map*
:nm[ap]	{lhs} {rhs}	|mapmode-n|	*:nm* *:nmap*
:vm[ap]	{lhs} {rhs}	|mapmode-v|	*:vm* *:vmap*
:xm[ap]	{lhs} {rhs}	|mapmode-x|	*:xm* *:xmap*
:smap	{lhs} {rhs}	|mapmode-s|	*:smap*
:om[ap]	{lhs} {rhs}	|mapmode-o|	*:om* *:omap*
:map!	{lhs} {rhs}	|mapmode-ic|	*:map!*
:im[ap]	{lhs} {rhs}	|mapmode-i|	*:im* *:imap*
:lm[ap]	{lhs} {rhs}	|mapmode-l|	*:lm* *:lmap*
:cm[ap]	{lhs} {rhs}	|mapmode-c|	*:cm* *:cmap*
:tma[p]	{lhs} {rhs}	|mapmode-t|	*:tma* *:tmap*

Map the key sequence {lhs} to {rhs} for the modes where the map command applies. The result, including {rhs}, is then further scanned for mappings. This allows for nested and recursive use of mappings.

```
                                                *:nore* *:norem*
:no[remap]  {lhs} {rhs}         |mapmode-nvo|   *:no*  *:noremap* *:nor*
:nn[oremap] {lhs} {rhs}         |mapmode-n|     *:nn*  *:nnoremap*
:vn[oremap] {lhs} {rhs}         |mapmode-v|     *:vn*  *:vnoremap*
:xn[oremap] {lhs} {rhs}         |mapmode-x|     *:xn*  *:xnoremap*
:snor[emap] {lhs} {rhs}         |mapmode-s|     *:snor* *:snoremap*
:ono[remap] {lhs} {rhs}         |mapmode-o|     *:ono* *:onoremap*
:no[remap]! {lhs} {rhs}         |mapmode-ic|    *:no!* *:noremap!*
:ino[remap] {lhs} {rhs}         |mapmode-i|     *:ino* *:inoremap*
:ln[oremap] {lhs} {rhs}         |mapmode-l|     *:ln*  *:lnoremap*
:cno[remap] {lhs} {rhs}         |mapmode-c|     *:cno* *:cnoremap*
:tno[remap] {lhs} {rhs}         |mapmode-t|     *:tno* *:tnoremap*
                        Map the key sequence {lhs} to {rhs} for the modes
                        where the map command applies.  Disallow mapping of
                        {rhs}, to avoid nested and recursive mappings.  Often
                        used to redefine a command.  {not in Vi}


:unm[ap]  {lhs}                 |mapmode-nvo|           *:unm*  *:unmap*
:nun[map] {lhs}                 |mapmode-n|             *:nun*  *:nunmap*
:vu[nmap] {lhs}                 |mapmode-v|             *:vu*   *:vunmap*
:xu[nmap] {lhs}                 |mapmode-x|             *:xu*   *:xunmap*
:sunm[ap] {lhs}                 |mapmode-s|             *:sunm* *:sunmap*
:ou[nmap] {lhs}                 |mapmode-o|             *:ou*   *:ounmap*
:unm[ap]! {lhs}                 |mapmode-ic|            *:unm!* *:unmap!*
:iu[nmap] {lhs}                 |mapmode-i|             *:iu*   *:iunmap*
:lu[nmap] {lhs}                 |mapmode-l|             *:lu*   *:lunmap*
:cu[nmap] {lhs}                 |mapmode-c|             *:cu*   *:cunmap*
:tunma[p] {lhs}                 |mapmode-t|             *:tunma* *:tunmap*
                        Remove the mapping of {lhs} for the modes where the
                        map command applies.  The mapping may remain defined
                        for other modes where it applies.
                        Note: Trailing spaces are included in the {lhs}.  This
                        unmap does NOT work: >
                                :map @@ foo
                                :unmap @@ | print

:mapc[lear]                     |mapmode-nvo|           *:mapc*   *:mapclear*
:nmapc[lear]                    |mapmode-n|             *:nmapc*  *:nmapclear*
:vmapc[lear]                    |mapmode-v|             *:vmapc*  *:vmapclear*
:xmapc[lear]                    |mapmode-x|             *:xmapc*  *:xmapclear*
:smapc[lear]                    |mapmode-s|             *:smapc*  *:smapclear*
:omapc[lear]                    |mapmode-o|             *:omapc*  *:omapclear*
:mapc[lear]!                    |mapmode-ic|            *:mapc!*  *:mapclear!*
:imapc[lear]                    |mapmode-i|             *:imapc*  *:imapclear*
:lmapc[lear]                    |mapmode-l|             *:lmapc*  *:lmapclear*
:cmapc[lear]                    |mapmode-c|             *:cmapc*  *:cmapclear*
:tmapc[lear]                    |mapmode-t|             *:tmapc*  *:tmapclear*
                        Remove ALL mappings for the modes where the map
                        command applies.  {not in Vi}
                        Use the <buffer> argument to remove buffer-local
                        mappings |:map-<buffer>|
                        Warning: This also removes the default mappings.

:map                            |mapmode-nvo|
:nm[ap]                         |mapmode-n|
:vm[ap]                         |mapmode-v|
:xm[ap]                         |mapmode-x|
:sm[ap]                         |mapmode-s|
:om[ap]                         |mapmode-o|
:map!                           |mapmode-ic|
:im[ap]                         |mapmode-i|
```

```
:lm[ap]                         |mapmode-l|
:cm[ap]                         |mapmode-c|
:tma[p]                         |mapmode-t|
                        List all key mappings for the modes where the map
                        command applies.  Note that ":map" and ":map!" are
                        used most often, because they include the other modes.

:map    {lhs}                   |mapmode-nvo|          *:map_l*
:nm[ap] {lhs}                   |mapmode-n|            *:nmap_l*
:vm[ap] {lhs}                   |mapmode-v|            *:vmap_l*
:xm[ap] {lhs}                   |mapmode-x|            *:xmap_l*
:sm[ap] {lhs}                   |mapmode-s|            *:smap_l*
:om[ap] {lhs}                   |mapmode-o|            *:omap_l*
:map!   {lhs}                   |mapmode-ic|           *:map_l!*
:im[ap] {lhs}                   |mapmode-i|            *:imap_l*
:lm[ap] {lhs}                   |mapmode-l|            *:lmap_l*
:cm[ap] {lhs}                   |mapmode-c|            *:cmap_l*
:tma[p] {lhs}                   |mapmode-t|            *:tmap_l*
                        List the key mappings for the key sequences starting
                        with {lhs} in the modes where the map command applies.
                        {not in Vi}

These commands are used to map a key or key sequence to a string of
characters.  You can use this to put command sequences under function keys,
translate one key into another, etc.  See |:mkexrc| for how to save and
restore the current mappings.

                                                        *map-ambiguous*
When two mappings start with the same sequence of characters, they are
ambiguous.  Example: >
        :imap aa foo
        :imap aaa bar
When Vim has read "aa", it will need to get another character to be able to
decide if "aa" or "aaa" should be mapped.  This means that after typing "aa"
that mapping won't get expanded yet, Vim is waiting for another character.
If you type a space, then "foo" will get inserted, plus the space.  If you
type "a", then "bar" will get inserted.
{Vi does not allow ambiguous mappings}


1.2 SPECIAL ARGUMENTS                                   *:map-arguments*

"<buffer>", "<nowait>", "<silent>", "<special>", "<script>", "<expr>" and
"<unique>" can be used in any order.  They must appear right after the
command, before any other arguments.

                                *:map-local* *:map-<buffer>* *E224* *E225*
If the first argument to one of these commands is "<buffer>" the mapping will
be effective in the current buffer only.  Example: >
        :map <buffer>  ,w  /[.,;]<CR>
Then you can map ",w" to something else in another buffer: >
        :map <buffer>  ,w  /[#&!]<CR>
The local buffer mappings are used before the global ones.  See <nowait> below
to make a short local mapping not taking effect when a longer global one
exists.
The "<buffer>" argument can also be used to clear mappings: >
        :unmap <buffer> ,w
        :mapclear <buffer>
Local mappings are also cleared when a buffer is deleted, but not when it is
unloaded.  Just like local option values.
Also see |map-precedence|.
```

:map-<nowait> *:map-nowait*
When defining a buffer-local mapping for "," there may be a global mapping
that starts with ",". Then you need to type another character for Vim to know
whether to use the "," mapping or the longer one. To avoid this add the
<nowait> argument. Then the mapping will be used when it matches, Vim does
not wait for more characters to be typed. However, if the characters were
already typed they are used.

:map-<silent> *:map-silent*
To define a mapping which will not be echoed on the command line, add
"<silent>" as the first argument. Example: >
:map <silent> ,h /Header<CR>
The search string will not be echoed when using this mapping. Messages from
the executed command are still given though. To shut them up too, add a
":silent" in the executed command: >
:map <silent> ,h :exe ":silent normal /Header\r"<CR>
Prompts will still be given, e.g., for inputdialog().
Using "<silent>" for an abbreviation is possible, but will cause redrawing of
the command line to fail.

:map-<special> *:map-special*
Define a mapping with <> notation for special keys, even though the "<" flag
may appear in 'cpoptions'. This is useful if the side effect of setting
'cpoptions' is not desired. Example: >
:map <special> <F12> /Header<CR>
<
:map-<script> *:map-script*
If the first argument to one of these commands is "<script>" and it is used to
define a new mapping or abbreviation, the mapping will only remap characters
in the {rhs} using mappings that were defined local to a script, starting with
"<SID>". This can be used to avoid that mappings from outside a script
interfere (e.g., when CTRL-V is remapped in mswin.vim), but do use other
mappings defined in the script.
Note: ":map <script>" and ":noremap <script>" do the same thing. The
"<script>" overrules the command name. Using ":noremap <script>" is
preferred, because it's clearer that remapping is (mostly) disabled.

:map-<unique> *E226* *E227*
If the first argument to one of these commands is "<unique>" and it is used to
define a new mapping or abbreviation, the command will fail if the mapping or
abbreviation already exists. Example: >
:map <unique> ,w /[#&!]<CR>
When defining a local mapping, there will also be a check if a global map
already exists which is equal.
Example of what will fail: >
:map ,w /[#&!]<CR>
:map <buffer> <unique> ,w /[.,;]<CR>
If you want to map a key and then have it do what it was originally mapped to,
have a look at |maparg()|.

:map-<expr> *:map-expression*
If the first argument to one of these commands is "<expr>" and it is used to
define a new mapping or abbreviation, the argument is an expression. The
expression is evaluated to obtain the {rhs} that is used. Example: >
:inoremap <expr> . InsertDot()
The result of the InsertDot() function will be inserted. It could check the
text before the cursor and start omni completion when some condition is met.

For abbreviations |v:char| is set to the character that was typed to trigger
the abbreviation. You can use this to decide how to expand the {lhs}. You
should not either insert or change the v:char.

```
Be very careful about side effects!  The expression is evaluated while
obtaining characters, you may very well make the command dysfunctional.
For this reason the following is blocked:
- Changing the buffer text |textlock|.
- Editing another buffer.
- The |:normal| command.
- Moving the cursor is allowed, but it is restored afterwards.
If you want the mapping to do any of these let the returned characters do
that.

You can use getchar(), it consumes typeahead if there is any. E.g., if you
have these mappings: >
  inoremap <expr> <C-L> nr2char(getchar())
  inoremap <expr> <C-L>x "foo"
If you now type CTRL-L nothing happens yet, Vim needs the next character to
decide what mapping to use.  If you type 'x' the second mapping is used and
"foo" is inserted.  If you type any other key the first mapping is used,
getchar() gets the typed key and returns it.

Here is an example that inserts a list number that increases: >
        let counter = 0
        inoremap <expr> <C-L> ListItem()
        inoremap <expr> <C-R> ListReset()

        func ListItem()
          let g:counter += 1
          return g:counter . '. '
        endfunc

        func ListReset()
          let g:counter = 0
          return ''
        endfunc

CTRL-L inserts the next number, CTRL-R resets the count.  CTRL-R returns an
empty string, so that nothing is inserted.

Note that there are some tricks to make special keys work and escape CSI bytes
in the text.  The |:map| command also does this, thus you must avoid that it
is done twice.  This does not work: >
        :imap <expr> <F3> "<Char-0x611B>"
Because the <Char- sequence is escaped for being a |:imap| argument and then
again for using <expr>.  This does work: >
        :imap <expr> <F3> "\u611B"
Using 0x80 as a single byte before other text does not work, it will be seen
as a special key.


1.3 MAPPING AND MODES                                   *:map-modes*
                        *mapmode-nvo* *mapmode-n* *mapmode-v* *mapmode-o*

There are six sets of mappings
- For Normal mode: When typing commands.
- For Visual mode: When typing commands while the Visual area is highlighted.
- For Select mode: like Visual mode but typing text replaces the selection.
- For Operator-pending mode: When an operator is pending (after "d", "y", "c",
  etc.).  See below: |omap-info|.
- For Insert mode.  These are also used in Replace mode.
- For Command-line mode: When entering a ":" or "/" command.

Special case: While typing a count for a command in Normal mode, mapping zero
is disabled.  This makes it possible to map zero without making it impossible
```

to type a count with a zero.

map-overview *map-modes*

Overview of which map command works in which mode. More details below.

COMMANDS			MODES ~
:map	:noremap	:unmap	Normal, Visual, Select, Operator-pending
:nmap	:nnoremap	:nunmap	Normal
:vmap	:vnoremap	:vunmap	Visual and Select
:smap	:snoremap	:sunmap	Select
:xmap	:xnoremap	:xunmap	Visual
:omap	:onoremap	:ounmap	Operator-pending
:map!	:noremap!	:unmap!	Insert and Command-line
:imap	:inoremap	:iunmap	Insert
:lmap	:lnoremap	:lunmap	Insert, Command-line, Lang-Arg
:cmap	:cnoremap	:cunmap	Command-line
:tmap	:tnoremap	:tunmap	Terminal-Job

COMMANDS				MODES ~		
				Normal	Visual+Select	Operator-pending ~
:map	:noremap	:unmap	:mapclear	yes	yes	yes
:nmap	:nnoremap	:nunmap	:nmapclear	yes	-	-
:vmap	:vnoremap	:vunmap	:vmapclear	-	yes	-
:omap	:onoremap	:ounmap	:omapclear	-	-	yes

:nunmap can also be used outside of a monastery.

mapmode-x *mapmode-s*

Some commands work both in Visual and Select mode, some in only one. Note that quite often "Visual" is mentioned where both Visual and Select mode apply. |Select-mode-mapping|

NOTE: Mapping a printable character in Select mode may confuse the user. It's better to explicitly use :xmap and :smap for printable characters. Or use :sunmap after defining the mapping.

COMMANDS				MODES ~	
				Visual	Select ~
:vmap	:vnoremap	:vunmap	:vmapclear	yes	yes
:xmap	:xnoremap	:xunmap	:xmapclear	yes	-
:smap	:snoremap	:sunmap	:smapclear	-	yes

mapmode-ic *mapmode-i* *mapmode-c* *mapmode-l*

Some commands work both in Insert mode and Command-line mode, some not:

COMMANDS				MODES ~		
				Insert	Command-line	Lang-Arg ~
:map!	:noremap!	:unmap!	:mapclear!	yes	yes	-
:imap	:inoremap	:iunmap	:imapclear	yes	-	-
:cmap	:cnoremap	:cunmap	:cmapclear	-	yes	-
:lmap	:lnoremap	:lunmap	:lmapclear	yes*	yes*	yes*

The original Vi did not have separate mappings for Normal/Visual/Operator-pending mode and for Insert/Command-line mode. Therefore the ":map" and ":map!" commands enter and display mappings for several modes. In Vim you can use the ":nmap", ":vmap", ":omap", ":cmap" and ":imap" commands to enter mappings for each mode separately.

mapmode-t

The terminal mappings are used in a terminal window, when typing keys for the job running in the terminal. See |terminal-typing|.

omap-info

Operator-pending mappings can be used to define a movement command that can be

used with any operator. Simple example: ":omap { w" makes "y{" work like "yw"
and "d{" like "dw".

To ignore the starting cursor position and select different text, you can have
the omap start Visual mode to select the text to be operated upon. Example
that operates on a function name in the current line: >
        onoremap <silent> F :<C-U>normal! 0f(hviw<CR>
The CTRL-U (<C-U>) is used to remove the range that Vim may insert. The
Normal mode commands find the first '(' character and select the first word
before it. That usually is the function name.

To enter a mapping for Normal and Visual mode, but not Operator-pending mode,
first define it for all three modes, then unmap it for Operator-pending mode:
        :map   xx something-difficult
        :ounmap xx
Likewise for a mapping for Visual and Operator-pending mode or Normal and
Operator-pending mode.

                                                language-mapping
":lmap" defines a mapping that applies to:
- Insert mode
- Command-line mode
- when entering a search pattern
- the argument of the commands that accept a text character, such as "r" and
  "f"
- for the input() line
Generally: Whenever a character is to be typed that is part of the text in the
buffer, not a Vim command character. "Lang-Arg" isn't really another mode,
it's just used here for this situation.
   The simplest way to load a set of related language mappings is by using the
'keymap' option. See |45.5|.
   In Insert mode and in Command-line mode the mappings can be disabled with
the CTRL-^ command |i_CTRL-^| |c_CTRL-^|. These commands change the value of
the 'iminsert' option. When starting to enter a normal command line (not a
search pattern) the mappings are disabled until a CTRL-^ is typed. The state
last used is remembered for Insert mode and Search patterns separately. The
state for Insert mode is also used when typing a character as an argument to
command like "f" or "t".
   Language mappings will never be applied to already mapped characters. They
are only used for typed characters. This assumes that the language mapping
was already done when typing the mapping.

1.4 LISTING MAPPINGS *map-listing*

When listing mappings the characters in the first two columns are:

CHAR	MODE ~
<Space>	Normal, Visual, Select and Operator-pending
n	Normal
v	Visual and Select
s	Select
x	Visual
o	Operator-pending
!	Insert and Command-line
i	Insert
l	":lmap" mappings for Insert, Command-line and Lang-Arg
c	Command-line
t	Terminal-Job

Just before the {rhs} a special character can appear:
        *   indicates that it is not remappable

```
        &       indicates that only script-local mappings are remappable
        @       indicates a buffer-local mapping

Everything from the first non-blank after {lhs} up to the end of the line
(or '|') is considered to be part of {rhs}.  This allows the {rhs} to end
with a space.

Note: When using mappings for Visual mode, you can use the "'<" mark, which
is the start of the last selected Visual area in the current buffer |'<|.

The |:filter| command can be used to select what mappings to list.  The
pattern is matched against the {lhs} and {rhs} in the raw form.

                                                            *:map-verbose*
When 'verbose' is non-zero, listing a key map will also display where it was
last defined.  Example: >

        :verbose map <C-W>*
        n  <C-W>*      * <C-W><C-S>*
                Last set from /home/abcd/.vimrc

See |:verbose-cmd| for more information.


1.5 MAPPING SPECIAL KEYS                                *:map-special-keys*

There are three ways to map a special key:
1. The Vi-compatible method: Map the key code.  Often this is a sequence that
   starts with <Esc>.  To enter a mapping like this you type ":map " and then
   you have to type CTRL-V before hitting the function key.  Note that when
   the key code for the key is in the termcap (the t_ options), it will
   automatically be translated into the internal code and become the second
   way of mapping (unless the 'k' flag is included in 'cpoptions').
2. The second method is to use the internal code for the function key.  To
   enter such a mapping type CTRL-K and then hit the function key, or use
   the form "#1", "#2", .. "#9", "#0", "<Up>", "<S-Down>", "<S-F7>", etc.
   (see table of keys |key-notation|, all keys from <Up> can be used).  The
   first ten function keys can be defined in two ways: Just the number, like
   "#2", and with "<F>", like "<F2>".  Both stand for function key 2.  "#0"
   refers to function key 10, defined with option 't_f10', which may be
   function key zero on some keyboards.  The <> form cannot be used when
   'cpoptions' includes the '<' flag.
3. Use the termcap entry, with the form <t_xx>, where "xx" is the name of the
   termcap entry.  Any string entry can be used.  For example: >
     :map <t_F3> G
<  Maps function key 13 to "G".  This does not work if 'cpoptions' includes
   the '<' flag.

The advantage of the second and third method is that the mapping will work on
different terminals without modification (the function key will be
translated into the same internal code or the actual key code, no matter what
terminal you are using.  The termcap must be correct for this to work, and you
must use the same mappings).

DETAIL: Vim first checks if a sequence from the keyboard is mapped.  If it
isn't the terminal key codes are tried (see |terminal-options|).  If a
terminal code is found it is replaced with the internal code.  Then the check
for a mapping is done again (so you can map an internal code to something
else).  What is written into the script file depends on what is recognized.
If the terminal key code was recognized as a mapping the key code itself is
written to the script file.  If it was recognized as a terminal code the
internal code is written to the script file.
```

1.6 SPECIAL CHARACTERS *:map-special-chars*
map_backslash *map-backslash*
Note that only CTRL-V is mentioned here as a special character for mappings
and abbreviations. When 'cpoptions' does not contain 'B', a backslash can
also be used like CTRL-V. The <> notation can be fully used then |<>|. But
you cannot use "<C-V>" like CTRL-V to escape the special meaning of what
follows.

To map a backslash, or use a backslash literally in the {rhs}, the special
sequence "<Bslash>" can be used. This avoids the need to double backslashes
when using nested mappings.

map_CTRL-C *map-CTRL-C*
Using CTRL-C in the {lhs} is possible, but it will only work when Vim is
waiting for a key, not when Vim is busy with something. When Vim is busy
CTRL-C interrupts/breaks the command.
When using the GUI version on MS-Windows CTRL-C can be mapped to allow a Copy
command to the clipboard. Use CTRL-Break to interrupt Vim.

map_space_in_lhs *map-space_in_lhs*
To include a space in {lhs} precede it with a CTRL-V (type two CTRL-Vs for
each space).
map_space_in_rhs *map-space_in_rhs*
If you want a {rhs} that starts with a space, use "<Space>". To be fully Vi
compatible (but unreadable) don't use the |<>| notation, precede {rhs} with a
single CTRL-V (you have to type CTRL-V two times).
map_empty_rhs *map-empty-rhs*
You can create an empty {rhs} by typing nothing after a single CTRL-V (you
have to type CTRL-V two times). Unfortunately, you cannot do this in a vimrc
file.
<Nop>
An easier way to get a mapping that doesn't produce anything, is to use
"<Nop>" for the {rhs}. This only works when the |<>| notation is enabled.
For example, to make sure that function key 8 does nothing at all: >
:map <F8> <Nop>
:map! <F8> <Nop>
<
map-multibyte
It is possible to map multibyte characters, but only the whole character. You
cannot map the first byte only. This was done to prevent problems in this
scenario: >
:set encoding=latin1
:imap <M-C> foo
:set encoding=utf-8
The mapping for <M-C> is defined with the latin1 encoding, resulting in a 0xc3
byte. If you type the character **\E1** (0xe1 <M-a>) in UTF-8 encoding this is the
two bytes 0xc3 0xa1. You don't want the 0xc3 byte to be mapped then or
otherwise it would be impossible to type the **\E1** character.

<Leader> *mapleader*
To define a mapping which uses the "mapleader" variable, the special string
"<Leader>" can be used. It is replaced with the string value of "mapleader".
If "mapleader" is not set or empty, a backslash is used instead. Example: >
:map <Leader>A oanother line<Esc>
Works like: >
:map \A oanother line<Esc>
But after: >
:let mapleader = ","
It works like: >
:map ,A oanother line<Esc>

Note that the value of "mapleader" is used at the moment the mapping is
defined. Changing "mapleader" after that has no effect for already defined
mappings.

<LocalLeader> *maplocalleader*
<LocalLeader> is just like <Leader>, except that it uses "maplocalleader"
instead of "mapleader". <LocalLeader> is to be used for mappings which are
local to a buffer. Example: >
      :map <buffer> <LocalLeader>A oanother line<Esc>
<
In a global plugin <Leader> should be used and in a filetype plugin
<LocalLeader>. "mapleader" and "maplocalleader" can be equal. Although, if
you make them different, there is a smaller chance of mappings from global
plugins to clash with mappings for filetype plugins. For example, you could
keep "mapleader" at the default backslash, and set "maplocalleader" to an
underscore.

map-<SID>
In a script the special key name "<SID>" can be used to define a mapping
that's local to the script. See |<SID>| for details.

<Plug>
The special key name "<Plug>" can be used for an internal mapping, which is
not to be matched with any key sequence. This is useful in plugins
|using-<Plug>|.

<Char> *<Char->*
To map a character by its decimal, octal or hexadecimal number the <Char>
construct can be used:
        <Char-123>   character 123
        <Char-033>   character 27
        <Char-0x7f>   character 127
        <S-Char-114>   character 114 ('r') shifted ('R')
This is useful to specify a (multi-byte) character in a 'keymap' file.
Upper and lowercase differences are ignored.

map-comments
It is not possible to put a comment after these commands, because the '"'
character is considered to be part of the {lhs} or {rhs}. However, one can
use |", since this starts a new, empty command with a comment.

map_bar *map-bar*
Since the '|' character is used to separate a map command from the next
command, you will have to do something special to include a '|' in {rhs}.
There are three methods:
   use   works when   example   ~
   <Bar>   '<' is not in 'cpoptions'   :map _l :!ls <Bar> more^M
   \| 'b' is not in 'cpoptions'   :map _l :!ls \| more^M
   ^V| always, in Vim and Vi   :map _l :!ls ^V| more^M

(here ^V stands for CTRL-V; to get one CTRL-V you have to type it twice; you
cannot use the <> notation "<C-V>" here).

All three work when you use the default setting for 'cpoptions'.

When 'b' is present in 'cpoptions', "\|" will be recognized as a mapping
ending in a '\' and then another command. This is Vi compatible, but
illogical when compared to other commands.

map_return *map-return*
When you have a mapping that contains an Ex command, you need to put a line

terminator after it to have it executed. The use of <CR> is recommended for this (see |<>|). Example: >
```
:map _ls :!ls -l %:S<CR>:echo "the end"<CR>
```

To avoid mapping of the characters you type in insert or Command-line mode, type a CTRL-V first. The mapping in Insert mode is disabled if the 'paste' option is on.

map-error

Note that when an error is encountered (that causes an error message or beep) the rest of the mapping is not executed. This is Vi-compatible.

Note that the second character (argument) of the commands @zZtTfF[]rm'`"v and CTRL-X is not mapped. This was done to be able to use all the named registers and marks, even when the command with the same name has been mapped.

1.7 WHAT KEYS TO MAP *map-which-keys*

If you are going to map something, you will need to choose which key(s) to use for the {lhs}. You will have to avoid keys that are used for Vim commands, otherwise you would not be able to use those commands anymore. Here are a few suggestions:
- Function keys <F2>, <F3>, etc.. Also the shifted function keys <S-F1>, <S-F2>, etc. Note that <F1> is already used for the help command.
- Meta-keys (with the ALT key pressed). Depending on your keyboard accented characters may be used as well. |:map-alt-keys|
- Use the '_' or ',' character and then any other character. The "_" and "," commands do exist in Vim (see |_| and |,|), but you probably never use them.
- Use a key that is a synonym for another command. For example: CTRL-P and CTRL-N. Use an extra character to allow more mappings.
- The key defined by <Leader> and one or more other keys. This is especially useful in scripts. |mapleader|

See the file "index" for keys that are not used and thus can be mapped without losing any builtin function. You can also use ":help {key}^D" to find out if a key is used for some command. ({key} is the specific key you want to find out about, ^D is CTRL-D).

1.8 EXAMPLES *map-examples*

A few examples (given as you type them, for "<CR>" you type four characters; the '<' flag must not be present in 'cpoptions' for this to work). >

```
:map <F3> o#include
:map <M-g> /foo<CR>cwbar<Esc>
:map _x d/END/e<CR>
:map! qq quadrillion questions
```

Multiplying a count

When you type a count before triggering a mapping, it's like the count was typed before the {lhs}. For example, with this mapping: >
```
:map <F4> 3w
```
Typing 2<F4> will result in "23w". Thus not moving 2 * 3 words but 23 words. If you want to multiply counts use the expression register: >
```
:map <F4> @='3w'<CR>
```
The part between quotes is the expression being executed. |@=|

1.9 USING MAPPINGS *map-typing*

Vim will compare what you type with the start of a mapped sequence. If there is an incomplete match, it will get more characters until there either is a complete match or until there is no match at all. Example: If you map! "qq", the first 'q' will not appear on the screen until you type another character. This is because Vim cannot know if the next character will be a 'q' or not. If the 'timeout' option is on (which is the default) Vim will only wait for one second (or as long as specified with the 'timeoutlen' option). After that it assumes that the 'q' is to be interpreted as such. If you type slowly, or your system is slow, reset the 'timeout' option. Then you might want to set the 'ttimeout' option.

map-precedence
Buffer-local mappings (defined using |:map-<buffer>|) take precedence over global mappings. When a buffer-local mapping is the same as a global mapping, Vim will use the buffer-local mapping. In addition, Vim will use a complete mapping immediately if it was defined with <nowait>, even if a longer mapping has the same prefix. For example, given the following two mappings: >
    :map <buffer> <nowait> \a   :echo "Local \a"<CR>
    :map   \abc :echo "Global \abc"<CR>
When typing \a the buffer-local mapping will be used immediately. Vim will not wait for more characters to see if the user might be typing \abc.

map-keys-fails
There are situations where key codes might not be recognized:
- Vim can only read part of the key code. Mostly this is only the first character. This happens on some Unix versions in an xterm.
- The key code is after character(s) that are mapped. E.g., "<F1><F1>" or "g<F1>".

The result is that the key code is not recognized in this situation, and the mapping fails. There are two actions needed to avoid this problem:

- Remove the 'K' flag from 'cpoptions'. This will make Vim wait for the rest of the characters of the function key.
- When using <F1> to <F4> the actual key code generated may correspond to <xF1> to <xF4>. There are mappings from <xF1> to <F1>, <xF2> to <F2>, etc., but these are not recognized after another half a mapping. Make sure the key codes for <F1> to <F4> are correct: >
        :set <F1>=<type CTRL-V><type F1>
< Type the <F1> as four characters. The part after the "=" must be done with the actual keys, not the literal text.
Another solution is to use the actual key code in the mapping for the second special key: >
        :map <F1><Esc>OP :echo "yes"<CR>
Don't type a real <Esc>, Vim will recognize the key code and replace it with <F1> anyway.

Another problem may be that when keeping ALT or Meta pressed the terminal prepends ESC instead of setting the 8th bit. See |:map-alt-keys|.

recursive_mapping
If you include the {lhs} in the {rhs} you have a recursive mapping. When {lhs} is typed, it will be replaced with {rhs}. When the {lhs} which is included in {rhs} is encountered it will be replaced with {rhs}, and so on. This makes it possible to repeat a command an infinite number of times. The only problem is that the only way to stop this is by causing an error. The macros to solve a maze uses this, look there for an example. There is one exception: If the {rhs} starts with {lhs}, the first character is not mapped again (this is Vi compatible).
For example: >

```
   :map ab abcd
will execute the "a" command and insert "bcd" in the text.  The "ab" in the
{rhs} will not be mapped again.

If you want to exchange the meaning of two keys you should use the :noremap
command.  For example: >
   :noremap k j
   :noremap j k
This will exchange the cursor up and down commands.

With the normal :map command, when the 'remap' option is on, mapping takes
place until the text is found not to be a part of a {lhs}.  For example, if
you use: >
   :map x y
   :map y x
Vim will replace x with y, and then y with x, etc.  When this has happened
'maxmapdepth' times (default 1000), Vim will give the error message
"recursive mapping".

                                                        *:map-undo*
If you include an undo command inside a mapped sequence, this will bring the
text back in the state before executing the macro.  This is compatible with
the original Vi, as long as there is only one undo command in the mapped
sequence (having two undo commands in a mapped sequence did not make sense
in the original Vi, you would get back the text before the first undo).


1.10 MAPPING ALT-KEYS                                   *:map-alt-keys*

In the GUI Vim handles the Alt key itself, thus mapping keys with ALT should
always work.  But in a terminal Vim gets a sequence of bytes and has to figure
out whether ALT was pressed or not.

By default Vim assumes that pressing the ALT key sets the 8th bit of a typed
character.  Most decent terminals can work that way, such as xterm, aterm and
rxvt.  If your <A-k> mappings don't work it might be that the terminal is
prefixing the character with an ESC character.  But you can just as well type
ESC before a character, thus Vim doesn't know what happened (except for
checking the delay between characters, which is not reliable).

As of this writing, some mainstream terminals like gnome-terminal and konsole
use the ESC prefix.  There doesn't appear a way to have them use the 8th bit
instead.  Xterm should work well by default.  Aterm and rxvt should work well
when started with the "--meta8" argument.  You can also tweak resources like
"metaSendsEscape", "eightBitInput" and "eightBitOutput".

On the Linux console, this behavior can be toggled with the "setmetamode"
command.  Bear in mind that not using an ESC prefix could get you in trouble
with other programs.  You should make sure that bash has the "convert-meta"
option set to "on" in order for your Meta keybindings to still work on it
(it's the default readline behavior, unless changed by specific system
configuration).  For that, you can add the line: >

        set convert-meta on

to your ~/.inputrc file. If you're creating the file, you might want to use: >

        $include /etc/inputrc

as the first line, if that file exists on your system, to keep global options.
This may cause a problem for entering special characters, such as the umlaut.
Then you should use CTRL-V before that character.
```

Bear in mind that convert-meta has been reported to have troubles when used in
UTF-8 locales. On terminals like xterm, the "metaSendsEscape" resource can be
toggled on the fly through the "Main Options" menu, by pressing Ctrl-LeftClick
on the terminal; that's a good last resource in case you want to send ESC when
using other applications but not when inside Vim.

1.11 MAPPING AN OPERATOR *:map-operator*

An operator is used before a {motion} command. To define your own operator
you must create mapping that first sets the 'operatorfunc' option and then
invoke the |g@| operator. After the user types the {motion} command the
specified function will be called.

g@ *E774* *E775*

g@{motion} Call the function set by the 'operatorfunc' option.
The '[mark is positioned at the start of the text
moved over by {motion}, the '] mark on the last
character of the text.
The function is called with one String argument:
"line" {motion} was |linewise|
"char" {motion} was |characterwise|
"block" {motion} was |blockwise-visual|
Although "block" would rarely appear, since it can
only result from Visual mode where "g@" is not useful.
{not available when compiled without the |+eval|
feature}

Here is an example that counts the number of spaces with <F4>: >

```
nmap <silent> <F4> :set opfunc=CountSpaces<CR>g@
vmap <silent> <F4> :<C-U>call CountSpaces(visualmode(), 1)<CR>

function! CountSpaces(type, ...)
  let sel_save = &selection
  let &selection = "inclusive"
  let reg_save = @@

  if a:0  " Invoked from Visual mode, use gv command.
    silent exe "normal! gvy"
  elseif a:type == 'line'
    silent exe "normal! '[V']y"
  else
    silent exe "normal! `[v`]y"
  endif

  echomsg strlen(substitute(@@, '[^ ]', '', 'g'))

  let &selection = sel_save
  let @@ = reg_save
endfunction
```

Note that the 'selection' option is temporarily set to "inclusive" to be able
to yank exactly the right text by using Visual mode from the '[to the ']
mark.

Also note that there is a separate mapping for Visual mode. It removes the
"'<,'>" range that ":" inserts in Visual mode and invokes the function with
visualmode() and an extra argument.

==

2. Abbreviations *abbreviations* *Abbreviations*

Abbreviations are used in Insert mode, Replace mode and Command-line mode.
If you enter a word that is an abbreviation, it is replaced with the word it
stands for. This can be used to save typing for often used long words. And
you can use it to automatically correct obvious spelling errors.
Examples:

```
        :iab ms Microsoft
        :iab tihs this
```

There are three types of abbreviations:

full-id The "full-id" type consists entirely of keyword characters (letters
         and characters from 'iskeyword' option). This is the most common
         abbreviation.

         Examples: "foo", "g3", "-1"

end-id   The "end-id" type ends in a keyword character, but all the other
         characters are not keyword characters.

         Examples: "#i", "..f", "$/7"

non-id   The "non-id" type ends in a non-keyword character, the other
         characters may be of any type, excluding space and tab. {this type
         is not supported by Vi}

         Examples: "def#", "4/7$"

Examples of strings that cannot be abbreviations: "a.b", "#def", "a b", "_$r"

An abbreviation is only recognized when you type a non-keyword character.
This can also be the <Esc> that ends insert mode or the <CR> that ends a
command. The non-keyword character which ends the abbreviation is inserted
after the expanded abbreviation. An exception to this is the character <C-]>,
which is used to expand an abbreviation without inserting any extra
characters.

Example: >
```
   :ab hh       hello
<           "hh<Space>" is expanded to "hello<Space>"
            "hh<C-]>" is expanded to "hello"
```

The characters before the cursor must match the abbreviation. Each type has
an additional rule:

full-id In front of the match is a non-keyword character, or this is where
         the line or insertion starts. Exception: When the abbreviation is
         only one character, it is not recognized if there is a non-keyword
         character in front of it, other than a space or a tab.

end-id   In front of the match is a keyword character, or a space or a tab,
         or this is where the line or insertion starts.

non-id   In front of the match is a space, tab or the start of the line or
         the insertion.

Examples: ({CURSOR} is where you type a non-keyword character) >
```
   :ab foo   four old otters
<               " foo{CURSOR}"    is expanded to " four old otters"
                " foobar{CURSOR}" is not expanded
```

```
                "barfoo{CURSOR}"  is not expanded
>
   :ab #i #include
<               "#i{CURSOR}"      is expanded to "#include"
                ">#i{CURSOR}"     is not expanded
>
   :ab ;; <endofline>
<               "test;;"          is not expanded
                "test ;;"         is expanded to "test <endofline>"

To avoid the abbreviation in Insert mode: Type CTRL-V before the character
that would trigger the abbreviation.  E.g. CTRL-V <Space>.  Or type part of
the abbreviation, exit insert mode with <Esc>, re-enter insert mode with "a"
and type the rest.

To avoid the abbreviation in Command-line mode: Type CTRL-V twice somewhere in
the abbreviation to avoid it to be replaced.  A CTRL-V in front of a normal
character is mostly ignored otherwise.

It is possible to move the cursor after an abbreviation: >
   :iab if if ()<Left>
This does not work if 'cpoptions' includes the '<' flag. |<>|

You can even do more complicated things.  For example, to consume the space
typed after an abbreviation: >
   func Eatchar(pat)
      let c = nr2char(getchar(0))
      return (c =~ a:pat) ? '' : c
   endfunc
   iabbr <silent> if if ()<Left><C-R>=Eatchar('\s')<CR>

There are no default abbreviations.

Abbreviations are never recursive.  You can use ":ab f f-o-o" without any
problem.  But abbreviations can be mapped.  {some versions of Vi support
recursive abbreviations, for no apparent reason}

Abbreviations are disabled if the 'paste' option is on.

                                *:abbreviate-local* *:abbreviate-<buffer>*
Just like mappings, abbreviations can be local to a buffer.  This is mostly
used in a |filetype-plugin| file.  Example for a C plugin file: >
        :abb <buffer> FF  for (i = 0; i < ; ++i)
<
                                                *:ab* *:abbreviate*
:ab[breviate]           list all abbreviations.  The character in the first
                        column indicates the mode where the abbreviation is
                        used: 'i' for insert mode, 'c' for Command-line
                        mode, '!' for both.  These are the same as for
                        mappings, see |map-listing|.

                                                *:abbreviate-verbose*
When 'verbose' is non-zero, listing an abbreviation will also display where it
was last defined.  Example: >

        :verbose abbreviate
        !  teh           the
                Last set from /home/abcd/vim/abbr.vim

See |:verbose-cmd| for more information.

:ab[breviate] {lhs}     list the abbreviations that start with {lhs}
```

```
                        You may need to insert a CTRL-V (type it twice) to
                        avoid that a typed {lhs} is expanded, since
                        command-line abbreviations apply here.

:ab[breviate] [<expr>] [<buffer>] {lhs} {rhs}
                        add abbreviation for {lhs} to {rhs}.  If {lhs} already
                        existed it is replaced with the new {rhs}.  {rhs} may
                        contain spaces.
                        See |:map-<expr>| for the optional <expr> argument.
                        See |:map-<buffer>| for the optional <buffer> argument.

                                                *:una* *:unabbreviate*
:una[bbreviate] {lhs}   Remove abbreviation for {lhs} from the list.  If none
                        is found, remove abbreviations in which {lhs} matches
                        with the {rhs}.  This is done so that you can even
                        remove abbreviations after expansion.  To avoid
                        expansion insert a CTRL-V (type it twice).

                                                *:norea* *:noreabbrev*
:norea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but no remapping for this {rhs} {not
                        in Vi}

                                                *:ca* *:cabbrev*
:ca[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Command-line mode only.  {not
                        in Vi}

                                                *:cuna* *:cunabbrev*
:cuna[bbrev] {lhs}      same as ":una", but for Command-line mode only.  {not
                        in Vi}

                                                *:cnorea* *:cnoreabbrev*
:cnorea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Command-line mode only and no
                        remapping for this {rhs} {not in Vi}

                                                *:ia* *:iabbrev*
:ia[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Insert mode only.  {not in Vi}

                                                *:iuna* *:iunabbrev*
:iuna[bbrev] {lhs}      same as ":una", but for insert mode only.  {not in
                        Vi}

                                                *:inorea* *:inoreabbrev*
:inorea[bbrev] [<expr>] [<buffer>] [lhs] [rhs]
                        same as ":ab", but for Insert mode only and no
                        remapping for this {rhs} {not in Vi}

                                                        *:abc* *:abclear*
:abc[lear] [<buffer>]   Remove all abbreviations.  {not in Vi}

                                                        *:iabc* *:iabclear*
:iabc[lear] [<buffer>]  Remove all abbreviations for Insert mode.  {not in Vi}

                                                        *:cabc* *:cabclear*
:cabc[lear] [<buffer>]  Remove all abbreviations for Command-line mode.  {not
                        in Vi}

                                                        *using_CTRL-V*
It is possible to use special characters in the rhs of an abbreviation.
```

CTRL-V has to be used to avoid the special meaning of most non printable
characters. How many CTRL-Vs need to be typed depends on how you enter the
abbreviation. This also applies to mappings. Let's use an example here.

Suppose you want to abbreviate "esc" to enter an <Esc> character. When you
type the ":ab" command in Vim, you have to enter this: (here ^V is a CTRL-V
and ^[is <Esc>)

You type: ab esc ^V^V^V^V^V^[

        All keyboard input is subjected to ^V quote interpretation, so
        the first, third, and fifth ^V characters simply allow the second,
        and fourth ^Vs, and the ^[, to be entered into the command-line.

You see: ab esc ^V^V^[

        The command-line contains two actual ^Vs before the ^[. This is
        how it should appear in your .exrc file, if you choose to go that
        route. The first ^V is there to quote the second ^V; the :ab
        command uses ^V as its own quote character, so you can include quoted
        whitespace or the | character in the abbreviation. The :ab command
        doesn't do anything special with the ^[character, so it doesn't need
        to be quoted. (Although quoting isn't harmful; that's why typing 7
        [but not 8!] ^Vs works.)

Stored as: esc   ^V^[

        After parsing, the abbreviation's short form ("esc") and long form
        (the two characters "^V^[") are stored in the abbreviation table.
        If you give the :ab command with no arguments, this is how the
        abbreviation will be displayed.

        Later, when the abbreviation is expanded because the user typed in
        the word "esc", the long form is subjected to the same type of
        ^V interpretation as keyboard input. So the ^V protects the ^[
        character from being interpreted as the "exit Insert mode" character.
        Instead, the ^[is inserted into the text.

Expands to: ^[

[example given by Steve Kirkendall]

==
3. Local mappings and functions   *script-local*

When using several Vim script files, there is the danger that mappings and
functions used in one script use the same name as in other scripts. To avoid
this, they can be made local to the script.

                                                <SID> *<SNR>* *E81*
The string "<SID>" can be used in a mapping or menu. This requires that the
'<' flag is not present in 'cpoptions'.
   When executing the map command, Vim will replace "<SID>" with the special
key code <SNR>, followed by a number that's unique for the script, and an
underscore. Example: >
        :map <SID>Add
could define a mapping "<SNR>23_Add".

When defining a function in a script, "s:" can be prepended to the name to
make it local to the script. But when a mapping is executed from outside of
the script, it doesn't know in which script the function was defined. To
avoid this problem, use "<SID>" instead of "s:". The same translation is done

as for mappings. This makes it possible to define a call to the function in
a mapping.

When a local function is executed, it runs in the context of the script it was
defined in. This means that new functions and mappings it defines can also
use "s:" or "<SID>" and it will use the same unique number as when the
function itself was defined. Also, the "s:var" local script variables can be
used.

When executing an autocommand or a user command, it will run in the context of
the script it was defined in. This makes it possible that the command calls a
local function or uses a local mapping.

Otherwise, using "<SID>" outside of a script context is an error.

If you need to get the script number to use in a complicated script, you can
use this function: >
        function s:SID()
          return matchstr(expand('<sfile>'), '<SNR>\zs\d\+\ze_SID$')
        endfun

The "<SNR>" will be shown when listing functions and mappings. This is useful
to find out what they are defined to.

The |:scriptnames| command can be used to see which scripts have been sourced
and what their <SNR> number is.

This is all {not in Vi} and {not available when compiled without the |+eval|
feature}.

==
4. User-defined commands   *user-commands*

It is possible to define your own Ex commands. A user-defined command can act
just like a built-in command (it can have a range or arguments, arguments can
be completed as filenames or buffer names, etc), except that when the command
is executed, it is transformed into a normal Ex command and then executed.

For starters: See section |40.2| in the user manual.

                                        E183 *E841* *user-cmd-ambiguous*
All user defined commands must start with an uppercase letter, to avoid
confusion with builtin commands. Exceptions are these builtin commands:
        :Next
        :X
They cannot be used for a user defined command. ":Print" is also an existing
command, but it is deprecated and can be overruled.

The other characters of the user command can be uppercase letters, lowercase
letters or digits. When using digits, note that other commands that take a
numeric argument may become ambiguous. For example, the command ":Cc2" could
be the user command ":Cc2" without an argument, or the command ":Cc" with
argument "2". It is advised to put a space between the command name and the
argument to avoid these problems.

When using a user-defined command, the command can be abbreviated. However, if
an abbreviation is not unique, an error will be issued. Furthermore, a
built-in command will always take precedence.

Example: >
        :command Rename ...
        :command Renumber ...

```
        :Rena                           " Means "Rename"
        :Renu                           " Means "Renumber"
        :Ren                            " Error - ambiguous
        :command Paste ...
        :P                              " The built-in :Print
```

It is recommended that full names for user-defined commands are used in scripts.

:com[mand] *:com* *:command*
Lists all user-defined commands. When listing commands, the characters in the first two columns are
! Command has the -bang attribute
" Command has the -register attribute
b Command is local to current buffer
(see below for details on attributes)
The list can be filtered on command name with |:filter|, e.g., to list all commands with "Pyth" in the name: >
filter Pyth command

:com[mand] {cmd} List the user-defined commands that start with {cmd}

:command-verbose
When 'verbose' is non-zero, listing a command will also display where it was last defined. Example: >

```
    :verbose command TOhtml
<       Name        Args Range Complete  Definition ~
        TOhtml      0    %               :call Convert2HTML(<line1>, <line2>) ~
            Last set from /usr/share/vim/vim-7.0/plugin/tohtml.vim ~
```

See |:verbose-cmd| for more information.

E174 *E182*
:com[mand][!] [{attr}...] {cmd} {rep}
Define a user command. The name of the command is {cmd} and its replacement text is {rep}. The command's attributes (see below) are {attr}. If the command already exists, an error is reported, unless a ! is specified, in which case the command is redefined.

:delc[ommand] {cmd} *:delc* *:delcommand* *E184*
Delete the user-defined command {cmd}.

:comc[lear] *:comc* *:comclear*
Delete all user-defined commands.

Command attributes

User-defined commands are treated by Vim just like any other Ex commands. They can have arguments, or have a range specified. Arguments are subject to completion as filenames, buffers, etc. Exactly how this works depends upon the command's attributes, which are specified when the command is defined.

There are a number of attributes, split into four categories: argument handling, completion behavior, range handling, and special cases. The attributes are described below, by category.

Argument handling *E175* *E176* *:command-nargs*

By default, a user defined command will take no arguments (and an error is

```
reported if any are supplied).  However, it is possible to specify that the
command can take arguments, using the -nargs attribute.  Valid cases are:

        -nargs=0    No arguments are allowed (the default)
        -nargs=1    Exactly one argument is required, it includes spaces
        -nargs=*    Any number of arguments are allowed (0, 1, or many),
                    separated by white space
        -nargs=?    0 or 1 arguments are allowed
        -nargs=+    Arguments must be supplied, but any number are allowed

Arguments are considered to be separated by (unescaped) spaces or tabs in this
context, except when there is one argument, then the white space is part of
the argument.

Note that arguments are used as text, not as expressions.  Specifically,
"s:var" will use the script-local variable in the script where the command was
defined, not where it is invoked!  Example:
    script1.vim: >
        :let s:error = "None"
        :command -nargs=1 Error echoerr <args>
<   script2.vim: >
        :source script1.vim
        :let s:error = "Wrong!"
        :Error s:error
Executing script2.vim will result in "None" being echoed.  Not what you
intended!  Calling a function may be an alternative.

Completion behavior                                 *:command-completion* *E179*
                                        *E180* *E181* *:command-complete*
By default, the arguments of user defined commands do not undergo completion.
However, by specifying one or the other of the following attributes, argument
completion can be enabled:

        -complete=augroup       autocmd groups
        -complete=buffer        buffer names
        -complete=behave        :behave suboptions
        -complete=color         color schemes
        -complete=command       Ex command (and arguments)
        -complete=compiler      compilers
        -complete=cscope        |:cscope| suboptions
        -complete=dir           directory names
        -complete=environment   environment variable names
        -complete=event         autocommand events
        -complete=expression    Vim expression
        -complete=file          file and directory names
        -complete=file_in_path  file and directory names in |'path'|
        -complete=filetype      filetype names |'filetype'|
        -complete=function      function name
        -complete=help          help subjects
        -complete=highlight     highlight groups
        -complete=history       :history suboptions
        -complete=locale        locale names (as output of locale -a)
        -complete=mapclear      buffer argument
        -complete=mapping       mapping name
        -complete=menu          menus
        -complete=messages      |:messages| suboptions
        -complete=option        options
        -complete=packadd       optional package |pack-add| names
        -complete=shellcmd      Shell command
        -complete=sign          |:sign| suboptions
        -complete=syntax        syntax file names |'syntax'|
        -complete=syntime       |:syntime| suboptions
```

```
        -complete=tag           tags
        -complete=tag_listfiles tags, file names are shown when CTRL-D is hit
        -complete=user          user names
        -complete=var           user variables
        -complete=custom,{func} custom completion, defined via {func}
        -complete=customlist,{func} custom completion, defined via {func}

Note: That some completion methods might expand environment variables.


Custom completion                       *:command-completion-custom*
                                        *:command-completion-customlist*
                                        *E467* *E468*
It is possible to define customized completion schemes via the "custom,{func}"
or the "customlist,{func}" completion argument.  The {func} part should be a
function with the following signature: >

        :function {func}(ArgLead, CmdLine, CursorPos)

The function need not use all these arguments. The function should provide the
completion candidates as the return value.

For the "custom" argument, the function should return the completion
candidates one per line in a newline separated string.

For the "customlist" argument, the function should return the completion
candidates as a Vim List.  Non-string items in the list are ignored.

The function arguments are:
        ArgLead         the leading portion of the argument currently being
                        completed on
        CmdLine         the entire command line
        CursorPos       the cursor position in it (byte index)
The function may use these for determining context.  For the "custom"
argument, it is not necessary to filter candidates against the (implicit
pattern in) ArgLead.  Vim will filter the candidates with its regexp engine
after function return, and this is probably more efficient in most cases. For
the "customlist" argument, Vim will not filter the returned completion
candidates and the user supplied function should filter the candidates.

The following example lists user names to a Finger command >
    :com -complete=custom,ListUsers -nargs=1 Finger !finger <args>
    :fun ListUsers(A,L,P)
    :    return system("cut -d: -f1 /etc/passwd")
    :endfun

The following example completes filenames from the directories specified in
the 'path' option: >
    :com -nargs=1 -bang -complete=customlist,EditFileComplete
                        \ EditFile edit<bang> <args>
    :fun EditFileComplete(A,L,P)
    :    return split(globpath(&path, a:A), "\n")
    :endfun
<
This example does not work for file names with spaces!


Range handling                          *E177* *E178* *:command-range*
                                                      *:command-count*
By default, user-defined commands do not accept a line number range.  However,
it is possible to specify that the command does take a range (the -range
attribute), or that it takes an arbitrary count value, either in the line
```

```
number position (-range=N, like the |:split| command) or as a "count"
argument (-count=N, like the |:Next| command).  The count will then be
available in the argument with |<count>|.

Possible attributes are:

        -range      Range allowed, default is current line
        -range=%    Range allowed, default is whole file (1,$)
        -range=N    A count (default N) which is specified in the line
                    number position (like |:split|); allows for zero line
                    number.
        -count=N    A count (default N) which is specified either in the line
                    number position, or as an initial argument (like |:Next|).
                    Specifying -count (without a default) acts like -count=0

Note that -range=N and -count=N are mutually exclusive - only one should be
specified.

                                        *:command-addr*
It is possible that the special characters in the range like ., $ or % which
by default correspond to the current line, last line and the whole buffer,
relate to arguments, (loaded) buffers, windows or tab pages.

Possible values are:
        -addr=lines             Range of lines (this is the default)
        -addr=arguments         Range for arguments
        -addr=buffers           Range for buffers (also not loaded buffers)
        -addr=loaded_buffers    Range for loaded buffers
        -addr=windows           Range for windows
        -addr=tabs              Range for tab pages

Special cases                           *:command-bang* *:command-bar*
                                        *:command-register* *:command-buffer*
There are some special cases as well:

        -bang       The command can take a ! modifier (like :q or :w)
        -bar        The command can be followed by a "|" and another command.
                    A "|" inside the command argument is not allowed then.
                    Also checks for a " to start a comment.
        -register   The first argument to the command can be an optional
                    register name (like :del, :put, :yank).
        -buffer     The command will only be available in the current buffer.

In the cases of the -count and -register attributes, if the optional argument
is supplied, it is removed from the argument list and is available to the
replacement text separately.
Note that these arguments can be abbreviated, but that is a deprecated
feature.  Use the full name for new scripts.

Replacement text

The replacement text for a user defined command is scanned for special escape
sequences, using <...> notation.  Escape sequences are replaced with values
from the entered command line, and all other text is copied unchanged.  The
resulting string is executed as an Ex command.  To avoid the replacement use
<lt> in place of the initial <.  Thus to include "<bang>" literally use
"<lt>bang>".

The valid escape sequences are

                                                *<line1>*
        <line1> The starting line of the command range.
```

```
                                                *<line2>*
        <line2> The final line of the command range.
                                                *<range>*
        <range> The number of items in the command range: 0, 1 or 2
                                                *<count>*
        <count> Any count supplied (as described for the '-range'
                and '-count' attributes).
                                                *<bang>*
        <bang>  (See the '-bang' attribute) Expands to a ! if the
                command was executed with a ! modifier, otherwise
                expands to nothing.
                                                *<mods>*
        <mods>  The command modifiers, if specified. Otherwise, expands to
                nothing. Supported modifiers are |:aboveleft|, |:belowright|,
                |:botright|, |:browse|, |:confirm|, |:hide|, |:keepalt|,
                |:keepjumps|, |:keepmarks|, |:keeppatterns|, |:leftabove|,
                |:lockmarks|, |:noswapfile| |:rightbelow|, |:silent|, |:tab|,
                |:topleft|, |:verbose|, and |:vertical|.
                Note that these are not yet supported: |:noautocmd|,
                |:sandbox| and |:unsilent|.
                Examples: >
                    command! -nargs=+ -complete=file MyEdit
                                \ for f in expand(<q-args>, 0, 1) |
                                \ exe '<mods> split ' . f |
                                \ endfor

                    function! SpecialEdit(files, mods)
                        for f in expand(a:files, 0, 1)
                            exe a:mods . ' split ' . f
                        endfor
                    endfunction
                    command! -nargs=+ -complete=file Sedit
                                \ call SpecialEdit(<q-args>, <q-mods>)
<
                                                *<reg>* *<register>*
        <reg>   (See the '-register' attribute) The optional register,
                if specified.  Otherwise, expands to nothing.  <register>
                is a synonym for this.
                                                *<args>*
        <args>  The command arguments, exactly as supplied (but as
                noted above, any count or register can consume some
                of the arguments, which are then not part of <args>).
        <lt>    A single '<' (Less-Than) character.  This is needed if you
                want to get a literal copy of one of these escape sequences
                into the expansion - for example, to get <bang>, use
                <lt>bang>.

                                                        *<q-args>*
If the first two characters of an escape sequence are "q-" (for example,
<q-args>) then the value is quoted in such a way as to make it a valid value
for use in an expression.  This uses the argument as one single value.
When there is no argument <q-args> is an empty string.
                                                        *<f-args>*
To allow commands to pass their arguments on to a user-defined function, there
is a special form <f-args> ("function args").  This splits the command
arguments at spaces and tabs, quotes each argument individually, and the
<f-args> sequence is replaced by the comma-separated list of quoted arguments.
See the Mycmd example below.  If no arguments are given <f-args> is removed.
   To embed whitespace into an argument of <f-args>, prepend a backslash.
<f-args> replaces every pair of backslashes (\\) with one backslash.  A
backslash followed by a character other than white space or a backslash
remains unmodified.  Overview:
```

```
        command            <f-args> ~
        XX ab              'ab'
        XX a\b             'a\b'
        XX a\ b            'a b'
        XX a\  b           'a ', 'b'
        XX a\\b            'a\b'
        XX a\\ b           'a\', 'b'
        XX a\\\b           'a\\b'
        XX a\\\ b          'a\ b'
        XX a\\\\b          'a\\b'
        XX a\\\\ b         'a\\', 'b'

Examples >

   " Delete everything after here to the end
   :com Ddel +,$d

   " Rename the current buffer
   :com -nargs=1 -bang -complete=file Ren f <args>|w<bang>

   " Replace a range with the contents of a file
   " (Enter this all as one line)
   :com -range -nargs=1 -complete=file
         Replace <line1>-pu_|<line1>,<line2>d|r <args>|<line1>d

   " Count the number of lines in the range
   :com! -range -nargs=0 Lines  echo <line2> - <line1> + 1 "lines"

   " Call a user function (example of <f-args>)
   :com -nargs=* Mycmd call Myfunc(<f-args>)

When executed as: >
        :Mycmd arg1 arg2
This will invoke: >
        :call Myfunc("arg1","arg2")

   :" A more substantial example
   :function Allargs(command)
   :   let i = 0
   :   while i < argc()
   :      if filereadable(argv(i))
   :         execute "e " . argv(i)
   :         execute a:command
   :      endif
   :      let i = i + 1
   :   endwhile
   :endfunction
   :command -nargs=+ -complete=command Allargs call Allargs(<q-args>)

The command Allargs takes any Vim command(s) as argument and executes it on all
files in the argument list.  Usage example (note use of the "e" flag to ignore
errors and the "update" command to write modified buffers): >
        :Allargs %s/foo/bar/ge|update
This will invoke: >
        :call Allargs("%s/foo/bar/ge|update")
<
When defining a user command in a script, it will be able to call functions
local to the script and use mappings local to the script.  When the user
invokes the user command, it will run in the context of the script it was
defined in.  This matters if |<SID>| is used in a command.
```

```
 vim:tw=78:ts=8:ft=help:norl:
*tagsrch.txt*   For Vim version 8.0.  Last change: 2016 Sep 20


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Tags and special searches                              *tags-and-searches*

See section |29.1| of the user manual for an introduction.

1. Jump to a tag               |tag-commands|
2. Tag stack                   |tag-stack|
3. Tag match list              |tag-matchlist|
4. Tags details                |tag-details|
5. Tags file format            |tags-file-format|
6. Include file searches       |include-search|

==============================================================================
1. Jump to a tag                                       *tag-commands*

                                                       *tag* *tags*
A tag is an identifier that appears in a "tags" file.  It is a sort of label
that can be jumped to.  For example: In C programs each function name can be
used as a tag.  The "tags" file has to be generated by a program like ctags,
before the tag commands can be used.

With the ":tag" command the cursor will be positioned on the tag.  With the
CTRL-] command, the keyword on which the cursor is standing is used as the
tag.  If the cursor is not on a keyword, the first keyword to the right of the
cursor is used.

The ":tag" command works very well for C programs.  If you see a call to a
function and wonder what that function does, position the cursor inside of the
function name and hit CTRL-].  This will bring you to the function definition.
An easy way back is with the CTRL-T command.  Also read about the tag stack
below.

                                              *:ta* *:tag* *E426* *E429*
:[count]ta[g][!] {ident}
                        Jump to the definition of {ident}, using the
                        information in the tags file(s).  Put {ident} in the
                        tag stack.  See |tag-!| for [!].
                        {ident} can be a regexp pattern, see |tag-regexp|.
                        When there are several matching tags for {ident}, jump
                        to the [count] one.  When [count] is omitted the
                        first one is jumped to. See |tag-matchlist| for
                        jumping to other matching tags.

g<LeftMouse>                                           *g<LeftMouse>*
<C-LeftMouse>                                  *<C-LeftMouse>* *CTRL-]*
CTRL-]                  Jump to the definition of the keyword under the
                        cursor.  Same as ":tag {ident}", where {ident} is the
                        keyword under or after cursor.
                        When there are several matching tags for {ident}, jump
                        to the [count] one.  When no [count] is given the
                        first one is jumped to. See |tag-matchlist| for
                        jumping to other matching tags.
                        {Vi: identifier after the cursor}

                                                       *v_CTRL-]*
{Visual}CTRL-]          Same as ":tag {ident}", where {ident} is the text that
```

```
                        is highlighted.  {not in Vi}

                                                        *telnet-CTRL-]*
CTRL-] is the default telnet escape key.  When you type CTRL-] to jump to a
tag, you will get the telnet prompt instead.  Most versions of telnet allow
changing or disabling the default escape key.  See the telnet man page.  You
can 'telnet -E {Hostname}' to disable the escape character, or 'telnet -e
{EscapeCharacter} {Hostname}' to specify another escape character.  If
possible, try to use "ssh" instead of "telnet" to avoid this problem.

                                                        *tag-priority*
When there are multiple matches for a tag, this priority is used:
1. "FSC"  A full matching static tag for the current file.
2. "F C"  A full matching global tag for the current file.
3. "F  "  A full matching global tag for another file.
4. "FS "  A full matching static tag for another file.
5. " SC"  An ignore-case matching static tag for the current file.
6. "  C"  An ignore-case matching global tag for the current file.
7. "   "  An ignore-case matching global tag for another file.
8. " S "  An ignore-case matching static tag for another file.

Note that when the current file changes, the priority list is mostly not
changed, to avoid confusion when using ":tnext".  It is changed when using
":tag {ident}".

The ignore-case matches are not found for a ":tag" command when:
- the 'ignorecase' option is off and 'tagcase' is "followic"
- 'tagcase' is "match"
- 'tagcase' is "smart" and the pattern contains an upper case character.
- 'tagcase' is "followscs" and 'smartcase' option is on and the pattern
  contains an upper case character.

The ignore-case matches are found when:
- a pattern is used (starting with a "/")
- for ":tselect"
- when 'tagcase' is "followic" and 'ignorecase' is off
- when 'tagcase' is "match"
- when 'tagcase' is "followscs" and the 'smartcase' option is off

Note that using ignore-case tag searching disables binary searching in the
tags file, which causes a slowdown.  This can be avoided by fold-case sorting
the tag file. See the 'tagbsearch' option for an explanation.

==============================================================================
2. Tag stack                            *tag-stack* *tagstack* *E425*

On the tag stack is remembered which tags you jumped to, and from where.
Tags are only pushed onto the stack when the 'tagstack' option is set.

g<RightMouse>                                           *g<RightMouse>*
<C-RightMouse>                                  *<C-RightMouse>* *CTRL-T*
CTRL-T                  Jump to [count] older entry in the tag stack
                        (default 1).  {not in Vi}

                                                *:po* *:pop* *E555* *E556*
:[count]po[p][!]        Jump to [count] older entry in tag stack (default 1).
                        See |tag-!| for [!].  {not in Vi}

:[count]ta[g][!]        Jump to [count] newer entry in tag stack (default 1).
                        See |tag-!| for [!].  {not in Vi}

                                                        *:tags*
```

```
:tags                   Show the contents of the tag stack.  The active
                        entry is marked with a '>'.  {not in Vi}

The output of ":tags" looks like this:

   # TO tag      FROM line  in file/text
   1  1 main             1  harddisk2:text/vim/test
 > 2  2 FuncA          58  i = FuncA(10);
   3  1 FuncC         357  harddisk2:text/vim/src/amiga.c

This list shows the tags that you jumped to and the cursor position before
that jump.  The older tags are at the top, the newer at the bottom.

The '>' points to the active entry.  This is the tag that will be used by the
next ":tag" command.  The CTRL-T and ":pop" command will use the position
above the active entry.

Below the "TO" is the number of the current match in the match list.  Note
that this doesn't change when using ":pop" or ":tag".

The line number and file name are remembered to be able to get back to where
you were before the tag command.  The line number will be correct, also when
deleting/inserting lines, unless this was done by another program (e.g.
another instance of Vim).

For the current file, the "file/text" column shows the text at the position.
An indent is removed and a long line is truncated to fit in the window.

You can jump to previously used tags with several commands.  Some examples:

        ":pop" or CTRL-T        to position before previous tag
        {count}CTRL-T           to position before {count} older tag
        ":tag"                  to newer tag
        ":0tag"                 to last used tag

The most obvious way to use this is while browsing through the call graph of
a program.  Consider the following call graph:

        main  --->  FuncA  --->  FuncC
              --->  FuncB

(Explanation: main calls FuncA and FuncB; FuncA calls FuncC).
You can get from main to FuncA by using CTRL-] on the call to FuncA.  Then
you can CTRL-] to get to FuncC.  If you now want to go back to main you can
use CTRL-T twice.  Then you can CTRL-] to FuncB.

If you issue a ":ta {ident}" or CTRL-] command, this tag is inserted at the
current position in the stack.  If the stack was full (it can hold up to 20
entries), the oldest entry is deleted and the older entries shift one
position up (their index number is decremented by one).  If the last used
entry was not at the bottom, the entries below the last used one are
deleted.  This means that an old branch in the call graph is lost.  After the
commands explained above the tag stack will look like this:

   # TO tag     FROM line  in file/text
   1  1 main            1  harddisk2:text/vim/test
   2  1 FuncB          59  harddisk2:text/vim/src/main.c

                                                           *E73*
When you try to use the tag stack while it doesn't contain anything you will
get an error message.
```

```
==============================================================================
3. Tag match list                                   *tag-matchlist* *E427* *E428*

When there are several matching tags, these commands can be used to jump
between them.  Note that these commands don't change the tag stack, they keep
the same entry.

                                                            *:ts* *:tselect*
:ts[elect][!] [ident]   List the tags that match [ident], using the
                        information in the tags file(s).
                        When [ident] is not given, the last tag name from the
                        tag stack is used.
                        With a '>' in the first column is indicated which is
                        the current position in the list (if there is one).
                        [ident] can be a regexp pattern, see |tag-regexp|.
                        See |tag-priority| for the priorities used in the
                        listing.  {not in Vi}
                        Example output:

>
         nr pri kind tag               file
          1 F   f    mch_delay         os_amiga.c
                        mch_delay(msec, ignoreinput)
        > 2 F   f    mch_delay         os_msdos.c
                        mch_delay(msec, ignoreinput)
          3 F   f    mch_delay         os_unix.c
                        mch_delay(msec, ignoreinput)
        Enter nr of choice (<CR> to abort):
<
                        See |tag-priority| for the "pri" column.  Note that
                        this depends on the current file, thus using
                        ":tselect xxx" can produce different results.
                        The "kind" column gives the kind of tag, if this was
                        included in the tags file.
                        The "info" column shows information that could be
                        found in the tags file.  It depends on the program
                        that produced the tags file.
                        When the list is long, you may get the |more-prompt|.
                        If you already see the tag you want to use, you can
                        type 'q' and enter the number.

                                                            *:sts* *:stselect*
:sts[elect][!] [ident]  Does ":tselect[!] [ident]" and splits the window for
                        the selected tag.  {not in Vi}

                                                            *g]*
g]                      Like CTRL-], but use ":tselect" instead of ":tag".
                        {not in Vi}

                                                            *v_g]*
{Visual}g]              Same as "g]", but use the highlighted text as the
                        identifier.  {not in Vi}

                                                            *:tj* *:tjump*
:tj[ump][!] [ident]     Like ":tselect", but jump to the tag directly when
                        there is only one match.  {not in Vi}

                                                            *:stj* *:stjump*
:stj[ump][!] [ident]    Does ":tjump[!] [ident]" and splits the window for the
                        selected tag.  {not in Vi}

                                                            *g_CTRL-]*
```

```
g CTRL-]                Like CTRL-], but use ":tjump" instead of ":tag".
                        {not in Vi}

                                                        *v_g_CTRL-]*
{Visual}g CTRL-]        Same as "g CTRL-]", but use the highlighted text as
                        the identifier.  {not in Vi}

                                                        *:tn* *:tnext*
:[count]tn[ext][!]      Jump to [count] next matching tag (default 1).  See
                        |tag-!| for [!].  {not in Vi}

                                                        *:tp* *:tprevious*
:[count]tp[revious][!]  Jump to [count] previous matching tag (default 1).
                        See |tag-!| for [!].  {not in Vi}

                                                        *:tN* *:tNext*
:[count]tN[ext][!]      Same as ":tprevious".  {not in Vi}

                                                        *:tr* *:trewind*
:[count]tr[ewind][!]    Jump to first matching tag.  If [count] is given, jump
                        to [count]th matching tag.  See |tag-!| for [!].  {not
                        in Vi}

                                                        *:tf* *:tfirst*
:[count]tf[irst][!]     Same as ":trewind".  {not in Vi}

                                                        *:tl* *:tlast*
:tl[ast][!]             Jump to last matching tag.  See |tag-!| for [!].  {not
                        in Vi}

                                                        *:lt* *:ltag*
:lt[ag][!] [ident]      Jump to tag [ident] and add the matching tags to a new
                        location list for the current window.  [ident] can be
                        a regexp pattern, see |tag-regexp|.  When [ident] is
                        not given, the last tag name from the tag stack is
                        used.  The search pattern to locate the tag line is
                        prefixed with "\V" to escape all the special
                        characters (very nomagic). The location list showing
                        the matching tags is independent of the tag stack.
                        See |tag-!| for [!].
                        {not in Vi}

When there is no other message, Vim shows which matching tag has been jumped
to, and the number of matching tags: >
        tag 1 of 3 or more
The " or more" is used to indicate that Vim didn't try all the tags files yet.
When using ":tnext" a few times, or with ":tlast", more matches may be found.

When you didn't see this message because of some other message, or you just
want to know where you are, this command will show it again (and jump to the
same tag as last time): >
        :0tn
<
                                                        *tag-skip-file*
When a matching tag is found for which the file doesn't exist, this match is
skipped and the next matching tag is used.  Vim reports this, to notify you of
missing files.  When the end of the list of matches has been reached, an error
message is given.

                                                        *tag-preview*
The tag match list can also be used in the preview window.  The commands are
the same as above, with a "p" prepended.
```

{not available when compiled without the |+quickfix| feature}

:pts *:ptselect*
:pts[elect][!] [ident] Does ":tselect[!] [ident]" and shows the new tag in a "Preview" window. See |:ptag| for more info. {not in Vi}

:ptj *:ptjump*
:ptj[ump][!] [ident] Does ":tjump[!] [ident]" and shows the new tag in a "Preview" window. See |:ptag| for more info. {not in Vi}

:ptn *:ptnext*
:[count]ptn[ext][!] ":tnext" in the preview window. See |:ptag|. {not in Vi}

:ptp *:ptprevious*
:[count]ptp[revious][!] ":tprevious" in the preview window. See |:ptag|. {not in Vi}

:ptN *:ptNext*
:[count]ptN[ext][!] Same as ":ptprevious". {not in Vi}

:ptr *:ptrewind*
:[count]ptr[ewind][!] ":trewind" in the preview window. See |:ptag|. {not in Vi}

:ptf *:ptfirst*
:[count]ptf[irst][!] Same as ":ptrewind". {not in Vi}

:ptl *:ptlast*
:ptl[ast][!] ":tlast" in the preview window. See |:ptag|. {not in Vi}

==
4. Tags details *tag-details*

static-tag
A static tag is a tag that is defined for a specific file. In a C program this could be a static function.

In Vi jumping to a tag sets the current search pattern. This means that the "n" command after jumping to a tag does not search for the same pattern that it did before jumping to the tag. Vim does not do this as we consider it to be a bug. You can still find the tag search pattern in the search history. If you really want the old Vi behavior, set the 't' flag in 'cpoptions'.

tag-binary-search
Vim uses binary searching in the tags file to find the desired tag quickly (when enabled at compile time |+tag_binary|). But this only works if the tags file was sorted on ASCII byte value. Therefore, if no match was found, another try is done with a linear search. If you only want the linear search, reset the 'tagbsearch' option. Or better: Sort the tags file!

Note that the binary searching is disabled when not looking for a tag with a specific name. This happens when ignoring case and when a regular expression is used that doesn't start with a fixed string. Tag searching can be a lot slower then. The former can be avoided by case-fold sorting the tags file. See 'tagbsearch' for details.

tag-regexp
The ":tag" and ":tselect" commands accept a regular expression argument. See

```
|pattern| for the special characters that can be used.
When the argument starts with '/', it is used as a pattern.  If the argument
does not start with '/', it is taken literally, as a full tag name.
Examples: >
    :tag main
<       jumps to the tag "main" that has the highest priority. >
    :tag /^get
<       jumps to the tag that starts with "get" and has the highest priority. >
    :tag /norm
<       lists all the tags that contain "norm", including "id_norm".
When the argument both exists literally, and match when used as a regexp, a
literal match has a higher priority.  For example, ":tag /open" matches "open"
before "open_file" and "file_open".
When using a pattern case is ignored.  If you want to match case use "\C" in
the pattern.

                                                        *tag-!*
If the tag is in the current file this will always work.  Otherwise the
performed actions depend on whether the current file was changed, whether a !
is added to the command and on the 'autowrite' option:

  tag in       file        autowrite                    ~
current file  changed   !   option        action       ~
-------------------------------------------------------------------------
    yes          x      x     x   goto tag
    no           no     x     x   read other file, goto tag
    no          yes    yes    x   abandon current file, read other file, goto
                                   tag
    no          yes     no    on  write current file, read other file, goto
                                   tag
    no          yes     no   off  fail
-------------------------------------------------------------------------

- If the tag is in the current file, the command will always work.
- If the tag is in another file and the current file was not changed, the
  other file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and a ! is
  added to the command, the changes to the current file are lost, the other
  file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and the
  'autowrite' option is on, the current file will be written, the other
  file will be made the current file and read into the buffer.
- If the tag is in another file, the current file was changed and the
  'autowrite' option is off, the command will fail.  If you want to save
  the changes, use the ":w" command and then use ":tag" without an argument.
  This works because the tag is put on the stack anyway.  If you want to lose
  the changes you can use the ":tag!" command.

                                                        *tag-security*
Note that Vim forbids some commands, for security reasons.  This works like
using the 'secure' option for exrc/vimrc files in the current directory.  See
|trojan-horse| and |sandbox|.
When the {tagaddress} changes a buffer, you will get a warning message:
        "WARNING: tag command changed a buffer!!!"
In a future version changing the buffer will be impossible.  All this for
security reasons: Somebody might hide a nasty command in the tags file, which
would otherwise go unnoticed.  Example: >
        :$d|/tag-function-name/
{this security prevention is not present in Vi}

In Vi the ":tag" command sets the last search pattern when the tag is searched
for.  In Vim this is not done, the previous search pattern is still remembered,
```

unless the 't' flag is present in 'cpoptions'. The search pattern is always put in the search history, so you can modify it if searching fails.

emacs-tags *emacs_tags* *E430*

Emacs style tag files are only supported if Vim was compiled with the |+emacs_tags| feature enabled. Sorry, there is no explanation about Emacs tag files here, it is only supported for backwards compatibility :-).

Lines in Emacs tags files can be very long. Vim only deals with lines of up to about 510 bytes. To see whether lines are ignored set 'verbose' to 5 or higher.

tags-option

The 'tags' option is a list of file names. Each of these files is searched for the tag. This can be used to use a different tags file than the default file "tags". It can also be used to access a common tags file.

The next file in the list is not used when:
- A matching static tag for the current buffer has been found.
- A matching global tag has been found.

This also depends on whether case is ignored. Case is ignored when:
- 'tagcase' is "followic" and 'ignorecase' is set
- 'tagcase' is "ignore"
- 'tagcase' is "smart" and the pattern only contains lower case characters.
- 'tagcase' is "followscs" and 'smartcase' is set and the pattern only contains lower case characters.

If case is not ignored, and the tags file only has a match without matching case, the next tags file is searched for a match with matching case. If no tag with matching case is found, the first match without matching case is used. If case is ignored, and a matching global tag with or without matching case is found, this one is used, no further tags files are searched.

When a tag file name starts with "./", the '.' is replaced with the path of the current file. This makes it possible to use a tags file in the directory where the current file is (no matter what the current directory is). The idea of using "./" is that you can define which tag file is searched first: In the current directory ("tags,./tags") or in the directory of the current file ("./tags,tags").

For example: >
```
:set tags=./tags,tags,/home/user/commontags
```

In this example the tag will first be searched for in the file "tags" in the directory where the current file is. Next the "tags" file in the current directory. If it is not found there, then the file "/home/user/commontags" will be searched for the tag.

This can be switched off by including the 'd' flag in 'cpoptions', to make it Vi compatible. "./tags" will then be the tags file in the current directory, instead of the tags file in the directory where the current file is.

Instead of the comma a space may be used. Then a backslash is required for the space to be included in the string option: >
```
:set tags=tags\ /home/user/commontags
```

To include a space in a file name use three backslashes. To include a comma in a file name use two backslashes. For example, use: >
```
:set tags=tag\\\ file,/home/user/common\\,tags
```

for the files "tag file" and "/home/user/common,tags". The 'tags' option will

have the value "tag\ file,/home/user/common\,tags".

If the 'tagrelative' option is on (which is the default) and using a tag file in another directory, file names in that tag file are relative to the directory where the tag file is.

==
5. Tags file format *tags-file-format* *E431*

ctags *jtags*
A tags file can be created with an external command, for example "ctags". It will contain a tag for each function. Some versions of "ctags" will also make a tag for each "#defined" macro, typedefs, enums, etc.

Some programs that generate tags files:

ctags	As found on most Unix systems. Only supports C. Only does the basic work.
	Exuberant_ctags
exuberant ctags	This a very good one. It works for C, C++, Java, Fortran, Eiffel and others. It can generate tags for many items. See http://ctags.sourceforge.net.
etags	Connected to Emacs. Supports many languages.
JTags	For Java, in Java. It can be found at http://www.fleiner.com/jtags/.
ptags.py	For Python, in Python. Found in your Python source directory at Tools/scripts/ptags.py.
ptags	For Perl, in Perl. It can be found at http://www.eleves.ens.fr:8080/home/nthiery/Tags/.
gnatxref	For Ada. See http://www.gnuada.org/. gnatxref is part of the gnat package.

The lines in the tags file must have one of these three formats:

1. {tagname} {TAB} {tagfile} {TAB} {tagaddress}
2. {tagfile}:{tagname} {TAB} {tagfile} {TAB} {tagaddress}
3. {tagname} {TAB} {tagfile} {TAB} {tagaddress} {term} {field} ..

The first is a normal tag, which is completely compatible with Vi. It is the only format produced by traditional ctags implementations. This is often used for functions that are global, also referenced in other files.

The lines in the tags file can end in <LF> or <CR><LF>. On the Macintosh <CR> also works. The <CR> and <NL> characters can never appear inside a line.

tag-old-static
The second format is for a static tag only. It is obsolete now, replaced by the third format. It is only supported by Elvis 1.x and Vim and a few versions of ctags. A static tag is often used for functions that are local, only referenced in the file {tagfile}. Note that for the static tag, the two occurrences of {tagfile} must be exactly the same. Also see |tags-option| below, for how static tags are used.

The third format is new. It includes additional information in optional fields at the end of each line. It is backwards compatible with Vi. It is only supported by new versions of ctags (such as Exuberant ctags).

{tagname}	The identifier. Normally the name of a function, but it can be any identifier. It cannot contain a <Tab>.
{TAB}	One <Tab> character. Note: previous versions allowed any white space here. This has been abandoned to allow spaces in {tagfile}. It can be re-enabled by including the

```
                |+tag_any_white| feature at compile time. *tag-any-white*
{tagfile}       The file that contains the definition of {tagname}.  It can
                have an absolute or relative path.  It may contain environment
                variables and wildcards (although the use of wildcards is
                doubtful).  It cannot contain a <Tab>.
{tagaddress}    The Ex command that positions the cursor on the tag.  It can
                be any Ex command, although restrictions apply (see
                |tag-security|).  Posix only allows line numbers and search
                commands, which are mostly used.
{term}          ;" The two characters semicolon and double quote.  This is
                interpreted by Vi as the start of a comment, which makes the
                following be ignored.  This is for backwards compatibility
                with Vi, it ignores the following fields.
{field} ..      A list of optional fields.  Each field has the form:

                        <Tab>{fieldname}:{value}

                The {fieldname} identifies the field, and can only contain
                alphabetical characters [a-zA-Z].
                The {value} is any string, but cannot contain a <Tab>.
                These characters are special:
                        "\t" stands for a <Tab>
                        "\r" stands for a <CR>
                        "\n" stands for a <NL>
                        "\\" stands for a single '\' character

                There is one field that doesn't have a ':'.  This is the kind
                of the tag.  It is handled like it was preceded with "kind:".
                See the documentation of ctags for the kinds it produces.

                The only other field currently recognized by Vim is "file:"
                (with an empty value).  It is used for a static tag.

The first lines in the tags file can contain lines that start with
        !_TAG_
These are sorted to the first lines, only rare tags that start with "!" can
sort to before them.  Vim recognizes two items.  The first one is the line
that indicates if the file was sorted.  When this line is found, Vim uses
binary searching for the tags file:
        !_TAG_FILE_SORTED<Tab>1<Tab>{anything} ~

A tag file may be case-fold sorted to avoid a linear search when case is
ignored.  (Case is ignored when 'ignorecase' is set and 'tagcase' is
"followic", or when 'tagcase' is "ignore".)  See 'tagbsearch' for details.
The value '2' should be used then:
        !_TAG_FILE_SORTED<Tab>2<Tab>{anything} ~

The other tag that Vim recognizes, but only when compiled with the
|+multi_byte| feature, is the encoding of the tags file:
        !_TAG_FILE_ENCODING<Tab>utf-8<Tab>{anything} ~
Here "utf-8" is the encoding used for the tags.  Vim will then convert the tag
being searched for from 'encoding' to the encoding of the tags file.  And when
listing tags the reverse happens.  When the conversion fails the unconverted
tag is used.

                                                        *tag-search*
The command can be any Ex command, but often it is a search command.
Examples:
        tag1    file1   /^main(argc, argv)/ ~
        tag2    file2   108 ~

The command is always executed with 'magic' not set.  The only special
```

characters in a search pattern are "^" (begin-of-line) and "$" (<EOL>).
See |pattern|. Note that you must put a backslash before each backslash in
the search text. This is for backwards compatibility with Vi.

E434 *E435*
If the command is a normal search command (it starts and ends with "/" or
"?"), some special handling is done:
- Searching starts on line 1 of the file.
  The direction of the search is forward for "/", backward for "?".
  Note that 'wrapscan' does not matter, the whole file is always searched. (Vi
  does use 'wrapscan', which caused tags sometimes not be found.) {Vi starts
  searching in line 2 of another file. It does not find a tag in line 1 of
  another file when 'wrapscan' is not set}
- If the search fails, another try is done ignoring case. If that fails too,
  a search is done for:
        "^tagname[\t]*("
  (the tag with '^' prepended and "[\t]*(" appended). When using function
  names, this will find the function name when it is in column 0. This will
  help when the arguments to the function have changed since the tags file was
  made. If this search also fails another search is done with:
        "^[#a-zA-Z_].*\<tagname[\t]*("
  This means: A line starting with '#' or an identifier and containing the tag
  followed by white space and a '('. This will find macro names and function
  names with a type prepended. {the extra searches are not in Vi}

==
6. Include file searches   *include-search* *definition-search*
                                                E387 *E388* *E389*

These commands look for a string in the current file and in all encountered
included files (recursively). This can be used to find the definition of a
variable, function or macro. If you only want to search in the current
buffer, use the commands listed at |pattern-searches|.

These commands are not available when the |+find_in_path| feature was disabled
at compile time.

When a line is encountered that includes another file, that file is searched
before continuing in the current buffer. Files included by included files are
also searched. When an include file could not be found it is silently
ignored. Use the |:checkpath| command to discover which files could not be
found, possibly your 'path' option is not set up correctly. Note: the
included file is searched, not a buffer that may be editing that file. Only
for the current file the lines in the buffer are used.

The string can be any keyword or a defined macro. For the keyword any match
will be found. For defined macros only lines that match with the 'define'
option will be found. The default is "^#\s*define", which is for C programs.
For other languages you probably want to change this. See 'define' for an
example for C++. The string cannot contain an end-of-line, only matches
within a line are found.

When a match is found for a defined macro, the displaying of lines continues
with the next line when a line ends in a backslash.

The commands that start with "[" start searching from the start of the current
file. The commands that start with "]" start at the current cursor position.

The 'include' option is used to define a line that includes another file. The
default is "\^#\s*include", which is for C programs. Note: Vim does not
recognize C syntax, if the 'include' option matches a line inside
"#ifdef/#endif" or inside a comment, it is searched anyway. The 'isfname'

option is used to recognize the file name that comes after the matched
pattern.

The 'path' option is used to find the directory for the include files that
do not have an absolute path.

The 'comments' option is used for the commands that display a single line or
jump to a line. It defines patterns that may start a comment. Those lines
are ignored for the search, unless [!] is used. One exception: When the line
matches the pattern "^# *define" it is not considered to be a comment.

If you want to list matches, and then select one to jump to, you could use a
mapping to do that for you. Here is an example: >

```
  :map <F4> [I:let nr = input("Which one: ")<Bar>exe "normal " . nr ."[\t"<CR>
<
```

[i
[i Display the first line that contains the keyword under the cursor. The search starts at the beginning of the file. Lines that look like a comment are ignored (see 'comments' option). If a count is given, the count'th matching line is displayed, and comment lines are not ignored. {not in Vi}

]i
]i like "[i", but start at the current cursor position. {not in Vi}

:is *:isearch*
:[range]is[earch][!] [count] [/]pattern[/]
Like "[i" and "]i", but search in [range] lines (default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}

[I
[I Display all lines that contain the keyword under the cursor. Filenames and line numbers are displayed for the found lines. The search starts at the beginning of the file. {not in Vi}

]I
]I like "[I", but start at the current cursor position. {not in Vi}

:il *:ilist*
:[range]il[ist][!] [/]pattern[/]
Like "[I" and "]I", but search in [range] lines (default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}

[_CTRL-I
[CTRL-I Jump to the first line that contains the keyword under the cursor. The search starts at the beginning of the file. Lines that look like a comment are ignored (see 'comments' option). If a count is given, the count'th matching line is jumped to, and comment lines are not ignored. {not in Vi}

]_CTRL-I
] CTRL-I like "[CTRL-I", but start at the current cursor position. {not in Vi}

:ij *:ijump*
:[range]ij[ump][!] [count] [/]pattern[/]
Like "[CTRL-I" and "] CTRL-I", but search in
[range] lines (default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}

CTRL-W CTRL-I *CTRL-W_CTRL-I* *CTRL-W_i*
CTRL-W i Open a new window, with the cursor on the first line
that contains the keyword under the cursor. The
search starts at the beginning of the file. Lines
that look like a comment line are ignored (see
'comments' option). If a count is given, the count'th
matching line is jumped to, and comment lines are not
ignored. {not in Vi}

:isp *:isplit*
:[range]isp[lit][!] [count] [/]pattern[/]
Like "CTRL-W i" and "CTRL-W i", but search in
[range] lines (default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}

[d
[d Display the first macro definition that contains the
macro under the cursor. The search starts from the
beginning of the file. If a count is given, the
count'th matching line is displayed. {not in Vi}

]d
]d like "[d", but start at the current cursor position.
{not in Vi}

:ds *:dsearch*
:[range]ds[earch][!] [count] [/]string[/]
Like "[d" and "]d", but search in [range] lines
(default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}

[D
[D Display all macro definitions that contain the macro
under the cursor. Filenames and line numbers are
displayed for the found lines. The search starts
from the beginning of the file. {not in Vi}

]D
]D like "[D", but start at the current cursor position.
{not in Vi}

:dli *:dlist*
:[range]dli[st][!] [/]string[/]
Like `[D` and `]D`, but search in [range] lines
(default: whole file).
See |:search-args| for [/] and [!]. {not in Vi}
Note that `:dl` works like `:delete` with the "l"
flag, not `:dlist`.

[_CTRL-D
[CTRL-D Jump to the first macro definition that contains the
keyword under the cursor. The search starts from
the beginning of the file. If a count is given, the
count'th matching line is jumped to. {not in Vi}

]_CTRL-D

```
] CTRL-D                like "[ CTRL-D", but start at the current cursor
                        position.  {not in Vi}

                                                        *:dj* *:djump*
:[range]dj[ump][!] [count] [/]string[/]
                        Like "[ CTRL-D"  and "] CTRL-D", but search  in
                        [range] lines (default: whole file).
                        See |:search-args| for [/] and [!].  {not in Vi}

CTRL-W CTRL-D                                   *CTRL-W_CTRL-D* *CTRL-W_d*
CTRL-W d                Open a new window, with the cursor on the first
                        macro definition line that contains the keyword
                        under the cursor.  The search starts from the
                        beginning of the file.  If a count is given, the
                        count'th matching line is jumped to.  {not in Vi}

                                                        *:dsp* *:dsplit*
:[range]dsp[lit][!] [count] [/]string[/]
                        Like "CTRL-W d", but search in [range] lines
                        (default: whole file).
                        See |:search-args| for [/] and [!].  {not in Vi}

                                                        *:che* *:checkpath*
:che[ckpath]            List all the included files that could not be found.
                        {not in Vi}

:che[ckpath]!           List all the included files.  {not in Vi}

                                                                *:search-args*
Common arguments for the commands above:
[!]     When included, find matches in lines that are recognized as comments.
        When excluded, a match is ignored when the line is recognized as a
        comment (according to 'comments'), or the match is in a C comment
        (after "//" or inside /* */).  Note that a match may be missed if a
        line is recognized as a comment, but the comment ends halfway the line.
        And if the line is a comment, but it is not recognized (according to
        'comments') a match may be found in it anyway.  Example: >
                /* comment
                   foobar */
<       A match for "foobar" is found, because this line is not recognized as
        a comment (even though syntax highlighting does recognize it).
        Note: Since a macro definition mostly doesn't look like a comment, the
        [!] makes no difference for ":dlist", ":dsearch" and ":djump".
[/]     A pattern can be surrounded by '/'.  Without '/' only whole words are
        matched, using the pattern "\<pattern\>".  Only after the second '/' a
        next command can be appended with '|'.  Example: >
        :isearch /string/ | echo "the last one"
<       For a ":djump", ":dsplit", ":dlist" and ":dsearch" command the pattern
        is used as a literal string, not as a search pattern.

 vim:tw=78:ts=8:ft=help:norl:
*quickfix.txt*  For Vim version 8.0.  Last change: 2017 Sep 13


                  VIM REFERENCE MANUAL    by Bram Moolenaar


This subject is introduced in section |30.1| of the user manual.

1. Using QuickFix commands              |quickfix|
2. The error window                     |quickfix-window|
3. Using more than one list of errors   |quickfix-error-lists|
```

4. Using :make |:make_makeprg|
5. Using :grep |grep|
6. Selecting a compiler |compiler-select|
7. The error format |error-file-format|
8. The directory stack |quickfix-directory-stack|
9. Specific error file formats |errorformats|

{Vi does not have any of these commands}

The quickfix commands are not available when the |+quickfix| feature was
disabled at compile time.

==
1. Using QuickFix commands *quickfix* *Quickfix* *E42*

Vim has a special mode to speedup the edit-compile-edit cycle. This is
inspired by the quickfix option of the Manx's Aztec C compiler on the Amiga.
The idea is to save the error messages from the compiler in a file and use Vim
to jump to the errors one by one. You can examine each problem and fix it,
without having to remember all the error messages.

In Vim the quickfix commands are used more generally to find a list of
positions in files. For example, |:vimgrep| finds pattern matches. You can
use the positions in a script with the |getqflist()| function. Thus you can
do a lot more than the edit/compile/fix cycle!

If you have the error messages in a file you can start Vim with: >
vim -q filename

From inside Vim an easy way to run a command and handle the output is with the
|:make| command (see below).

The 'errorformat' option should be set to match the error messages from your
compiler (see |errorformat| below).

quickfix-ID
Each quickfix list has a unique identifier called the quickfix ID and this
number will not change within a Vim session. The getqflist() function can be
used to get the identifier assigned to a list. There is also a quickfix list
number which may change whenever more than ten lists are added to a quickfix
stack.

location-list *E776*
A location list is a window-local quickfix list. You get one after commands
like `:lvimgrep`, `:lgrep`, `:lhelpgrep`, `:lmake`, etc., which create a
location list instead of a quickfix list as the corresponding `:vimgrep`,
`:grep`, `:helpgrep`, `:make` do.
A location list is associated with a window and each window can have a
separate location list. A location list can be associated with only one
window. The location list is independent of the quickfix list.

When a window with a location list is split, the new window gets a copy of the
location list. When there are no longer any references to a location list,
the location list is destroyed.

The following quickfix commands can be used. The location list commands are
similar to the quickfix commands, replacing the 'c' prefix in the quickfix
command with 'l'.

E924
If the current window was closed by an |autocommand| while processing a
location list command, it will be aborted.

E925 *E926*
If the current quickfix or location list was changed by an |autocommand| while
processing a quickfix or location list command, it will be aborted.

:cc
:cc[!] [nr] Display error [nr]. If [nr] is omitted, the same error is displayed again. Without [!] this doesn't work when jumping to another buffer, the current buffer has been changed, there is the only window for the buffer and both 'hidden' and 'autowrite' are off. When jumping to another buffer with [!] any changes to the current buffer are lost, unless 'hidden' is set or there is another window for this buffer. The 'switchbuf' settings are respected when jumping to a buffer.

:ll
:ll[!] [nr] Same as ":cc", except the location list for the current window is used instead of the quickfix list.

:cn *:cnext* *E553*
:[count]cn[ext][!] Display the [count] next error in the list that includes a file name. If there are no file names at all, go to the [count] next error. See |:cc| for [!] and 'switchbuf'.

:lne *:lnext*
:[count]lne[xt][!] Same as ":cnext", except the location list for the current window is used instead of the quickfix list.

:[count]cN[ext][!] *:cp* *:cprevious* *:cN* *:cNext*
:[count]cp[revious][!] Display the [count] previous error in the list that includes a file name. If there are no file names at all, go to the [count] previous error. See |:cc| for [!] and 'switchbuf'.

:[count]lN[ext][!] *:lp* *:lprevious* *:lN* *:lNext*
:[count]lp[revious][!] Same as ":cNext" and ":cprevious", except the location list for the current window is used instead of the quickfix list.

:cnf *:cnfile*
:[count]cnf[ile][!] Display the first error in the [count] next file in the list that includes a file name. If there are no file names at all or if there is no next file, go to the [count] next error. See |:cc| for [!] and 'switchbuf'.

:lnf *:lnfile*
:[count]lnf[ile][!] Same as ":cnfile", except the location list for the current window is used instead of the quickfix list.

:[count]cNf[ile][!] *:cpf* *:cpfile* *:cNf* *:cNfile*
:[count]cpf[ile][!] Display the last error in the [count] previous file in the list that includes a file name. If there are no file names at all or if there is no next file, go to the [count] previous error. See |:cc| for [!] and 'switchbuf'.

```
:[count]lNf[ile][!]                    *:lpf* *:lpfile* *:lNf* *:lNfile*
:[count]lpf[ile][!]     Same as ":cNfile" and ":cpfile", except the location
                        list for the current window is used instead of the
                        quickfix list.

                                                        *:crewind* *:cr*
:cr[ewind][!] [nr]      Display error [nr].  If [nr] is omitted, the FIRST
                        error is displayed.  See |:cc|.

                                                        *:lrewind* *:lr*
:lr[ewind][!] [nr]      Same as ":crewind", except the location list for the
                        current window is used instead of the quickfix list.

                                                        *:cfirst* *:cfir*
:cfir[st][!] [nr]       Same as ":crewind".

                                                        *:lfirst* *:lfir*
:lfir[st][!] [nr]       Same as ":lrewind".

                                                        *:clast* *:cla*
:cla[st][!] [nr]        Display error [nr].  If [nr] is omitted, the LAST
                        error is displayed.  See |:cc|.

                                                        *:llast* *:lla*
:lla[st][!] [nr]        Same as ":clast", except the location list for the
                        current window is used instead of the quickfix list.

                                                        *:cq* *:cquit*
:cq[uit][!]             Quit Vim with an error code, so that the compiler
                        will not compile the same file again.
                        WARNING: All changes in files are lost!  Also when the
                        [!] is not used.  It works like ":qall!" |:qall|,
                        except that Vim returns a non-zero exit code.

                                                        *:cf* *:cfile*
:cf[ile][!] [errorfile] Read the error file and jump to the first error.
                        This is done automatically when Vim is started with
                        the -q option.  You can use this command when you
                        keep Vim running while compiling.  If you give the
                        name of the errorfile, the 'errorfile' option will
                        be set to [errorfile].  See |:cc| for [!].
                        If the encoding of the error file differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.

                                                        *:lf* *:lfile*
:lf[ile][!] [errorfile] Same as ":cfile", except the location list for the
                        current window is used instead of the quickfix list.
                        You can not use the -q command-line option to set
                        the location list.


:cg[etfile] [errorfile]                                 *:cg* *:cgetfile*
                        Read the error file.  Just like ":cfile" but don't
                        jump to the first error.
                        If the encoding of the error file differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.


:lg[etfile] [errorfile]                                 *:lg* *:lgetfile*
                        Same as ":cgetfile", except the location list for the
```

```
                        current window is used instead of the quickfix list.

                                                    *:caddf* *:caddfile*
:caddf[ile] [errorfile] Read the error file and add the errors from the
                        errorfile to the current quickfix list. If a quickfix
                        list is not present, then a new list is created.
                        If the encoding of the error file differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.

                                                    *:laddf* *:laddfile*
:laddf[ile] [errorfile] Same as ":caddfile", except the location list for the
                        current window is used instead of the quickfix list.

                                                *:cb* *:cbuffer* *E681*
:cb[uffer][!] [bufnr]   Read the error list from the current buffer.
                        When [bufnr] is given it must be the number of a
                        loaded buffer.  That buffer will then be used instead
                        of the current buffer.
                        A range can be specified for the lines to be used.
                        Otherwise all lines in the buffer are used.
                        See |:cc| for [!].

                                                *:lb* *:lbuffer*
:lb[uffer][!] [bufnr]   Same as ":cbuffer", except the location list for the
                        current window is used instead of the quickfix list.

                                                *:cgetb* *:cgetbuffer*
:cgetb[uffer] [bufnr]   Read the error list from the current buffer.  Just
                        like ":cbuffer" but don't jump to the first error.

                                                *:lgetb* *:lgetbuffer*
:lgetb[uffer] [bufnr]   Same as ":cgetbuffer", except the location list for
                        the current window is used instead of the quickfix
                        list.

                                                    *:cad* *:caddbuffer*
:cad[dbuffer] [bufnr]   Read the error list from the current buffer and add
                        the errors to the current quickfix list.  If a
                        quickfix list is not present, then a new list is
                        created. Otherwise, same as ":cbuffer".

                                                    *:laddb* *:laddbuffer*
:laddb[uffer] [bufnr]   Same as ":caddbuffer", except the location list for
                        the current window is used instead of the quickfix
                        list.

                                                    *:cex* *:cexpr* *E777*
:cex[pr][!] {expr}      Create a quickfix list using the result of {expr} and
                        jump to the first error.
                        If {expr} is a String, then each new-line terminated
                        line in the String is processed using the global value
                        of 'errorformat' and the result is added to the
                        quickfix list.
                        If {expr} is a List, then each String item in the list
                        is processed and added to the quickfix list.  Non
                        String items in the List are ignored.
                        See |:cc| for [!].
                        Examples: >
                                :cexpr system('grep -n xyz *')
                                :cexpr getline(1, '$')
<
```

```
                                                        *:lex* *:lexpr*
:lex[pr][!] {expr}      Same as |:cexpr|, except the location list for the
                        current window is used instead of the quickfix list.

                                                        *:cgete* *:cgetexpr*
:cgete[xpr] {expr}      Create a quickfix list using the result of {expr}.
                        Just like |:cexpr|, but don't jump to the first error.

                                                        *:lgete* *:lgetexpr*
:lgete[xpr] {expr}      Same as |:cgetexpr|, except the location list for the
                        current window is used instead of the quickfix list.

                                                        *:cadde* *:caddexpr*
:cadde[xpr] {expr}      Evaluate {expr} and add the resulting lines to the
                        current quickfix list. If a quickfix list is not
                        present, then a new list is created. The current
                        cursor position will not be changed. See |:cexpr| for
                        more information.
                        Example: >
    :g/mypattern/caddexpr expand("%") . ":" . line(".") .  ":" . getline(".")
<
                                                        *:lad* *:laddexpr*
:lad[dexpr] {expr}      Same as ":caddexpr", except the location list for the
                        current window is used instead of the quickfix list.

                                                        *:cl* *:clist*
:cl[ist] [from] [, [to]]
                        List all errors that are valid |quickfix-valid|.
                        If numbers [from] and/or [to] are given, the respective
                        range of errors is listed.  A negative number counts
                        from the last error backwards, -1 being the last error.
                        The 'switchbuf' settings are respected when jumping
                        to a buffer.

:cl[ist] +{count}       List the current and next {count} valid errors.  This
                        is similar to ":clist from from+count", where "from"
                        is the current error position.

:cl[ist]! [from] [, [to]]
                        List all errors.

:cl[ist]! +{count}      List the current and next {count} error lines.  This
                        is useful to see unrecognized lines after the current
                        one.  For example, if ":clist" shows:
        8384 testje.java:252: error: cannot find symbol ~
                        Then using ":cl! +3" shows the reason:
        8384 testje.java:252: error: cannot find symbol ~
        8385:   ZexitCode = Fmainx(); ~
        8386:              ^ ~
        8387:   symbol:   method Fmainx() ~

:lli[st] [from] [, [to]]                                *:lli* *:llist*
                        Same as ":clist", except the location list for the
                        current window is used instead of the quickfix list.

:lli[st]! [from] [, [to]]
                        List all the entries in the location list for the
                        current window.

If you insert or delete lines, mostly the correct error location is still
found because hidden marks are used.  Sometimes, when the mark has been
deleted for some reason, the message "line changed" is shown to warn you that
```

```
the error location may not be correct.  If you quit Vim and start again the
marks are lost and the error locations may not be correct anymore.

If vim is built with |+autocmd| support, two autocommands are available for
running commands before and after a quickfix command (':make', ':grep' and so
on) is executed. See |QuickFixCmdPre| and |QuickFixCmdPost| for details.

                                                *QuickFixCmdPost-example*
When 'encoding' differs from the locale, the error messages may have a
different encoding from what Vim is using.  To convert the messages you can
use this code: >
        function QfMakeConv()
           let qflist = getqflist()
           for i in qflist
              let i.text = iconv(i.text, "cp936", "utf-8")
           endfor
           call setqflist(qflist)
        endfunction

        au QuickfixCmdPost make call QfMakeConv()
Another option is using 'makeencoding'.


EXECUTE A COMMAND IN ALL THE BUFFERS IN QUICKFIX OR LOCATION LIST:
                                                        *:cdo*
:cdo[!] {cmd}           Execute {cmd} in each valid entry in the quickfix list.
                        It works like doing this: >
                                :cfirst
                                :{cmd}
                                :cnext
                                :{cmd}
                                etc.
<                       When the current file can't be |abandon|ed and the [!]
                        is not present, the command fails.
                        When an error is detected execution stops.
                        The last buffer (or where an error occurred) becomes
                        the current buffer.
                        {cmd} can contain '|' to concatenate several commands.

                        Only valid entries in the quickfix list are used.
                        A range can be used to select entries, e.g.: >
                                :10,$cdo cmd
<                       To skip entries 1 to 9.

                        Note: While this command is executing, the Syntax
                        autocommand event is disabled by adding it to
                        'eventignore'.  This considerably speeds up editing
                        each buffer.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}
                        Also see |:bufdo|, |:tabdo|, |:argdo|, |:windo|,
                        |:ldo|, |:cfdo| and |:lfdo|.

                                                        *:cfdo*
:cfdo[!] {cmd}          Execute {cmd} in each file in the quickfix list.
                        It works like doing this: >
                                :cfirst
                                :{cmd}
                                :cnfile
                                :{cmd}
                                etc.
<                       Otherwise it works the same as `:cdo`.
```

```
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

                                                        *:ldo*
:ld[o][!] {cmd}         Execute {cmd} in each valid entry in the location list
                        for the current window.
                        It works like doing this: >
                                :lfirst
                                :{cmd}
                                :lnext
                                :{cmd}
                                etc.
<                       Only valid entries in the location list are used.
                        Otherwise it works the same as `:cdo`.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

                                                        *:lfdo*
:lfdo[!] {cmd}          Execute {cmd} in each file in the location list for
                        the current window.
                        It works like doing this: >
                                :lfirst
                                :{cmd}
                                :lnfile
                                :{cmd}
                                etc.
<                       Otherwise it works the same as `:ldo`.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

==============================================================================
2. The error window                                     *quickfix-window*

                                        *:cope* *:copen* *w:quickfix_title*
:cope[n] [height]       Open a window to show the current list of errors.

                        When [height] is given, the window becomes that high
                        (if there is room).  When [height] is omitted the
                        window is made ten lines high.

                        If there already is a quickfix window, it will be made
                        the current window.  It is not possible to open a
                        second quickfix window.  If [height] is given the
                        existing window will be resized to it.

                        The window will contain a special buffer, with
                        'buftype' equal to "quickfix".  Don't change this!
                        The window will have the w:quickfix_title variable set
                        which will indicate the command that produced the
                        quickfix list. This can be used to compose a custom
                        status line if the value of 'statusline' is adjusted
                        properly. Whenever this buffer is modified by a
                        quickfix command or function, the |b:changedtick|
                        variable is incremented.

                                                        *:lop* *:lopen*
:lop[en] [height]       Open a window to show the location list for the
                        current window. Works only when the location list for
                        the current window is present.  You can have more than
                        one location window opened at a time.  Otherwise, it
                        acts the same as ":copen".
```

:ccl *:cclose*
:ccl[ose] Close the quickfix window.

:lcl *:lclose*
:lcl[ose] Close the window showing the location list for the current window.

:cw *:cwindow*
:cw[indow] [height] Open the quickfix window when there are recognized errors. If the window is already open and there are no recognized errors, close the window.

:lw *:lwindow*
:lw[indow] [height] Same as ":cwindow", except use the window showing the location list for the current window.

:cbo *:cbottom*
:cbo[ttom] Put the cursor in the last line of the quickfix window and scroll to make it visible. This is useful for when errors are added by an asynchronous callback. Only call it once in a while if there are many updates to avoid a lot of redrawing.

:lbo *:lbottom*
:lbo[ttom] Same as ":cbottom", except use the window showing the location list for the current window.

Normally the quickfix window is at the bottom of the screen. If there are vertical splits, it's at the bottom of the rightmost column of windows. To make it always occupy the full width: >
:botright cwindow
You can move the window around with |window-moving| commands.
For example, to move it to the top: CTRL-W K
The 'winfixheight' option will be set, which means that the window will mostly keep its height, ignoring 'winheight' and 'equalalways'. You can change the height manually (e.g., by dragging the status line above it with the mouse).

In the quickfix window, each line is one error. The line number is equal to the error number. The current entry is highlighted with the QuickFixLine highlighting. You can change it to your liking, e.g.: >
:hi QuickFixLine ctermbg=Yellow guibg=Yellow

You can use ":.cc" to jump to the error under the cursor.
Hitting the <Enter> key or double-clicking the mouse on a line has the same effect. The file containing the error is opened in the window above the quickfix window. If there already is a window for that file, it is used instead. If the buffer in the used window has changed, and the error is in another file, jumping to the error will fail. You will first have to make sure the window contains a buffer which can be abandoned.
CTRL-W_<Enter> *CTRL-W_<CR>*
You can use CTRL-W <Enter> to open a new window and jump to the error there.

When the quickfix window has been filled, two autocommand events are triggered. First the 'filetype' option is set to "qf", which triggers the FileType event. Then the BufReadPost event is triggered, using "quickfix" for the buffer name. This can be used to perform some action on the listed errors. Example: >
au BufReadPost quickfix setlocal modifiable
\ | silent exe 'g/^/s//\=line(".")." "/'
\ | setlocal nomodifiable
This prepends the line number to each line. Note the use of "\=" in the substitute string of the ":s" command, which is used to evaluate an

expression.
The BufWinEnter event is also triggered, again using "quickfix" for the buffer name.

Note: When adding to an existing quickfix list the autocommand are not triggered.

Note: Making changes in the quickfix window has no effect on the list of errors. 'modifiable' is off to avoid making changes. If you delete or insert lines anyway, the relation between the text and the error number is messed up. If you really want to do this, you could write the contents of the quickfix window to a file and use ":cfile" to have it parsed and used as the new error list.

location-list-window
The location list window displays the entries in a location list. When you open a location list window, it is created below the current window and displays the location list for the current window. The location list window is similar to the quickfix window, except that you can have more than one location list window open at a time. When you use a location list command in this window, the displayed location list is used.

When you select a file from the location list window, the following steps are used to find a window to edit the file:

1. If a window with the location list displayed in the location list window is present, then the file is opened in that window.
2. If the above step fails and if the file is already opened in another window, then that window is used.
3. If the above step fails then an existing window showing a buffer with 'buftype' not set is used.
4. If the above step fails, then the file is edited in a new window.

In all of the above cases, if the location list for the selected window is not yet set, then it is set to the location list displayed in the location list window.

==
3. Using more than one list of errors *quickfix-error-lists*

So far has been assumed that there is only one list of errors. Actually the ten last used lists are remembered. When starting a new list, the previous ones are automatically kept. Two commands can be used to access older error lists. They set one of the existing error lists as the current one.

:colder *:col* *E380*
:col[der] [count] Go to older error list. When [count] is given, do this [count] times. When already at the oldest error list, an error message is given.

:lolder *:lol*
:lol[der] [count] Same as `:colder`, except use the location list for the current window instead of the quickfix list.

:cnewer *:cnew* *E381*
:cnew[er] [count] Go to newer error list. When [count] is given, do this [count] times. When already at the newest error list, an error message is given.

:lnewer *:lnew*
:lnew[er] [count] Same as `:cnewer`, except use the location list for the current window instead of the quickfix list.

```
                                                *:chistory* *:chi*
:chi[story]             Show the list of error lists.  The current list is
                        marked with ">".  The output looks like:
                                  error list 1 of 3; 43 errors ~
                                > error list 2 of 3; 0 errors ~
                                  error list 3 of 3; 15 errors ~

                                                *:lhistory* *:lhi*
:lhi[story]             Show the list of location lists, otherwise like
                        `:chistory`.

When adding a new error list, it becomes the current list.

When ":colder" has been used and ":make" or ":grep" is used to add a new error
list, one newer list is overwritten.  This is especially useful if you are
browsing with ":grep" |grep|.  If you want to keep the more recent error
lists, use ":cnewer 99" first.

==============================================================================
4. Using :make                                          *:make_makeprg*

                                                        *:mak* *:make*
:mak[e][!] [arguments]  1. If vim was built with |+autocmd|, all relevant
                           |QuickFixCmdPre| autocommands are executed.
                        2. If the 'autowrite' option is on, write any changed
                           buffers
                        3. An errorfile name is made from 'makeef'.  If
                           'makeef' doesn't contain "##", and a file with this
                           name already exists, it is deleted.
                        4. The program given with the 'makeprg' option is
                           started (default "make") with the optional
                           [arguments] and the output is saved in the
                           errorfile (for Unix it is also echoed on the
                           screen).
                        5. The errorfile is read using 'errorformat'.
                        6. If vim was built with |+autocmd|, all relevant
                           |QuickFixCmdPost| autocommands are executed.
                           See example below.
                        7. If [!] is not given the first error is jumped to.
                        8. The errorfile is deleted.
                        9. You can now move through the errors with commands
                           like |:cnext| and |:cprevious|, see above.
                        This command does not accept a comment, any "
                        characters are considered part of the arguments.
                        If the encoding of the program output differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.

                                                        *:lmak* *:lmake*
:lmak[e][!] [arguments]
                        Same as ":make", except the location list for the
                        current window is used instead of the quickfix list.

The ":make" command executes the command given with the 'makeprg' option.
This is done by passing the command to the shell given with the 'shell'
option.  This works almost like typing

        ":!{makeprg} [arguments] {shellpipe} {errorfile}".

{makeprg} is the string given with the 'makeprg' option.  Any command can be
used, not just "make".  Characters '%' and '#' are expanded as usual on a
```

```
command-line.  You can use "%<" to insert the current file name without
extension, or "#<" to insert the alternate file name without extension, for
example: >
   :set makeprg=make\ #<.o

[arguments] is anything that is typed after ":make".
{shellpipe} is the 'shellpipe' option.
{errorfile} is the 'makeef' option, with ## replaced to make it unique.

The placeholder "$*" can be used for the argument list in {makeprg} if the
command needs some additional characters after its arguments.  The $* is
replaced then by all arguments.  Example: >
   :set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
or simpler >
   :let &mp = 'latex \\nonstopmode \\input\{$*}'
"$*" can be given multiple times, for example: >
   :set makeprg=gcc\ -o\ $*\ $*

The 'shellpipe' option defaults to ">" for the Amiga, MS-DOS and Win32.  This
means that the output of the compiler is saved in a file and not shown on the
screen directly.  For Unix "| tee" is used.  The compiler output is shown on
the screen and saved in a file the same time.  Depending on the shell used
"|& tee" or "2>&1| tee" is the default, so stderr output will be included.

If 'shellpipe' is empty, the {errorfile} part will be omitted.  This is useful
for compilers that write to an errorfile themselves (e.g., Manx's Amiga C).


Using QuickFixCmdPost to fix the encoding ~

It may be that 'encoding' is set to an encoding that differs from the messages
your build program produces.  This example shows how to fix this after Vim has
read the error messages: >

        function QfMakeConv()
           let qflist = getqflist()
           for i in qflist
              let i.text = iconv(i.text, "cp936", "utf-8")
           endfor
           call setqflist(qflist)
        endfunction

        au QuickfixCmdPost make call QfMakeConv()

(Example by Faque Cheng)
Another option is using 'makeencoding'.

==============================================================================
5. Using :vimgrep and :grep                                *grep* *lid*

Vim has two ways to find matches for a pattern: Internal and external.  The
advantage of the internal grep is that it works on all systems and uses the
powerful Vim search patterns.  An external grep program can be used when the
Vim grep does not do what you want.

The internal method will be slower, because files are read into memory.  The
advantages are:
- Line separators and encoding are automatically recognized, as if a file is
  being edited.
- Uses Vim search patterns.  Multi-line patterns can be used.
- When plugins are enabled: compressed and remote files can be searched.
        |gzip| |netrw|
```

To be able to do this Vim loads each file as if it is being edited. When there is no match in the file the associated buffer is wiped out again. The 'hidden' option is ignored here to avoid running out of memory or file descriptors when searching many files. However, when the |:hide| command modifier is used the buffers are kept loaded. This makes following searches in the same files a lot faster.

Note that |:copen| (or |:lopen| for |:lgrep|) may be used to open a buffer containing the search results in linked form. The |:silent| command may be used to suppress the default full screen grep output. The ":grep!" form of the |:grep| command doesn't jump to the first match automatically. These commands can be combined to create a NewGrep command: >

```
        command! -nargs=+ NewGrep execute 'silent grep! <args>' | copen 42
```

5.1 using Vim's internal grep

:vim *:vimgrep* *E682* *E683*

:vim[grep][!] /{pattern}/[g][j] {file} ...
Search for {pattern} in the files {file} ... and set the error list to the matches. Files matching 'wildignore' are ignored; files in 'suffixes' are searched last.
Without the 'g' flag each line is added only once.
With 'g' every match is added.

{pattern} is a Vim search pattern. Instead of enclosing it in / any non-ID character (see |'isident'|) can be used, so long as it does not appear in {pattern}.
'ignorecase' applies. To overrule it put |/\c| in the pattern to ignore case or |/\C| to match case.
'smartcase' is not used.
If {pattern} is empty (e.g. // is specified), the last used search pattern is used. |last-pattern|

When a number is put before the command this is used as the maximum number of matches to find. Use ":1vimgrep pattern file" to find only the first. Useful if you only want to check if there is a match and quit quickly when it's found.

Without the 'j' flag Vim jumps to the first match.
With 'j' only the quickfix list is updated.
With the [!] any changes in the current buffer are abandoned.

Every second or so the searched file name is displayed to give you an idea of the progress made.
Examples: >

```
        :vimgrep /an error/ *.c
        :vimgrep /\<FileName\>/ *.h include/*
        :vimgrep /myfunc/ **/*.c
```

< For the use of "**" see |starstar-wildcard|.

:vim[grep][!] {pattern} {file} ...
Like above, but instead of enclosing the pattern in a non-ID character use a white-separated pattern. The pattern must start with an ID character.
Example: >

```
                                :vimgrep Error *.c
<
                                                              *:lv* *:lvimgrep*
:lv[imgrep][!] /{pattern}/[g][j] {file} ...
:lv[imgrep][!] {pattern} {file} ...
                        Same as ":vimgrep", except the location list for the
                        current window is used instead of the quickfix list.

                                                    *:vimgrepa* *:vimgrepadd*
:vimgrepa[dd][!] /{pattern}/[g][j] {file} ...
:vimgrepa[dd][!] {pattern} {file} ...
                        Just like ":vimgrep", but instead of making a new list
                        of errors the matches are appended to the current
                        list.

                                                    *:lvimgrepa* *:lvimgrepadd*
:lvimgrepa[dd][!] /{pattern}/[g][j] {file} ...
:lvimgrepa[dd][!] {pattern} {file} ...
                        Same as ":vimgrepadd", except the location list for
                        the current window is used instead of the quickfix
                        list.
```

5.2 External grep

Vim can interface with "grep" and grep-like programs (such as the GNU id-utils) in a similar way to its compiler integration (see |:make| above).

[Unix trivia: The name for the Unix "grep" command comes from ":g/re/p", where "re" stands for Regular Expression.]

```
                                                              *:gr* *:grep*
:gr[ep][!] [arguments]  Just like ":make", but use 'grepprg' instead of
                        'makeprg' and 'grepformat' instead of 'errorformat'.
                        When 'grepprg' is "internal" this works like
                        |:vimgrep|.  Note that the pattern needs to be
                        enclosed in separator characters then.
                        If the encoding of the program output differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.

                                                              *:lgr* *:lgrep*
:lgr[ep][!] [arguments] Same as ":grep", except the location list for the
                        current window is used instead of the quickfix list.

                                                              *:grepa* *:grepadd*
:grepa[dd][!] [arguments]
                        Just like ":grep", but instead of making a new list of
                        errors the matches are appended to the current list.
                        Example: >
                                :call setqflist([])
                                :bufdo grepadd! something %
<                       The first command makes a new error list which is
                        empty.  The second command executes "grepadd" for each
                        listed buffer.  Note the use of ! to avoid that
                        ":grepadd" jumps to the first error, which is not
                        allowed with |:bufdo|.
                        An example that uses the argument list and avoids
                        errors for files without matches: >
                                :silent argdo try
                                  \ | grepadd! something %
                                  \ | catch /E480:/
                                  \ | endtry"
```

```
<
                        If the encoding of the program output differs from the
                        'encoding' option, you can use the 'makeencoding'
                        option to specify the encoding.

                                                        *:lgrepa* *:lgrepadd*
:lgrepa[dd][!] [arguments]
                        Same as ":grepadd", except the location list for the
                        current window is used instead of the quickfix list.
```

5.3 Setting up external grep

If you have a standard "grep" program installed, the :grep command may work well with the defaults. The syntax is very similar to the standard command: >

```
        :grep foo *.c
```

Will search all files with the .c extension for the substring "foo". The arguments to :grep are passed straight to the "grep" program, so you can use whatever options your "grep" supports.

By default, :grep invokes grep with the -n option (show file and line numbers). You can change this with the 'grepprg' option. You will need to set 'grepprg' if:

a) You are using a program that isn't called "grep"
b) You have to call grep with a full path
c) You want to pass other options automatically (e.g. case insensitive search.)

Once "grep" has executed, Vim parses the results using the 'grepformat' option. This option works in the same way as the 'errorformat' option - see that for details. You may need to change 'grepformat' from the default if your grep outputs in a non-standard format, or you are using some other program with a special format.

Once the results are parsed, Vim loads the first file containing a match and jumps to the appropriate line, in the same way that it jumps to a compiler error in |quickfix| mode. You can then use the |:cnext|, |:clist|, etc. commands to see the other matches.

5.4 Using :grep with id-utils

You can set up :grep to work with the GNU id-utils like this: >

```
        :set grepprg=lid\ -Rgrep\ -s
        :set grepformat=%f:%l:%m
```

then >

```
        :grep (regexp)
```

works just as you'd expect.
(provided you remembered to mkid first :)

5.5 Browsing source code with :vimgrep or :grep

Using the stack of error lists that Vim keeps, you can browse your files to look for functions and the functions they call. For example, suppose that you have to add an argument to the read_file() function. You enter this command: >

```
        :vimgrep /\<read_file\>/ *.c

You use ":cn" to go along the list of matches and add the argument.  At one
place you have to get the new argument from a higher level function msg(), and
need to change that one too.  Thus you use: >

        :vimgrep /\<msg\>/ *.c

While changing the msg() functions, you find another function that needs to
get the argument from a higher level.  You can again use ":vimgrep" to find
these functions.  Once you are finished with one function, you can use >

        :colder

to go back to the previous one.

This works like browsing a tree: ":vimgrep" goes one level deeper, creating a
list of branches.  ":colder" goes back to the previous level.  You can mix
this use of ":vimgrep" and "colder" to browse all the locations in a tree-like
way.  If you do this consistently, you will find all locations without the
need to write down a "todo" list.

==============================================================================
6. Selecting a compiler                                 *compiler-select*

                                                *:comp* *:compiler* *E666*
:comp[iler][!] {name}           Set options to work with compiler {name}.
                                Without the "!" options are set for the
                                current buffer.  With "!" global options are
                                set.
                                If you use ":compiler foo" in "file.foo" and
                                then ":compiler! bar" in another buffer, Vim
                                will keep on using "foo" in "file.foo".
                                {not available when compiled without the
                                |+eval| feature}


The Vim plugins in the "compiler" directory will set options to use the
selected compiler.  For `:compiler` local options are set, for `:compiler!`
global options.
                                                        *current_compiler*
To support older Vim versions, the plugins always use "current_compiler" and
not "b:current_compiler".  What the command actually does is the following:

- Delete the "current_compiler" and "b:current_compiler" variables.
- Define the "CompilerSet" user command.  With "!" it does ":set", without "!"
  it does ":setlocal".
- Execute ":runtime! compiler/{name}.vim".  The plugins are expected to set
  options with "CompilerSet" and set the "current_compiler" variable to the
  name of the compiler.
- Delete the "CompilerSet" user command.
- Set "b:current_compiler" to the value of "current_compiler".
- Without "!" the old value of "current_compiler" is restored.


For writing a compiler plugin, see |write-compiler-plugin|.


GCC                                     *quickfix-gcc*  *compiler-gcc*

There's one variable you can set for the GCC compiler:
```

```
g:compiler_gcc_ignore_unmatched_lines
                                Ignore lines that don't match any patterns
                                defined for GCC.  Useful if output from
                                commands run from make are generating false
                                positives.


MANX AZTEC C                                *quickfix-manx* *compiler-manx*

To use Vim with Manx's Aztec C compiler on the Amiga you should do the
following:
- Set the CCEDIT environment variable with the command: >
        mset "CCEDIT=vim -q"
- Compile with the -qf option.  If the compiler finds any errors, Vim is
  started and the cursor is positioned on the first error.  The error message
  will be displayed on the last line.  You can go to other errors with the
  commands mentioned above.  You can fix the errors and write the file(s).
- If you exit Vim normally the compiler will re-compile the same file.  If you
  exit with the :cq command, the compiler will terminate.  Do this if you
  cannot fix the error, or if another file needs to be compiled first.

There are some restrictions to the Quickfix mode on the Amiga.  The
compiler only writes the first 25 errors to the errorfile (Manx's
documentation does not say how to get more).  If you want to find the others,
you will have to fix a few errors and exit the editor.  After recompiling,
up to 25 remaining errors will be found.

If Vim was started from the compiler, the :sh and some :!  commands will not
work, because Vim is then running in the same process as the compiler and
stdin (standard input) will not be interactive.


PERL                                        *quickfix-perl* *compiler-perl*

The Perl compiler plugin doesn't actually compile, but invokes Perl's internal
syntax checking feature and parses the output for possible errors so you can
correct them in quick-fix mode.

Warnings are forced regardless of "no warnings" or "$^W = 0" within the file
being checked.  To disable this set g:perl_compiler_force_warnings to a zero
value.  For example: >
        let g:perl_compiler_force_warnings = 0


PYUNIT COMPILER                                          *compiler-pyunit*

This is not actually a compiler, but a unit testing framework for the
Python language.  It is included into standard Python distribution
starting from version 2.0.  For older versions, you can get it from
http://pyunit.sourceforge.net.

When you run your tests with the help of the framework, possible errors
are parsed by Vim and presented for you in quick-fix mode.

Unfortunately, there is no standard way to run the tests.
The alltests.py script seems to be used quite often, that's all.
Useful values for the 'makeprg' options therefore are:
 setlocal makeprg=./alltests.py " Run a testsuite
 setlocal makeprg=python\ %:S   " Run a single testcase

Also see http://vim.sourceforge.net/tip_view.php?tip_id=280.
```

TEX COMPILER *compiler-tex*

Included in the distribution compiler for TeX ($VIMRUNTIME/compiler/tex.vim)
uses make command if possible. If the compiler finds a file named "Makefile"
or "makefile" in the current directory, it supposes that you want to process
your *TeX files with make, and the makefile does the right work. In this case
compiler sets 'errorformat' for *TeX output and leaves 'makeprg' untouched. If
neither "Makefile" nor "makefile" is found, the compiler will not use make.
You can force the compiler to ignore makefiles by defining
b:tex_ignore_makefile or g:tex_ignore_makefile variable (they are checked for
existence only).

If the compiler chose not to use make, it need to choose a right program for
processing your input. If b:tex_flavor or g:tex_flavor (in this precedence)
variable exists, it defines TeX flavor for :make (actually, this is the name
of executed command), and if both variables do not exist, it defaults to
"latex". For example, while editing chapter2.tex \input-ed from mypaper.tex
written in AMS-TeX: >

```
        :let b:tex_flavor = 'amstex'
        :compiler tex
<       [editing...] >
        :make mypaper
```

Note that you must specify a name of the file to process as an argument (to
process the right file when editing \input-ed or \include-ed file; portable
solution for substituting % for no arguments is welcome). This is not in the
semantics of make, where you specify a target, not source, but you may specify
filename without extension ".tex" and mean this as "make filename.dvi or
filename.pdf or filename.some_result_extension according to compiler".

Note: tex command line syntax is set to usable both for MikTeX (suggestion
by Srinath Avadhanula) and teTeX (checked by Artem Chuprina). Suggestion
from |errorformat-LaTeX| is too complex to keep it working for different
shells and OSes and also does not allow to use other available TeX options,
if any. If your TeX doesn't support "-interaction=nonstopmode", please
report it with different means to express \nonstopmode from the command line.

==
7. The error format *error-file-format*

errorformat *E372* *E373* *E374*
E375 *E376* *E377* *E378*
The 'errorformat' option specifies a list of formats that are recognized. The
first format that matches with an error message is used. You can add several
formats for different messages your compiler produces, or even entries for
multiple compilers. See |efm-entries|.

Each entry in 'errorformat' is a scanf-like string that describes the format.
First, you need to know how scanf works. Look in the documentation of your
C compiler. Below you find the % items that Vim understands. Others are
invalid.

Special characters in 'errorformat' are comma and backslash. See
|efm-entries| for how to deal with them. Note that a literal "%" is matched
by "%%", thus it is not escaped with a backslash.
Keep in mind that in the `:make` and `:grep` output all NUL characters are
replaced with SOH (0x01).

Note: By default the difference between upper and lowercase is ignored. If
you want to match case, add "\C" to the pattern |/\C|.

```
Basic items

        %f              file name (finds a string)
        %l              line number (finds a number)
        %c              column number (finds a number representing character
                        column of the error, (1 <tab> == 1 character column))
        %v              virtual column number (finds a number representing
                        screen column of the error (1 <tab> == 8 screen
                        columns))
        %t              error type (finds a single character)
        %n              error number (finds a number)
        %m              error message (finds a string)
        %r              matches the "rest" of a single-line file message %O/P/Q
        %p              pointer line (finds a sequence of '-', '.', ' ' or
                        tabs and uses the length for the column number)
        %*{conv}        any scanf non-assignable conversion
        %%              the single '%' character
        %s              search text (finds a string)

The "%f" conversion may depend on the current 'isfname' setting.  "~/" is
expanded to the home directory and environment variables are expanded.

The "%f" and "%m" conversions have to detect the end of the string.  This
normally happens by matching following characters and items.  When nothing is
following the rest of the line is matched.  If "%f" is followed by a '%' or a
backslash, it will look for a sequence of 'isfname' characters.

On MS-DOS, MS-Windows and OS/2 a leading "C:" will be included in "%f", even
when using "%f:".  This means that a file name which is a single alphabetical
letter will not be detected.

The "%p" conversion is normally followed by a "^".  It's used for compilers
that output a line like: >
            ^
or >
   ---------^
to indicate the column of the error.  This is to be used in a multi-line error
message.  See |errorformat-javac| for a  useful example.

The "%s" conversion specifies the text to search for to locate the error line.
The text is used as a literal string.  The anchors "^" and "$" are added to
the text to locate the error line exactly matching the search text and the
text is prefixed with the "\V" atom to make it "very nomagic".  The "%s"
conversion can be used to locate lines without a line number in the error
output.  Like the output of the "grep" shell command.
When the pattern is present the line number will not be used.

Changing directory

The following uppercase conversion characters specify the type of special
format strings.  At most one of them may be given as a prefix at the begin
of a single comma-separated format pattern.
Some compilers produce messages that consist of directory names that have to
be prepended to each file name read by %f (example: GNU make).  The following
codes can be used to scan these directory names; they will be stored in an
internal directory stack.                                       *E379*
        %D              "enter directory" format string; expects a following
                          %f that finds the directory name
        %X              "leave directory" format string; expects following %f
```

```
When defining an "enter directory" or "leave directory" format, the "%D" or
"%X" has to be given at the start of that substring.  Vim tracks the directory
changes and prepends the current directory to each erroneous file found with a
relative path.  See |quickfix-directory-stack| for details, tips and
limitations.


Multi-line messages                                *errorformat-multi-line*

It is possible to read the output of programs that produce multi-line
messages, i.e. error strings that consume more than one line.  Possible
prefixes are:
        %E              start of a multi-line error message
        %W              start of a multi-line warning message
        %I              start of a multi-line informational message
        %A              start of a multi-line message (unspecified type)
        %>              for next line start with current pattern again |efm-%>|
        %C              continuation of a multi-line message
        %Z              end of a multi-line message
These can be used with '+' and '-', see |efm-ignore| below.

Using "\n" in the pattern won't work to match multi-line messages.

Example: Your compiler happens to write out errors in the following format
(leading line numbers not being part of the actual output):

     1  Error 275 ~
     2  line 42 ~
     3  column 3 ~
     4  ' ' expected after '--' ~

The appropriate error format string has to look like this: >
   :set efm=%EError\ %n,%Cline\ %l,%Ccolumn\ %c,%Z%m

And the |:clist| error message generated for this error is:

 1:42 col 3 error 275:  ' ' expected after '--'

Another example: Think of a Python interpreter that produces the following
error message (line numbers are not part of the actual output):

     1  ======================================================================
     2  FAIL: testGetTypeIdCachesResult (dbfacadeTest.DjsDBFacadeTest)
     3  ----------------------------------------------------------------------
     4  Traceback (most recent call last):
     5    File "unittests/dbfacadeTest.py", line 89, in testFoo
     6      self.assertEquals(34, dtid)
     7    File "/usr/lib/python2.2/unittest.py", line 286, in
     8   failUnlessEqual
     9      raise self.failureException, \
    10  AssertionError: 34 != 33
    11
    12  ----------------------------------------------------------------------
    13  Ran 27 tests in 0.063s

Say you want |:clist| write the relevant information of this message only,
namely:
 5 unittests/dbfacadeTest.py:89:  AssertionError: 34 != 33

Then the error format string could be defined as follows: >
  :set efm=%C\ %.%#,%A\ \ File\ \"%f\"\\,\ line\ %l%.%#,%Z%[%^\ ]%\\@=%m
```

```
Note that the %C string is given before the %A here: since the expression
' %.%#' (which stands for the regular expression ' .*') matches every line
starting with a space, followed by any characters to the end of the line,
it also hides line 7 which would trigger a separate error message otherwise.
Error format strings are always parsed pattern by pattern until the first
match occurs.
                                                          *efm-%>*
The %> item can be used to avoid trying patterns that appear earlier in
'errorformat'.  This is useful for patterns that match just about anything.
For example, if the error looks like this:

        Error in line 123 of foo.c: ~
        unknown variable "i" ~

This can be found with: >
        :set efm=xxx,%E%>Error in line %l of %f:,%Z%m
Where "xxx" has a pattern that would also match the second line.

Important: There is no memory of what part of the errorformat matched before;
every line in the error file gets a complete new run through the error format
lines.  For example, if one has: >
  setlocal efm=aa,bb,cc,dd,ee
Where aa, bb, etc. are error format strings.  Each line of the error file will
be matched to the pattern aa, then bb, then cc, etc.  Just because cc matched
the previous error line does _not_ mean that dd will be tried first on the
current line, even if cc and dd are multi-line errorformat strings.



Separate file name                      *errorformat-separate-filename*

These prefixes are useful if the file name is given once and multiple messages
follow that refer to this file name.
        %O              single-line file message: overread the matched part
        %P              single-line file message: push file %f onto the stack
        %Q              single-line file message: pop the last file from stack

Example: Given a compiler that produces the following error logfile (without
leading line numbers):

     1  [a1.tt]
     2  (1,17)  error: ';' missing
     3  (21,2)  warning: variable 'z' not defined
     4  (67,3)  error: end of file found before string ended
     5
     6  [a2.tt]
     7
     8  [a3.tt]
     9  NEW compiler v1.1
    10  (2,2)   warning: variable 'x' not defined
    11  (67,3)  warning: 's' already defined

This logfile lists several messages for each file enclosed in [...] which are
properly parsed by an error format like this: >
  :set efm=%+P[%f],(%l\\,%c)%*[\ ]%t%*[^:]:\ %m,%-Q

A call of |:clist| writes them accordingly with their correct filenames:

  2 a1.tt:1 col 17 error: ';' missing
  3 a1.tt:21 col 2 warning: variable 'z' not defined
  4 a1.tt:67 col 3 error: end of file found before string ended
  8 a3.tt:2 col 2 warning: variable 'x' not defined
```

```
  9 a3.tt:67 col 3 warning: 's' already defined
```

Unlike the other prefixes that all match against whole lines, %P, %Q and %O
can be used to match several patterns in the same line. Thus it is possible
to parse even nested files like in the following line:
  {"file1" {"file2" error1} error2 {"file3" error3 {"file4" error4 error5}}}
The %O then parses over strings that do not contain any push/pop file name
information. See |errorformat-LaTeX| for an extended example.

Ignoring and using whole messages *efm-ignore*

The codes '+' or '-' can be combined with the uppercase codes above; in that
case they have to precede the letter, e.g. '%+A' or '%-G':
	%-		do not include the matching multi-line in any output
	%+		include the whole matching line in the %m error string

One prefix is only useful in combination with '+' or '-', namely %G. It parses
over lines containing general information like compiler version strings or
other headers that can be skipped.
	%-G		ignore this message
	%+G		general message

Pattern matching

The scanf()-like "%*[]" notation is supported for backward-compatibility
with previous versions of Vim. However, it is also possible to specify
(nearly) any Vim supported regular expression in format strings.
Since meta characters of the regular expression language can be part of
ordinary matching strings or file names (and therefore internally have to
be escaped), meta symbols have to be written with leading '%':
	%\		The single '\' character. Note that this has to be
			escaped ("%\\") in ":set errorformat=" definitions.
	%.		The single '.' character.
	%#		The single '*'(!) character.
	%^		The single '^' character. Note that this is not
			useful, the pattern already matches start of line.
	%$		The single '$' character. Note that this is not
			useful, the pattern already matches end of line.
	%[		The single '[' character for a [] character range.
	%~		The single '~' character.
When using character classes in expressions (see |/\i| for an overview),
terms containing the "\+" quantifier can be written in the scanf() "%*"
notation. Example: "%\\d%\\+" ("\d\+", "any number") is equivalent to "%*\\d".
Important note: The \(...\) grouping of sub-matches can not be used in format
specifications because it is reserved for internal conversions.

Multiple entries in 'errorformat' *efm-entries*

To be able to detect output from several compilers, several format patterns
may be put in 'errorformat', separated by commas (note: blanks after the comma
are ignored). The first pattern that has a complete match is used. If no
match is found, matching parts from the last one will be used, although the
file name is removed and the error message is set to the whole message. If
there is a pattern that may match output from several compilers (but not in a
right way), put it after one that is more restrictive.

To include a comma in a pattern precede it with a backslash (you have to type
two in a ":set" command). To include a backslash itself give two backslashes
(you have to type four in a ":set" command). You also need to put a backslash

before a space for ":set".

Valid matches *quickfix-valid*

If a line does not completely match one of the entries in 'errorformat', the whole line is put in the error message and the entry is marked "not valid" These lines are skipped with the ":cn" and ":cp" commands (unless there is no valid line at all). You can use ":cl!" to display all the error messages.

If the error format does not contain a file name Vim cannot switch to the correct file. You will have to do this by hand.

Examples

The format of the file from the Amiga Aztec compiler is:

```
filename>linenumber:columnnumber:errortype:errornumber:errormessage

filename        name of the file in which the error was detected
linenumber      line number where the error was detected
columnnumber    column number where the error was detected
errortype       type of the error, normally a single 'E' or 'W'
errornumber     number of the error (for lookup in the manual)
errormessage    description of the error
```

This can be matched with this 'errorformat' entry:

```
%f>%l:%c:%t:%n:%m
```

Some examples for C compilers that produce single-line error outputs:

```
%f:%l:\ %t%*[^0123456789]%n:\ %m        for Manx/Aztec C error messages
                                        (scanf() doesn't understand [0-9])
%f\ %l\ %t%*[^0-9]%n:\ %m               for SAS C
\"%f\"\\,%*[^0-9]%l:\ %m                for generic C compilers
%f:%l:\ %m                              for GCC
%f:%l:\ %m,%Dgmake[%*\\d]:\ Entering\ directory\ `%f',
%Dgmake[%*\\d]:\ Leaving\ directory\ `%f'
                                        for GCC with gmake (concat the lines!)
%f(%l)\ :\ %*[^:]:\ %m                  old SCO C compiler (pre-OS5)
%f(%l)\ :\ %t%*[^0-9]%n:\ %m            idem, with error type and number
%f:%l:\ %m,In\ file\ included\ from\ %f:%l:,\^I\^Ifrom\ %f:%l%m
                                        for GCC, with some extras
```

Extended examples for the handling of multi-line messages are given below, see |errorformat-Jikes| and |errorformat-LaTeX|.

Note the backslash in front of a space and double quote. It is required for the :set command. There are two backslashes in front of a comma, one for the :set command and one to avoid recognizing the comma as a separator of error formats.

Filtering messages

If you have a compiler that produces error messages that do not fit in the format string, you could write a program that translates the error messages into this format. You can use this program with the ":make" command by changing the 'makeprg' option. For example: >

```
   :set mp=make\ \\\|&\ error_filter
```

The backslashes before the pipe character are required to avoid it to be recognized as a command separator. The backslash before each space is

required for the set command.

==
8. The directory stack *quickfix-directory-stack*

Quickfix maintains a stack for saving all used directories parsed from the make output. For GNU-make this is rather simple, as it always prints the absolute path of all directories it enters and leaves. Regardless if this is done via a 'cd' command in the makefile or with the parameter "-C dir" (change to directory before reading the makefile). It may be useful to use the switch "-w" to force GNU-make to print out the working directory before and after processing.

Maintaining the correct directory is more complicated if you don't use GNU-make. AIX-make for example doesn't print any information about its working directory. Then you need to enhance the makefile. In the makefile of LessTif there is a command which echoes "Making {target} in {dir}". The special problem here is that it doesn't print information on leaving the directory and that it doesn't print the absolute path.

To solve the problem with relative paths and missing "leave directory" messages Vim uses following algorithm:

1) Check if the given directory is a subdirectory of the current directory.
   If this is true, store it as the current directory.
2) If it is not a subdir of the current directory, try if this is a
   subdirectory of one of the upper directories.
3) If the directory still isn't found, it is assumed to be a subdirectory
   of Vim's current directory.

Additionally it is checked for every file, if it really exists in the identified directory. If not, it is searched in all other directories of the directory stack (NOT the directory subtree!). If it is still not found, it is assumed that it is in Vim's current directory.

There are limitations in this algorithm. These examples assume that make just prints information about entering a directory in the form "Making all in dir".

1) Assume you have following directories and files:
   ./dir1
   ./dir1/file1.c
   ./file1.c

   If make processes the directory "./dir1" before the current directory and there is an error in the file "./file1.c", you will end up with the file "./dir1/file.c" loaded by Vim.

   This can only be solved with a "leave directory" message.

2) Assume you have following directories and files:
   ./dir1
   ./dir1/dir2
   ./dir2

   You get the following:

Make output	Directory interpreted by Vim
Making all in dir1	./dir1
Making all in dir2	./dir1/dir2
Making all in dir2	./dir1/dir2

```
   This can be solved by printing absolute directories in the "enter directory"
   message or by printing "leave directory" messages.

To avoid this problem, ensure to print absolute directory names and "leave
directory" messages.

Examples for Makefiles:

Unix:
    libs:
            for dn in $(LIBDIRS); do                            \
                (cd $$dn; echo "Entering dir '$$(pwd)'"; make); \
                echo "Leaving dir";                             \
            done

Add
    %DEntering\ dir\ '%f',%XLeaving\ dir
to your 'errorformat' to handle the above output.

Note that Vim doesn't check if the directory name in a "leave directory"
messages is the current directory.  This is why you could just use the message
"Leaving dir".

==============================================================================
9. Specific error file formats                  *errorformats*

                                                *errorformat-Jikes*
Jikes(TM), a source-to-bytecode Java compiler published by IBM Research,
produces simple multi-line error messages.

An 'errorformat' string matching the produced messages is shown below.
The following lines can be placed in the user's |vimrc| to overwrite Vim's
recognized default formats, or see |:set+=| how to install this format
additionally to the default. >

  :set efm=%A%f:%l:%c:%*\\d:%*\\d:,
        \%C%*\\s%trror:%m,
        \%+C%*[^:]%trror:%m,
        \%C%*\\s%tarning:%m,
        \%C%m
<
Jikes(TM) produces a single-line error message when invoked with the option
"+E", and can be matched with the following: >

  :setl efm=%f:%l:%v:%*\\d:%*\\d:%*\\s%m
<
                                                *errorformat-javac*
This 'errorformat' has been reported to work well for javac, which outputs a
line with "^" to indicate the column of the error: >
  :setl efm=%A%f:%l:\ %m,%-Z%p^,%-C%.%#
or: >
  :setl efm=%A%f:%l:\ %m,%+Z%p^,%+C%.%#,%-G%.%#
<
Here is an alternative from Michael F. Lamb for Unix that filters the errors
first: >
  :setl errorformat=%Z%f:%l:\ %m,%A%p^,%-G%*[^sl]%.%#
  :setl makeprg=javac\ %:S\ 2>&1\ \\\|\ vim-javac-filter

You need to put the following in "vim-javac-filter" somewhere in your path
(e.g., in ~/bin) and make it executable: >
   #!/bin/sed -f
   /\^$/s/\t/\ /g;/:[0-9]\+:/{h;d};/^[ \t]*\^/G;
```

```
In English, that sed script:
- Changes single tabs to single spaces and
- Moves the line with the filename, line number, error message to just after
  the pointer line. That way, the unused error text between doesn't break
  vim's notion of a "multi-line message" and also doesn't force us to include
  it as a "continuation of a multi-line message."

                                                *errorformat-ant*
For ant (http://jakarta.apache.org/) the above errorformat has to be modified
to honour the leading [javac] in front of each javac output line: >
  :set efm=%A\ %#[javac]\ %f:%l:\ %m,%-Z\ %#[javac]\ %p^,%-C%.%#

The 'errorformat' can also be configured to handle ant together with either
javac or jikes.  If you're using jikes, you should tell ant to use jikes' +E
command line switch which forces jikes to generate one-line error messages.
This is what the second line (of a build.xml file) below does: >
  <property name = "build.compiler"       value = "jikes"/>
  <property name = "build.compiler.emacs" value = "true"/>

The 'errorformat' which handles ant with both javac and jikes is: >
  :set efm=\ %#[javac]\ %#%f:%l:%c:%*\\d:%*\\d:\ %t%[%^:]%#:%m,
           \%A\ %#[javac]\ %f:%l:\ %m,%-Z\ %#[javac]\ %p^,%-C%.%#
<
                                                *errorformat-jade*
parsing jade (see http://www.jclark.com/) errors is simple: >
  :set efm=jade:%f:%l:%c:%t:%m
<
                                                *errorformat-LaTeX*
The following is an example how an 'errorformat' string can be specified
for the (La)TeX typesetting system which displays error messages over
multiple lines.  The output of ":clist" and ":cc" etc. commands displays
multi-lines in a single line, leading white space is removed.
It should be easy to adopt the above LaTeX errorformat to any compiler output
consisting of multi-line errors.

The commands can be placed in a |vimrc| file or some other Vim script file,
e.g. a script containing LaTeX related stuff which is loaded only when editing
LaTeX sources.
Make sure to copy all lines of the example (in the given order), afterwards
remove the comment lines.  For the '\' notation at the start of some lines see
|line-continuation|.

               First prepare 'makeprg' such that LaTeX will report multiple
               errors; do not stop when the first error has occurred: >
 :set makeprg=latex\ \\\\nonstopmode\ \\\\input\\{$*}
<
               Start of multi-line error messages: >
 :set efm=%E!\ LaTeX\ %trror:\ %m,
        \%E!\ %m,
<              Start of multi-line warning messages; the first two also
               include the line number.  Meaning of some regular expressions:
                 - "%.%#"  (".*")   matches a (possibly empty) string
                 - "%*\\d" ("\d\+") matches a number >
        \%+WLaTeX\ %.%#Warning:\ %.%#line\ %l%.%#,
        \%+W%.%#\ at\ lines\ %l--%*\\d,
        \%WLaTeX\ %.%#Warning:\ %m,
<              Possible continuations of error/warning messages; the first
               one also includes the line number: >
        \%Cl.%l\ %m,
        \%+C\ \ %m.,
        \%+C%.%#-%.%#,
```

```
        \%+C%.%#[]%.%#,
        \%+C[]%.%#,
        \%+C%.%#%[{}\\]%.%#,
        \%+C<%.%#>%.%#,
        \%C\ \ %m,
<               Lines that match the following patterns do not contain any
                important information; do not include them in messages: >
        \%-GSee\ the\ LaTeX%m,
        \%-GType\ \ H\ <return>%m,
        \%-G\ ...%.%#,
        \%-G%.%#\ (C)\ %.%#,
        \%-G(see\ the\ transcript%.%#),
<               Generally exclude any empty or whitespace-only line from
                being displayed: >
        \%-G\\s%#,
<               The LaTeX output log does not specify the names of erroneous
                source files per line; rather they are given globally,
                enclosed in parentheses.
                The following patterns try to match these names and store
                them in an internal stack.  The patterns possibly scan over
                the same input line (one after another), the trailing "%r"
                conversion indicates the "rest" of the line that will be
                parsed in the next go until the end of line is reached.

                Overread a file name enclosed in '('...')'; do not push it
                on a stack since the file apparently does not contain any
                error: >
        \%+O(%f)%r,
<               Push a file name onto the stack.  The name is given after '(': >
        \%+P(%f%r,
        \%+P\ %\\=(%f%r,
        \%+P%*[^()](%f%r,
        \%+P[%\\d%[^()]%#(%f%r,
<               Pop the last stored file name when a ')' is scanned: >
        \%+Q)%r,
        \%+Q%*[^()])%r,
        \%+Q[%\\d%*[^()])%r
```

Note that in some cases file names in the LaTeX output log cannot be parsed
properly. The parser might have been messed up by unbalanced parentheses
then. The above example tries to catch the most relevant cases only.
You can customize the given setting to suit your own purposes, for example,
all the annoying "Overfull ..." warnings could be excluded from being
recognized as an error.
Alternatively to filtering the LaTeX compiler output, it is also possible
to directly read the *.log file that is produced by the [La]TeX compiler.
This contains even more useful information about possible error causes.
However, to properly parse such a complex file, an external filter should
be used. See the description further above how to make such a filter known
by Vim.

errorformat-Perl

In $VIMRUNTIME/tools you can find the efm_perl.pl script, which filters Perl
error messages into a format that quickfix mode will understand. See the
start of the file about how to use it. (This script is deprecated, see
|compiler-perl|.)

vim:tw=78:ts=8:ft=help:norl:

windows.txt   For Vim version 8.0. Last change: 2017 Sep 25

VIM REFERENCE MANUAL by Bram Moolenaar

Editing with multiple windows and buffers. *windows* *buffers*

The commands which have been added to use multiple windows and buffers are explained here. Additionally, there are explanations for commands that work differently when used in combination with more than one window.

The basics are explained in chapter 7 and 8 of the user manual |usr_07.txt| |usr_08.txt|.

{Vi does not have any of these commands}
{not able to use multiple windows when the |+windows| feature was disabled at compile time}
{not able to use vertically split windows when the |+vertsplit| feature was disabled at compile time}

==========

1. Introduction *windows-intro* *window*

Summary:
A buffer is the in-memory text of a file.
A window is a viewport on a buffer.
A tab page is a collection of windows.

A window is a viewport onto a buffer. You can use multiple windows on one buffer, or several windows on different buffers.

A buffer is a file loaded into memory for editing. The original file remains unchanged until you write the buffer to the file.

A buffer can be in one of three states:

active-buffer
active: The buffer is displayed in a window. If there is a file for this buffer, it has been read into the buffer. The buffer may have been modified since then and thus be different from the file.

hidden-buffer
hidden: The buffer is not displayed. If there is a file for this buffer, it has been read into the buffer. Otherwise it's the same as an active buffer, you just can't see it.

inactive-buffer
inactive: The buffer is not displayed and does not contain anything. Options for the buffer are remembered if the file was once loaded. It can contain marks from the |viminfo| file. But the buffer doesn't contain text.

In a table:

state	displayed in window	loaded	":buffers" shows
active	yes	yes	'a'
hidden	no	yes	'h'
inactive	no	no	' '

Note: All CTRL-W commands can also be executed with |:wincmd|, for those
places where a Normal mode command can't be used or is inconvenient.

The main Vim window can hold several split windows. There are also tab pages
|tab-page|, each of which can hold multiple windows.
window-ID *winid* *windowid*
Each window has a unique identifier called the window ID. This identifier
will not change within a Vim session. The |win_getid()| and |win_id2tabwin()|
functions can be used to convert between the window/tab number and the
identifier. There is also the window number, which may change whenever
windows are opened or closed, see |winnr()|.

Each buffer has a unique number and the number will not change within a Vim
session. The |bufnr()| and |bufname()| functions can be used to convert
between a buffer name and the buffer number.

==
2. Starting Vim *windows-starting*

By default, Vim starts with one window, just like Vi.

The "-o" and "-O" arguments to Vim can be used to open a window for each file
in the argument list. The "-o" argument will split the windows horizontally;
the "-O" argument will split the windows vertically. If both "-o" and "-O"
are given, the last one encountered will be used to determine the split
orientation. For example, this will open three windows, split horizontally: >
```
vim -o file1 file2 file3
```

"-oN", where N is a decimal number, opens N windows split horizontally. If
there are more file names than windows, only N windows are opened and some
files do not get a window. If there are more windows than file names, the
last few windows will be editing empty buffers. Similarly, "-ON" opens N
windows split vertically, with the same restrictions.

If there are many file names, the windows will become very small. You might
want to set the 'winheight' and/or 'winwidth' options to create a workable
situation.

Buf/Win Enter/Leave |autocommand|s are not executed when opening the new
windows and reading the files, that's only done when they are really entered.

status-line
A status line will be used to separate windows. The 'laststatus' option tells
when the last window also has a status line:
```
'laststatus' = 0        never a status line
'laststatus' = 1        status line if there is more than one window
'laststatus' = 2        always a status line
```

You can change the contents of the status line with the 'statusline' option.
This option can be local to the window, so that you can have a different
status line in each window.

Normally, inversion is used to display the status line. This can be changed
with the 's' character in the 'highlight' option. For example, "sb" sets it to

bold characters. If no highlighting is used for the status line ("sn"), the '^' character is used for the current window, and '=' for other windows. If the mouse is supported and enabled with the 'mouse' option, a status line can be dragged to resize windows.

Note: If you expect your status line to be in reverse video and it isn't, check if the 'highlight' option contains "si". In version 3.0, this meant to invert the status line. Now it should be "sr", reverse the status line, as "si" now stands for italic! If italic is not available on your terminal, the status line is inverted anyway; you will only see this problem on terminals that have termcap codes for italics.

==
3. Opening and closing a window   *opening-window* *E36*

CTRL-W s   *CTRL-W_s*
CTRL-W S   *CTRL-W_S*
CTRL-W CTRL-S   *CTRL-W_CTRL-S*
:[N]sp[lit] [++opt] [+cmd] [file]   *:sp* *:split*
                Split current window in two. The result is two viewports on
                the same file.

                Make the new window N high (default is to use half the height
                of the current window). Reduces the current window height to
                create room (and others, if the 'equalalways' option is set,
                'eadirection' isn't "hor", and one of them is higher than the
                current or the new window).

                If [file] is given it will be edited in the new window. If it
                is not loaded in any buffer, it will be read. Else the new
                window will use the already loaded buffer.

                Note: CTRL-S does not work on all terminals and might block
                further input, use CTRL-Q to get going again.
                Also see |++opt| and |+cmd|.

CTRL-W CTRL-V   *CTRL-W_CTRL-V*
CTRL-W v   *CTRL-W_v*
:[N]vs[plit] [++opt] [+cmd] [file]   *:vs* *:vsplit*
                Like |:split|, but split vertically. The windows will be
                spread out horizontally if
                1. a width was not specified,
                2. 'equalalways' is set,
                3. 'eadirection' isn't "ver", and
                4. one of the other windows is wider than the current or new
                   window.
                Note: In other places CTRL-Q does the same as CTRL-V, but here
                it doesn't!

CTRL-W n   *CTRL-W_n*
CTRL-W CTRL_N   *CTRL-W_CTRL-N*
:[N]new [++opt] [+cmd]   *:new*
                Create a new window and start editing an empty file in it.
                Make new window N high (default is to use half the existing
                height). Reduces the current window height to create room (and
                others, if the 'equalalways' option is set and 'eadirection'
                isn't "hor").
                Also see |++opt| and |+cmd|.
                If 'fileformats' is not empty, the first format given will be
                used for the new buffer. If 'fileformats' is empty, the
                'fileformat' of the current buffer is used. This can be
                overridden with the |++opt| argument.

```
                Autocommands are executed in this order:
                1. WinLeave for the current window
                2. WinEnter for the new window
                3. BufLeave for the current buffer
                4. BufEnter for the new buffer
                This behaves like a ":split" first, and then an ":enew"
                command.

:[N]vne[w] [++opt] [+cmd] [file]                        *:vne* *:vnew*
                Like |:new|, but split vertically.  If 'equalalways' is set
                and 'eadirection' isn't "ver" the windows will be spread out
                horizontally, unless a width was specified.

:[N]new [++opt] [+cmd] {file}
:[N]sp[lit] [++opt] [+cmd] {file}                       *:split_f*
                Create a new window and start editing file {file} in it.  This
                behaves like a ":split" first, and then an ":e" command.
                If [+cmd] is given, execute the command when the file has been
                loaded |+cmd|.
                Also see |++opt|.
                Make new window N high (default is to use half the existing
                height).  Reduces the current window height to create room
                (and others, if the 'equalalways' option is set).

:[N]sv[iew] [++opt] [+cmd] {file}               *:sv* *:sview* *splitview*
                Same as ":split", but set 'readonly' option for this buffer.

:[N]sf[ind] [++opt] [+cmd] {file}               *:sf* *:sfind* *splitfind*
                Same as ":split", but search for {file} in 'path' like in
                |:find|.  Doesn't split if {file} is not found.

CTRL-W CTRL-^                                   *CTRL-W_CTRL-^* *CTRL-W_^*
CTRL-W ^        Does ":split #", split window in two and edit alternate file.
                When a count is given, it becomes ":split #N", split window
                and edit buffer N.

                                                *CTRL-W_:*
CTRL-W :        Does the same as typing |:| : edit a command line.  Useful in a
                terminal window, where all Vim commands must be preceded with
                CTRL-W or 'termkey'.

Note that the 'splitbelow' and 'splitright' options influence where a new
window will appear.

                                                *:vert* *:vertical*
:vert[ical] {cmd}
                Execute {cmd}.  If it contains a command that splits a window,
                it will be split vertically.
                Doesn't work for |:execute| and |:normal|.

:lefta[bove] {cmd}                              *:lefta* *:leftabove*
:abo[veleft] {cmd}                              *:abo* *:aboveleft*
                Execute {cmd}.  If it contains a command that splits a window,
                it will be opened left (vertical split) or above (horizontal
                split) the current window.  Overrules 'splitbelow' and
                'splitright'.
                Doesn't work for |:execute| and |:normal|.

:rightb[elow] {cmd}                             *:rightb* *:rightbelow*
:bel[owright] {cmd}                             *:bel* *:belowright*
                Execute {cmd}.  If it contains a command that splits a window,
                it will be opened right (vertical split) or below (horizontal
```

split) the current window. Overrules 'splitbelow' and
'splitright'.
Doesn't work for |:execute| and |:normal|.

:topleft *E442*
:to[pleft] {cmd}
Execute {cmd}. If it contains a command that splits a window,
it will appear at the top and occupy the full width of the Vim
window. When the split is vertical the window appears at the
far left and occupies the full height of the Vim window.
Doesn't work for |:execute| and |:normal|.

:bo *:botright*
:bo[tright] {cmd}
Execute {cmd}. If it contains a command that splits a window,
it will appear at the bottom and occupy the full width of the
Vim window. When the split is vertical the window appears at
the far right and occupies the full height of the Vim window.
Doesn't work for |:execute| and |:normal|.

These command modifiers can be combined to make a vertically split window
occupy the full height. Example: >
:vertical topleft split tags
Opens a vertically split, full-height window on the "tags" file at the far
left of the Vim window.

Closing a window

:q[uit]
:{count}q[uit]
CTRL-W q *CTRL-W_q*
CTRL-W CTRL-Q *CTRL-W_CTRL-Q*
Without {count}: Quit the current window. If {count} is
given quit the {count} window.

When quitting the last window (not counting a help window),
exit Vim.

When 'hidden' is set, and there is only one window for the
current buffer, it becomes hidden. When 'hidden' is not set,
and there is only one window for the current buffer, and the
buffer was changed, the command fails.

(Note: CTRL-Q does not work on all terminals).

If [count] is greater than the last window number the last
window will be closed: >
:1quit " quit the first window
:$quit " quit the last window
:9quit " quit the last window
" if there are fewer than 9 windows opened
:-quit " quit the previous window
:+quit " quit the next window
:+2quit " quit the second next window
<
:q[uit]!
:{count}q[uit]!
Without {count}: Quit the current window. If {count} is
given quit the {count} window.

If this was the last window for a buffer, any changes to that
buffer are lost. When quitting the last window (not counting
help windows), exit Vim. The contents of the buffer are lost,
even when 'hidden' is set.

:clo[se][!]
:{count}clo[se][!]
CTRL-W c *CTRL-W_c* *:clo* *:close*
Without {count}: Close the current window. If {count} is
given close the {count} window.

When the 'hidden' option is set, or when the buffer was
changed and the [!] is used, the buffer becomes hidden (unless
there is another window editing it).

When there is only one window in the current tab page and
there is another tab page, this closes the current tab page.
|tab-page|.

This command fails when: *E444*
- There is only one window on the screen.
- When 'hidden' is not set, [!] is not used, the buffer has
changes, and there is no other window on this buffer.
Changes to the buffer are not written and won't get lost, so
this is a "safe" command.

CTRL-W CTRL-C *CTRL-W_CTRL-C*
You might have expected that CTRL-W CTRL-C closes the current
window, but that does not work, because the CTRL-C cancels the
command.

:hide
:hid[e]
:{count}hid[e]
Without {count}: Quit the current window, unless it is the
last window on the screen.
If {count} is given quit the {count} window.

The buffer becomes hidden (unless there is another window
editing it or 'bufhidden' is "unload", "delete" or "wipe").
If the window is the last one in the current tab page the tab
page is closed. |tab-page|

The value of 'hidden' is irrelevant for this command. Changes
to the buffer are not written and won't get lost, so this is a
"safe" command.

:hid[e] {cmd} Execute {cmd} with 'hidden' is set. The previous value of
'hidden' is restored after {cmd} has been executed.
Example: >
:hide edit Makefile
< This will edit "Makefile", and hide the current buffer if it
has any changes.

:on[ly][!]
:{count}on[ly][!]
CTRL-W o *CTRL-W_o* *E445*
CTRL-W CTRL-O *CTRL-W_CTRL-O* *:on* *:only*
Make the current window the only one on the screen. All other
windows are closed. For {count} see |:quit| command.

When the 'hidden' option is set, all buffers in closed windows

become hidden.

When 'hidden' is not set, and the 'autowrite' option is set, modified buffers are written. Otherwise, windows that have buffers that are modified are not removed, unless the [!] is given, then they become hidden. But modified buffers are never abandoned, so changes cannot get lost.

==
4. Moving cursor to other windows *window-move-cursor*

CTRL-W <Down> *CTRL-W_<Down>*
CTRL-W CTRL-J *CTRL-W_CTRL-J* *CTRL-W_j*
CTRL-W j Move cursor to Nth window below current one. Uses the cursor position to select between alternatives.

CTRL-W <Up> *CTRL-W_<Up>*
CTRL-W CTRL-K *CTRL-W_CTRL-K* *CTRL-W_k*
CTRL-W k Move cursor to Nth window above current one. Uses the cursor position to select between alternatives.

CTRL-W <Left> *CTRL-W_<Left>*
CTRL-W CTRL-H *CTRL-W_CTRL-H*
CTRL-W <BS> *CTRL-W_<BS>* *CTRL-W_h*
CTRL-W h Move cursor to Nth window left of current one. Uses the cursor position to select between alternatives.

CTRL-W <Right> *CTRL-W_<Right>*
CTRL-W CTRL-L *CTRL-W_CTRL-L* *CTRL-W_l*
CTRL-W l Move cursor to Nth window right of current one. Uses the cursor position to select between alternatives.

CTRL-W w *CTRL-W_w* *CTRL-W_CTRL-W*
CTRL-W CTRL-W Without count: move cursor to window below/right of the current one. If there is no window below or right, go to top-left window.
With count: go to Nth window (windows are numbered from top-left to bottom-right). To obtain the window number see |bufwinnr()| and |winnr()|. When N is larger than the number of windows go to the last window.

CTRL-W_W
CTRL-W W Without count: move cursor to window above/left of current one. If there is no window above or left, go to bottom-right window. With count: go to Nth window, like with CTRL-W w.

CTRL-W t *CTRL-W_t* *CTRL-W_CTRL-T*
CTRL-W CTRL-T Move cursor to top-left window.

CTRL-W b *CTRL-W_b* *CTRL-W_CTRL-B*
CTRL-W CTRL-B Move cursor to bottom-right window.

CTRL-W p *CTRL-W_p* *CTRL-W_CTRL-P*
CTRL-W CTRL-P Go to previous (last accessed) window.

CTRL-W_P *E441*
CTRL-W P Go to preview window. When there is no preview window this is an error.
{not available when compiled without the |+quickfix| feature}

If Visual mode is active and the new window is not for the same buffer, the Visual mode is ended. If the window is on the same buffer, the cursor

position is set to keep the same Visual area selected.

:winc *:wincmd*
These commands can also be executed with ":wincmd":

:[count]winc[md] {arg}
Like executing CTRL-W [count] {arg}. Example: >
:wincmd j
< Moves to the window below the current one.
This command is useful when a Normal mode cannot be used (for the |CursorHold| autocommand event). Or when a Normal mode command is inconvenient.
The count can also be a window number. Example: >
:exe nr . "wincmd w"
< This goes to window "nr".

==
5. Moving windows around *window-moving*

CTRL-W r *CTRL-W_r* *CTRL-W_CTRL-R* *E443*
CTRL-W CTRL-R Rotate windows downwards/rightwards. The first window becomes the second one, the second one becomes the third one, etc. The last window becomes the first window. The cursor remains in the same window.
This only works within the row or column of windows that the current window is in.

CTRL-W_R
CTRL-W R Rotate windows upwards/leftwards. The second window becomes the first one, the third one becomes the second one, etc. The first window becomes the last window. The cursor remains in the same window.
This only works within the row or column of windows that the current window is in.

CTRL-W x *CTRL-W_x* *CTRL-W_CTRL-X*
CTRL-W CTRL-X Without count: Exchange current window with next one. If there is no next window, exchange with previous window.
With count: Exchange current window with Nth window (first window is 1). The cursor is put in the other window.
When vertical and horizontal window splits are mixed, the exchange is only done in the row or column of windows that the current window is in.

The following commands can be used to change the window layout. For example, when there are two vertically split windows, CTRL-W K will change that in horizontally split windows. CTRL-W H does it the other way around.

CTRL-W_K
CTRL-W K Move the current window to be at the very top, using the full width of the screen. This works like closing the current window and then creating another one with ":topleft split", except that the current window contents is used for the new window.

CTRL-W_J
CTRL-W J Move the current window to be at the very bottom, using the full width of the screen. This works like closing the current window and then creating another one with ":botright split", except that the current window contents is used for the new window.

CTRL-W_H
CTRL-W H Move the current window to be at the far left, using the full height of the screen. This works like closing the current window and then creating another one with ":vert topleft split", except that the current window contents is used for the new window. {not available when compiled without the |+vertsplit| feature}

CTRL-W_L
CTRL-W L Move the current window to be at the far right, using the full height of the screen. This works like closing the current window and then creating another one with ":vert botright split", except that the current window contents is used for the new window. {not available when compiled without the |+vertsplit| feature}

CTRL-W_T
CTRL-W T Move the current window to a new tab page. This fails if there is only one window in the current tab page. When a count is specified the new tab page will be opened before the tab page with this index. Otherwise it comes after the current tab page.

==
6. Window resizing *window-resize*

CTRL-W_=
CTRL-W = Make all windows (almost) equally high and wide, but use 'winheight' and 'winwidth' for the current window. Windows with 'winfixheight' set keep their height and windows with 'winfixwidth' set keep their width.

:res[ize] -N *:res* *:resize* *CTRL-W_-*
CTRL-W - Decrease current window height by N (default 1). If used after |:vertical|: decrease width by N.

:res[ize] +N *CTRL-W_+*
CTRL-W + Increase current window height by N (default 1). If used after |:vertical|: increase width by N.

:res[ize] [N]
CTRL-W CTRL-_ *CTRL-W_CTRL-_* *CTRL-W__*
CTRL-W _ Set current window height to N (default: highest possible).

z{nr}<CR> Set current window height to {nr}.

CTRL-W_<
CTRL-W < Decrease current window width by N (default 1).

CTRL-W_>
CTRL-W > Increase current window width by N (default 1).

:vertical res[ize] [N] *:vertical-resize* *CTRL-W_bar*
CTRL-W | Set current window width to N (default: widest possible).

You can also resize a window by dragging a status line up or down with the mouse. Or by dragging a vertical separator line left or right. This only works if the version of Vim that is being used supports the mouse and the 'mouse' option has been set to enable it.

The option 'winheight' ('wh') is used to set the minimal window height of the current window. This option is used each time another window becomes the

current window. If the option is '0', it is disabled. Set 'winheight' to a very large value, e.g., '9999', to make the current window always fill all available space. Set it to a reasonable value, e.g., '10', to make editing in the current window comfortable.

The equivalent 'winwidth' ('wiw') option is used to set the minimal width of the current window.

When the option 'equalalways' ('ea') is set, all the windows are automatically made the same size after splitting or closing a window. If you don't set this option, splitting a window will reduce the size of the current window and leave the other windows the same. When closing a window, the extra lines are given to the window above it.

The 'eadirection' option limits the direction in which the 'equalalways' option is applied. The default "both" resizes in both directions. When the value is "ver" only the heights of windows are equalized. Use this when you have manually resized a vertically split window and want to keep this width. Likewise, "hor" causes only the widths of windows to be equalized.

The option 'cmdheight' ('ch') is used to set the height of the command-line. If you are annoyed by the |hit-enter| prompt for long messages, set this option to 2 or 3.

If there is only one window, resizing that window will also change the command line height. If there are several windows, resizing the current window will also change the height of the window below it (and sometimes the window above it).

The minimal height and width of a window is set with 'winminheight' and 'winminwidth'. These are hard values, a window will never become smaller.

==
7. Argument and buffer list commands *buffer-list*

args list	buffer list	meaning ~
1. :[N]argument [N]	11. :[N]buffer [N]	to arg/buf N
2. :[N]next [file ..]	12. :[N]bnext [N]	to Nth next arg/buf
3. :[N]Next [N]	13. :[N]bNext [N]	to Nth previous arg/buf
4. :[N]previous [N]	14. :[N]bprevious [N]	to Nth previous arg/buf
5. :rewind / :first	15. :brewind / :bfirst	to first arg/buf
6. :last	16. :blast	to last arg/buf
7. :all	17. :ball	edit all args/buffers
	18. :unhide	edit all loaded buffers
	19. :[N]bmod [N]	to Nth modified buf

split & args list	split & buffer list	meaning ~
21. :[N]sargument [N]	31. :[N]sbuffer [N]	split + to arg/buf N
22. :[N]snext [file ..]	32. :[N]sbnext [N]	split + to Nth next arg/buf
23. :[N]sNext [N]	33. :[N]sbNext [N]	split + to Nth previous arg/buf
24. :[N]sprevious [N]	34. :[N]sbprevious [N]	split + to Nth previous arg/buf
25. :srewind / :sfirst	35. :sbrewind / :sbfirst	split + to first arg/buf
26. :slast	36. :sblast	split + to last arg/buf
27. :sall	37. :sball	edit all args/buffers
	38. :sunhide	edit all loaded buffers
	39. :[N]sbmod [N]	split + to Nth modified buf

40. :args	list of arguments	
41. :buffers	list of buffers	

The meaning of [N] depends on the command:
 [N] is the number of buffers to go forward/backward on 2/12/22/32,

```
     3/13/23/33, and 4/14/24/34
 [N] is an argument number, defaulting to current argument, for 1 and 21
 [N] is a buffer number, defaulting to current buffer, for 11 and 31
 [N] is a count for 19 and 39

Note: ":next" is an exception, because it must accept a list of file names
for compatibility with Vi.


The argument list and multiple windows
--------------------------------------

The current position in the argument list can be different for each window.
Remember that when doing ":e file", the position in the argument list stays
the same, but you are not editing the file at that position.  To indicate
this, the file message (and the title, if you have one) shows
"(file (N) of M)", where "(N)" is the current position in the file list, and
"M" the number of files in the file list.

All the entries in the argument list are added to the buffer list.  Thus, you
can also get to them with the buffer list commands, like ":bnext".

:[N]al[l][!] [N]                                *:al* *:all* *:sal* *:sall*
:[N]sal[l][!] [N]
                Rearrange the screen to open one window for each argument.
                All other windows are closed.  When a count is given, this is
                the maximum number of windows to open.
                With the |:tab| modifier open a tab page for each argument.
                When there are more arguments than 'tabpagemax' further ones
                become split windows in the last tab page.
                When the 'hidden' option is set, all buffers in closed windows
                become hidden.
                When 'hidden' is not set, and the 'autowrite' option is set,
                modified buffers are written.  Otherwise, windows that have
                buffers that are modified are not removed, unless the [!] is
                given, then they become hidden.  But modified buffers are
                never abandoned, so changes cannot get lost.
                [N] is the maximum number of windows to open.  'winheight'
                also limits the number of windows opened ('winwidth' if
                |:vertical| was prepended).
                Buf/Win Enter/Leave autocommands are not executed for the new
                windows here, that's only done when they are really entered.

:[N]sa[rgument][!] [++opt] [+cmd] [N]                   *:sa* *:sargument*
                Short for ":split | argument [N]": split window and go to Nth
                argument.  But when there is no such argument, the window is
                not split.  Also see |++opt| and |+cmd|.

:[N]sn[ext][!] [++opt] [+cmd] [file ..]                 *:sn* *:snext*
                Short for ":split | [N]next": split window and go to Nth next
                argument.  But when there is no next file, the window is not
                split.  Also see |++opt| and |+cmd|.

:[N]spr[evious][!] [++opt] [+cmd] [N]                   *:spr* *:sprevious*
:[N]sN[ext][!] [++opt] [+cmd] [N]                       *:sN* *:sNext*
                Short for ":split | [N]Next": split window and go to Nth
                previous argument.  But when there is no previous file, the
                window is not split.  Also see |++opt| and |+cmd|.

                                                        *:sre* *:srewind*
:sre[wind][!] [++opt] [+cmd]
                Short for ":split | rewind": split window and go to first
```

```
                argument.  But when there is no argument list, the window is
                not split.  Also see |++opt| and |+cmd|.

                                                *:sfir* *:sfirst*
:sfir[st] [++opt] [+cmd]
                Same as ":srewind".

                                                *:sla* *:slast*
:sla[st][!] [++opt] [+cmd]
                Short for ":split | last": split window and go to last
                argument.  But when there is no argument list, the window is
                not split.  Also see |++opt| and |+cmd|.

                                                *:dr* *:drop*
:dr[op] [++opt] [+cmd] {file} ..
                Edit the first {file} in a window.
                - If the file is already open in a window change to that
                  window.
                - If the file is not open in a window edit the file in the
                  current window.  If the current buffer can't be |abandon|ed,
                  the window is split first.
                - Windows that are not in the argument list or are not full
                  width will be closed if possible.
                The |argument-list| is set, like with the |:next| command.
                The purpose of this command is that it can be used from a
                program that wants Vim to edit another file, e.g., a debugger.
                When using the |:tab| modifier each argument is opened in a
                tab page.  The last window is used if it's empty.
                Also see |++opt| and |+cmd|.
                {only available when compiled with a GUI}

==============================================================================
8. Do a command in all buffers or windows                       *list-repeat*

                                                        *:windo*
:[range]windo {cmd}     Execute {cmd} in each window or if [range] is given
                        only in windows for which the window number lies in
                        the [range].  It works like doing this: >
                                CTRL-W t
                                :{cmd}
                                CTRL-W w
                                :{cmd}
                                etc.
<                       This only operates in the current tab page.
                        When an error is detected on one window, further
                        windows will not be visited.
                        The last window (or where an error occurred) becomes
                        the current window.
                        {cmd} can contain '|' to concatenate several commands.
                        {cmd} must not open or close windows or reorder them.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}
                        Also see |:tabdo|, |:argdo|, |:bufdo|, |:cdo|, |:ldo|,
                        |:cfdo| and |:lfdo|

                                                        *:bufdo*
:[range]bufdo[!] {cmd}  Execute {cmd} in each buffer in the buffer list or if
                        [range] is given only for buffers for which their
                        buffer number is in the [range].  It works like doing
                        this: >
                                :bfirst
                                :{cmd}
```

```
                                :bnext
                                :{cmd}
                                etc.
<                       When the current file can't be |abandon|ed and the [!]
                        is not present, the command fails.
                        When an error is detected on one buffer, further
                        buffers will not be visited.
                        Unlisted buffers are skipped.
                        The last buffer (or where an error occurred) becomes
                        the current buffer.
                        {cmd} can contain '|' to concatenate several commands.
                        {cmd} must not delete buffers or add buffers to the
                        buffer list.
                        Note: While this command is executing, the Syntax
                        autocommand event is disabled by adding it to
                        'eventignore'.  This considerably speeds up editing
                        each buffer.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}
                        Also see |:tabdo|, |:argdo|, |:windo|, |:cdo|, |:ldo|,
                        |:cfdo| and |:lfdo|

Examples: >

        :windo set nolist nofoldcolumn | normal zn

This resets the 'list' option and disables folding in all windows. >

        :bufdo set fileencoding= | update

This resets the 'fileencoding' in each buffer and writes it if this changed
the buffer.  The result is that all buffers will use the 'encoding' encoding
(if conversion works properly).

==============================================================================
9. Tag or file name under the cursor                    *window-tag*

                                                        *:sta* *:stag*
:sta[g][!] [tagname]
                Does ":tag[!] [tagname]" and splits the window for the found
                tag.  See also |:tag|.

CTRL-W ]                                        *CTRL-W_]* *CTRL-W_CTRL-]*
CTRL-W CTRL-]   Split current window in two.  Use identifier under cursor as a
                tag and jump to it in the new upper window.
                In Visual mode uses the Visually selected text as a tag.
                Make new window N high.

                                                        *CTRL-W_g]*
CTRL-W g ]      Split current window in two.  Use identifier under cursor as a
                tag and perform ":tselect" on it in the new upper window.
                In Visual mode uses the Visually selected text as a tag.
                Make new window N high.

                                                        *CTRL-W_g_CTRL-]*
CTRL-W g CTRL-] Split current window in two.  Use identifier under cursor as a
                tag and perform ":tjump" on it in the new upper window.
                In Visual mode uses the Visually selected text as a tag.
                Make new window N high.

CTRL-W f                                        *CTRL-W_f* *CTRL-W_CTRL-F*
CTRL-W CTRL-F   Split current window in two.  Edit file name under cursor.
```

Like ":split gf", but window isn't split if the file does not
exist.
Uses the 'path' variable as a list of directory names where to
look for the file. Also the path for current file is
used to search for the file name.
If the name is a hypertext link that looks like
"type://machine/path", only "/path" is used.
If a count is given, the count'th matching file is edited.
{not available when the |+file_in_path| feature was disabled
at compile time}

CTRL-W F *CTRL-W_F*
Split current window in two. Edit file name under cursor and
jump to the line number following the file name. See |gF| for
details on how the line number is obtained.
{not available when the |+file_in_path| feature was disabled
at compile time}

CTRL-W gf *CTRL-W_gf*
Open a new tab page and edit the file name under the cursor.
Like "tab split" and "gf", but the new tab page isn't created
if the file does not exist.
{not available when the |+file_in_path| feature was disabled
at compile time}

CTRL-W gF *CTRL-W_gF*
Open a new tab page and edit the file name under the cursor
and jump to the line number following the file name. Like
"tab split" and "gF", but the new tab page isn't created if
the file does not exist.
{not available when the |+file_in_path| feature was disabled
at compile time}

Also see |CTRL-W_CTRL-I|: open window for an included file that includes
the keyword under the cursor.

==
10. The preview window *preview-window*

The preview window is a special window to show (preview) another file. It is
normally a small window used to show an include file or definition of a
function.
{not available when compiled without the |+quickfix| feature}

There can be only one preview window (per tab page). It is created with one
of the commands below. The 'previewheight' option can be set to specify the
height of the preview window when it's opened. The 'previewwindow' option is
set in the preview window to be able to recognize it. The 'winfixheight'
option is set to have it keep the same height when opening/closing other
windows.

:pta *:ptag*
:pta[g][!] [tagname]
Does ":tag[!] [tagname]" and shows the found tag in a
"Preview" window without changing the current buffer or cursor
position. If a "Preview" window already exists, it is re-used
(like a help window is). If a new one is opened,
'previewheight' is used for the height of the window. See
also |:tag|.
See below for an example. |CursorHold-example|
Small difference from |:tag|: When [tagname] is equal to the
already displayed tag, the position in the matching tag list

is not reset. This makes the CursorHold example work after a
|:ptnext|.

CTRL-W z *CTRL-W_z*
CTRL-W CTRL-Z *CTRL-W_CTRL-Z* *:pc* *:pclose*
:pc[lose][!] Close any "Preview" window currently open. When the 'hidden'
option is set, or when the buffer was changed and the [!] is
used, the buffer becomes hidden (unless there is another
window editing it). The command fails if any "Preview" buffer
cannot be closed. See also |:close|.

:pp *:ppop*
:[count]pp[op][!]
Does ":[count]pop[!]" in the preview window. See |:pop| and
|:ptag|. {not in Vi}

CTRL-W } *CTRL-W_}*
Use identifier under cursor as a tag and perform a :ptag on
it. Make the new Preview window (if required) N high. If N is
not given, 'previewheight' is used.

CTRL-W g } *CTRL-W_g}*
Use identifier under cursor as a tag and perform a :ptjump on
it. Make the new Preview window (if required) N high. If N is
not given, 'previewheight' is used.

:ped *:pedit*
:ped[it][!] [++opt] [+cmd] {file}
Edit {file} in the preview window. The preview window is
opened like with |:ptag|. The current window and cursor
position isn't changed. Useful example: >
:pedit +/fputc /usr/include/stdio.h
<
:ps *:psearch*
:[range]ps[earch][!] [count] [/]pattern[/]
Works like |:ijump| but shows the found match in the preview
window. The preview window is opened like with |:ptag|. The
current window and cursor position isn't changed. Useful
example: >
:psearch popen
< Like with the |:ptag| command, you can use this to
automatically show information about the word under the
cursor. This is less clever than using |:ptag|, but you don't
need a tags file and it will also find matches in system
include files. Example: >
:au! CursorHold *.[ch] nested exe "silent! psearch " . expand("<cword>")
< Warning: This can be slow.

Example *CursorHold-example* >

:au! CursorHold *.[ch] nested exe "silent! ptag " . expand("<cword>")

This will cause a ":ptag" to be executed for the keyword under the cursor,
when the cursor hasn't moved for the time set with 'updatetime'. The "nested"
makes other autocommands be executed, so that syntax highlighting works in the
preview window. The "silent!" avoids an error message when the tag could not
be found. Also see |CursorHold|. To disable this again: >

:au! CursorHold

A nice addition is to highlight the found tag, avoid the ":ptag" when there
is no word under the cursor, and a few other things: >

```
  :au! CursorHold *.[ch] nested call PreviewWord()
  :func PreviewWord()
  :  if &previewwindow                  " don't do this in the preview window
  :    return
  :  endif
  :  let w = expand("<cword>")          " get the word under cursor
  :  if w =~ '\a'                       " if the word contains a letter
  :
  :    " Delete any existing highlight before showing another tag
  :    silent! wincmd P                 " jump to preview window
  :    if &previewwindow                       " if we really get there...
  :      match none                     " delete existing highlight
  :      wincmd p                       " back to old window
  :    endif
  :
  :    " Try displaying a matching tag for the word under the cursor
  :    try
  :       exe "ptag " . w
  :    catch
  :      return
  :    endtry
  :
  :    silent! wincmd P                 " jump to preview window
  :    if &previewwindow                " if we really get there...
  :      if has("folding")
  :        silent! .foldopen            " don't want a closed fold
  :      endif
  :      call search("$", "b")          " to end of previous line
  :      let w = substitute(w, '\\', '\\\\', "")
  :      call search('\<\V' . w . '\>') " position cursor on match
  :      " Add a match highlight to the word at this position
  :      hi previewWord term=bold ctermbg=green guibg=green
  :      exe 'match previewWord "\%' . line(".") . 'l\%' . col(".") . 'c\k*"'
  :      wincmd p                       " back to old window
  :    endif
  :  endif
  :endfun
```

```
==============================================================================
11. Using hidden buffers                                *buffer-hidden*

A hidden buffer is not displayed in a window, but is still loaded into memory.
This makes it possible to jump from file to file, without the need to read or
write the file every time you get another buffer in a window.
{not available when compiled without the |+listcmds| feature}

                                                        *:buffer-!*
If the option 'hidden' ('hid') is set, abandoned buffers are kept for all
commands that start editing another file: ":edit", ":next", ":tag", etc.  The
commands that move through the buffer list sometimes make the current buffer
hidden although the 'hidden' option is not set.  This happens when a buffer is
modified, but is forced (with '!') to be removed from a window, and
'autowrite' is off or the buffer can't be written.

You can make a hidden buffer not hidden by starting to edit it with any
command.  Or by deleting it with the ":bdelete" command.

The 'hidden' is global, it is used for all buffers.  The 'bufhidden' option
can be used to make an exception for a specific buffer.  It can take these
values:
        <empty>         Use the value of 'hidden'.
```

```
        hide            Hide this buffer, also when 'hidden' is not set.
        unload          Don't hide but unload this buffer, also when 'hidden'
                        is set.
        delete          Delete the buffer.

                                                        *hidden-quit*
When you try to quit Vim while there is a hidden, modified buffer, you will
get an error message and Vim will make that buffer the current buffer.  You
can then decide to write this buffer (":wq") or quit without writing (":q!").
Be careful: there may be more hidden, modified buffers!

A buffer can also be unlisted.  This means it exists, but it is not in the
list of buffers. |unlisted-buffer|


:files[!] [flags]                               *:files*
:buffers[!] [flags]                             *:buffers* *:ls*
:ls[!] [flags]
                Show all buffers.  Example:

                        1 #h   "/test/text"             line 1 ~
                        2u     "asdf"                   line 0 ~
                        3 %a + "version.c"              line 1 ~

                When the [!] is included the list will show unlisted buffers
                (the term "unlisted" is a bit confusing then...).

                Each buffer has a unique number.  That number will not change,
                thus you can always go to a specific buffer with ":buffer N"
                or "N CTRL-^", where N is the buffer number.

                Indicators (chars in the same column are mutually exclusive):
                u       an unlisted buffer (only displayed when [!] is used)
                           |unlisted-buffer|
                 %      the buffer in the current window
                 #      the alternate buffer for ":e #" and CTRL-^
                  a     an active buffer: it is loaded and visible
                  h     a hidden buffer: It is loaded, but currently not
                           displayed in a window |hidden-buffer|
                   -    a buffer with 'modifiable' off
                   =    a readonly buffer
                   R    a terminal buffer with a running job
                   F    a terminal buffer with a finished job
                   ?    a terminal buffer without a job: `:terminal NONE`
                    +   a modified buffer
                    x   a buffer with read errors

                [flags] can be a combination of the following characters,
                which restrict the buffers to be listed:
                     +   modified buffers
                     -   buffers with 'modifiable' off
                     =   readonly buffers
                     a   active buffers
                     u   unlisted buffers (overrides the "!")
                     h   hidden buffers
                     x   buffers with a read error
                     %   current buffer
                     #   alternate buffer
                Combining flags means they are "and"ed together, e.g.:
                     h+   hidden buffers which are modified
                     a+   active buffers which are modified
```

```
                When using |:filter| the pattern is matched against the
                displayed buffer name, e.g.: >
                        filter /\.vim/ ls
<
                                                *:bad* *:badd*
:bad[d] [+lnum] {fname}
                Add file name {fname} to the buffer list, without loading it.
                If "lnum" is specified, the cursor will be positioned at that
                line when the buffer is first entered.  Note that other
                commands after the + will be ignored.

:[N]bd[elete][!]                        *:bd* *:bdel* *:bdelete* *E516*
:bd[elete][!] [N]
                Unload buffer [N] (default: current buffer) and delete it from
                the buffer list.  If the buffer was changed, this fails,
                unless when [!] is specified, in which case changes are lost.
                The file remains unaffected.  Any windows for this buffer are
                closed.  If buffer [N] is the current buffer, another buffer
                will be displayed instead.  This is the most recent entry in
                the jump list that points into a loaded buffer.
                Actually, the buffer isn't completely deleted, it is removed
                from the buffer list |unlisted-buffer| and option values,
                variables and mappings/abbreviations for the buffer are
                cleared. Examples: >
                    :.,$-bdelete    " delete buffers from the current one to
                                    " last but one
                    :%bdelete       " delete all buffers
<

:bdelete[!] {bufname}                                           *E93* *E94*
                Like ":bdelete[!] [N]", but buffer given by name.  Note that a
                buffer whose name is a number cannot be referenced by that
                name; use the buffer number instead.  Insert a backslash
                before a space in a buffer name.

:bdelete[!] N1 N2 ...
                Do ":bdelete[!]" for buffer N1, N2, etc.  The arguments can be
                buffer numbers or buffer names (but not buffer names that are
                a number).  Insert a backslash before a space in a buffer
                name.

:N,Mbdelete[!]  Do ":bdelete[!]" for all buffers in the range N to M
                |inclusive|.

:[N]bw[ipeout][!]                       *:bw* *:bwipe* *:bwipeout* *E517*
:bw[ipeout][!] {bufname}
:N,Mbw[ipeout][!]
:bw[ipeout][!] N1 N2 ...
                Like |:bdelete|, but really delete the buffer.  Everything
                related to the buffer is lost.  All marks in this buffer
                become invalid, option settings are lost, etc.  Don't use this
                unless you know what you are doing. Examples: >
                    :.+,$bwipeout   " wipe out all buffers after the current
                                    " one
                    :%bwipeout      " wipe out all buffers
<

:[N]bun[load][!]                                *:bun* *:bunload* *E515*
:bun[load][!] [N]
                Unload buffer [N] (default: current buffer).  The memory
                allocated for this buffer will be freed.  The buffer remains
                in the buffer list.
```

```
                If the buffer was changed, this fails, unless when [!] is
                specified, in which case the changes are lost.
                Any windows for this buffer are closed.  If buffer [N] is the
                current buffer, another buffer will be displayed instead.
                This is the most recent entry in the jump list that points
                into a loaded buffer.

:bunload[!] {bufname}
                Like ":bunload[!] [N]", but buffer given by name.  Note that a
                buffer whose name is a number cannot be referenced by that
                name; use the buffer number instead.  Insert a backslash
                before a space in a buffer name.

:N,Mbunload[!]  Do ":bunload[!]" for all buffers in the range N to M
                |inclusive|.

:bunload[!] N1 N2 ...
                Do ":bunload[!]" for buffer N1, N2, etc.  The arguments can be
                buffer numbers or buffer names (but not buffer names that are
                a number).  Insert a backslash before a space in a buffer
                name.

:[N]b[uffer][!] [+cmd] [N]              *:b* *:bu* *:buf* *:buffer* *E86*
                Edit buffer [N] from the buffer list.  If [N] is not given,
                the current buffer remains being edited.  See |:buffer-!| for
                [!].  This will also edit a buffer that is not in the buffer
                list, without setting the 'buflisted' flag.
                Also see |+cmd|.

:[N]b[uffer][!] [+cmd] {bufname}
                Edit buffer for {bufname} from the buffer list.  See
                |:buffer-!| for [!].  This will also edit a buffer that is not
                in the buffer list, without setting the 'buflisted' flag.
                Also see |+cmd|.

:[N]sb[uffer] [+cmd] [N]                                *:sb* *:sbuffer*
                Split window and edit buffer [N] from the buffer list.  If [N]
                is not given, the current buffer is edited.  Respects the
                "useopen" setting of 'switchbuf' when splitting.  This will
                also edit a buffer that is not in the buffer list, without
                setting the 'buflisted' flag.
                Also see |+cmd|.

:[N]sb[uffer] [+cmd] {bufname}
                Split window and edit buffer for {bufname} from the buffer
                list.  This will also edit a buffer that is not in the buffer
                list, without setting the 'buflisted' flag.
                Note: If what you want to do is split the buffer, make a copy
                under another name, you can do it this way: >
                        :w foobar | sp #
<               Also see |+cmd|.

:[N]bn[ext][!] [+cmd] [N]                               *:bn* *:bnext* *E87*
                Go to [N]th next buffer in buffer list.  [N] defaults to one.
                Wraps around the end of the buffer list.
                See |:buffer-!| for [!].
                Also see |+cmd|.
                If you are in a help buffer, this takes you to the next help
                buffer (if there is one).  Similarly, if you are in a normal
                (non-help) buffer, this takes you to the next normal buffer.
                This is so that if you have invoked help, it doesn't get in
                the way when you're browsing code/text buffers.  The next three
```

commands also work like this.

:sbn *:sbnext*
:[N]sbn[ext] [+cmd] [N]
Split window and go to [N]th next buffer in buffer list.
Wraps around the end of the buffer list. Uses 'switchbuf'
Also see |+cmd|.

:[N]bN[ext][!] [+cmd] [N] *:bN* *:bNext* *:bp* *:bprevious* *E88*
:[N]bp[revious][!] [+cmd] [N]
Go to [N]th previous buffer in buffer list. [N] defaults to
one. Wraps around the start of the buffer list.
See |:buffer-!| for [!] and 'switchbuf'.
Also see |+cmd|.

:[N]sbN[ext] [+cmd] [N] *:sbN* *:sbNext* *:sbp* *:sbprevious*
:[N]sbp[revious] [+cmd] [N]
Split window and go to [N]th previous buffer in buffer list.
Wraps around the start of the buffer list.
Uses 'switchbuf'.
Also see |+cmd|.

:br[ewind][!] [+cmd] *:br* *:brewind*
Go to first buffer in buffer list. If the buffer list is
empty, go to the first unlisted buffer.
See |:buffer-!| for [!].

:bf[irst] [+cmd] *:bf* *:bfirst*
Same as |:brewind|.
Also see |+cmd|.

:sbr[ewind] [+cmd] *:sbr* *:sbrewind*
Split window and go to first buffer in buffer list. If the
buffer list is empty, go to the first unlisted buffer.
Respects the 'switchbuf' option.
Also see |+cmd|.

:sbf[irst] [+cmd] *:sbf* *:sbfirst*
Same as ":sbrewind".

:bl[ast][!] [+cmd] *:bl* *:blast*
Go to last buffer in buffer list. If the buffer list is
empty, go to the last unlisted buffer.
See |:buffer-!| for [!].

:sbl[ast] [+cmd] *:sbl* *:sblast*
Split window and go to last buffer in buffer list. If the
buffer list is empty, go to the last unlisted buffer.
Respects 'switchbuf' option.

:[N]bm[odified][!] [+cmd] [N] *:bm* *:bmodified* *E84*
Go to [N]th next modified buffer. Note: this command also
finds unlisted buffers. If there is no modified buffer the
command fails.

:[N]sbm[odified] [+cmd] [N] *:sbm* *:sbmodified*
Split window and go to [N]th next modified buffer.
Respects 'switchbuf' option.
Note: this command also finds buffers not in the buffer list.

:[N]unh[ide] [N] *:unh* *:unhide* *:sun* *:sunhide*

```
:[N]sun[hide] [N]
                Rearrange the screen to open one window for each loaded buffer
                in the buffer list.  When a count is given, this is the
                maximum number of windows to open.

:[N]ba[ll] [N]                                  *:ba* *:ball* *:sba* *:sball*
:[N]sba[ll] [N] Rearrange the screen to open one window for each buffer in
                the buffer list.  When a count is given, this is the maximum
                number of windows to open.  'winheight' also limits the number
                of windows opened ('winwidth' if |:vertical| was prepended).
                Buf/Win Enter/Leave autocommands are not executed for the new
                windows here, that's only done when they are really entered.
                When the |:tab| modifier is used new windows are opened in a
                new tab, up to 'tabpagemax'.

Note: All the commands above that start editing another buffer, keep the
'readonly' flag as it was.  This differs from the ":edit" command, which sets
the 'readonly' flag each time the file is read.

==============================================================================
12. Special kinds of buffers                    *special-buffers*

Instead of containing the text of a file, buffers can also be used for other
purposes.  A few options can be set to change the behavior of a buffer:
        'bufhidden'     what happens when the buffer is no longer displayed
                        in a window.
        'buftype'       what kind of a buffer this is
        'swapfile'      whether the buffer will have a swap file
        'buflisted'     buffer shows up in the buffer list

A few useful kinds of a buffer:

quickfix        Used to contain the error list or the location list.  See
                |:cwindow| and |:lwindow|.  This command sets the 'buftype'
                option to "quickfix".  You are not supposed to change this!
                'swapfile' is off.

help            Contains a help file.  Will only be created with the |:help|
                command.  The flag that indicates a help buffer is internal
                and can't be changed.  The 'buflisted' option will be reset
                for a help buffer.

terminal        A terminal window buffer, see |terminal|. The contents cannot
                be read or changed until the job ends.

directory       Displays directory contents.  Can be used by a file explorer
                plugin.  The buffer is created with these settings: >
                        :setlocal buftype=nowrite
                        :setlocal bufhidden=delete
                        :setlocal noswapfile
<               The buffer name is the name of the directory and is adjusted
                when using the |:cd| command.

scratch         Contains text that can be discarded at any time.  It is kept
                when closing the window, it must be deleted explicitly.
                Settings: >
                        :setlocal buftype=nofile
                        :setlocal bufhidden=hide
                        :setlocal noswapfile
<               The buffer name can be used to identify the buffer, if you
                give it a meaningful name.
```

```
                                                *unlisted-buffer*
unlisted        The buffer is not in the buffer list.  It is not used for
                normal editing, but to show a help file, remember a file name
                or marks.  The ":bdelete" command will also set this option,
                thus it doesn't completely delete the buffer.  Settings: >
                        :setlocal nobuflisted
<

 vim:tw=78:ts=8:ft=help:norl:
*tabpage.txt*   For Vim version 8.0.  Last change: 2016 Oct 20


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Editing with windows in multiple tab pages.             *tab-page* *tabpage*

The commands which have been added to use multiple tab pages are explained
here.  Additionally, there are explanations for commands that work differently
when used in combination with more than one tab page.

1. Introduction                 |tab-page-intro|
2. Commands                     |tab-page-commands|
3. Other items                  |tab-page-other|
4. Setting 'tabline'            |setting-tabline|
5. Setting 'guitablabel'        |setting-guitablabel|

{Vi does not have any of these commands}
{not able to use multiple tab pages when the |+windows| feature was disabled
at compile time}

==============================================================================
1. Introduction                                         *tab-page-intro*

A tab page holds one or more windows.  You can easily switch between tab
pages, so that you have several collections of windows to work on different
things.

Usually you will see a list of labels at the top of the Vim window, one for
each tab page.  With the mouse you can click on the label to jump to that tab
page.  There are other ways to move between tab pages, see below.

Most commands work only in the current tab page.  That includes the |CTRL-W|
commands, |:windo|, |:all| and |:ball| (when not using the |:tab| modifier).
The commands that are aware of other tab pages than the current one are
mentioned below.

Tabs are also a nice way to edit a buffer temporarily without changing the
current window layout.  Open a new tab page, do whatever you want to do and
close the tab page.

==============================================================================
2. Commands                                             *tab-page-commands*

OPENING A NEW TAB PAGE:

When starting Vim "vim -p filename ..." opens each file argument in a separate
tab page (up to 'tabpagemax').  See |-p|

A double click with the mouse in the non-GUI tab pages line opens a new, empty
tab page.  It is placed left of the position of the click.  The first click
may select another tab page first, causing an extra screen update.
```

```
This also works in a few GUI versions, esp. Win32 and Motif.  But only when
clicking right of the labels.

In the GUI tab pages line you can use the right mouse button to open menu.
|tabline-menu|.

For the related autocommands see |tabnew-autocmd|.

:[count]tabe[dit]                                *:tabe* *:tabedit* *:tabnew*
:[count]tabnew
               Open a new tab page with an empty window, after the current
               tab page.  If [count] is given the new tab page appears after
               the tab page [count] otherwise the new tab page will appear
               after the current one. >
                   :tabnew     " opens tabpage after the current one
                   :.tabnew    " as above
                   :+tabnew    " opens tabpage after the next tab page
                               " note: it is one further than :tabnew
                   :-tabnew    " opens tabpage before the current one
                   :0tabnew    " opens tabpage before the first one
                   :$tabnew    " opens tabpage after the last one

:[count]tabe[dit] [++opt] [+cmd] {file}
:[count]tabnew [++opt] [+cmd] {file}
               Open a new tab page and edit {file}, like with |:edit|.
               For [count] see |:tabnew| above.

:[count]tabf[ind] [++opt] [+cmd] {file}                *:tabf* *:tabfind*
               Open a new tab page and edit {file} in 'path', like with
               |:find|.  For [count] see |:tabnew| above.
               {not available when the |+file_in_path| feature was disabled
               at compile time}

:[count]tab {cmd}                                      *:tab*
               Execute {cmd} and when it opens a new window open a new tab
               page instead.  Doesn't work for |:diffsplit|, |:diffpatch|,
               |:execute| and |:normal|.
               If [count] is given the new tab page appears after the tab
               page [count] otherwise the new tab page will appear after the
               current one.
               Examples: >
                   :tab split      " opens current buffer in new tab page
                   :tab help gt    " opens tab page with help for "gt"
                   :.tab help gt   " as above
                   :+tab help      " opens tab page with help after the next
                                   " tab page
                   :-tab help      " opens tab page with help before the
                                   " current one
                   :0tab help      " opens tab page with help before the
                                   " first one
                   :$tab help      " opens tab page with help after the last
                                   " one

CTRL-W gf      Open a new tab page and edit the file name under the cursor.
               See |CTRL-W_gf|.

CTRL-W gF      Open a new tab page and edit the file name under the cursor
               and jump to the line number following the file name.
               See |CTRL-W_gF|.

CLOSING A TAB PAGE:
```

```
Closing the last window of a tab page closes the tab page too, unless there is
only one tab page.

Using the mouse: If the tab page line is displayed you can click in the "X" at
the top right to close the current tab page.  A custom |'tabline'| may show
something else.

                                                        *:tabc* *:tabclose*
:tabc[lose][!]  Close current tab page.
                This command fails when:
                - There is only one tab page on the screen.             *E784*
                - When 'hidden' is not set, [!] is not used, a buffer has
                  changes, and there is no other window on this buffer.
                Changes to the buffer are not written and won't get lost, so
                this is a "safe" command. >
                    :tabclose       " close the current tab page

:{count}tabc[lose][!]
:tabc[lose][!] {count}
                Close tab page {count}.  Fails in the same way as `:tabclose`
                above. >
                    :-tabclose      " close the previous tab page
                    :+tabclose      " close the next tab page
                    :1tabclose      " close the first tab page
                    :$tabclose      " close the last tab page
                    :tabclose -2    " close the two previous tab page
                    :tabclose +     " close the next tab page
                    :tabclose 3     " close the third tab page
                    :tabclose $     " close the last tab page
<
                                                        *:tabo* *:tabonly*
:tabo[nly][!]   Close all other tab pages.
                When the 'hidden' option is set, all buffers in closed windows
                become hidden.
                When 'hidden' is not set, and the 'autowrite' option is set,
                modified buffers are written.  Otherwise, windows that have
                buffers that are modified are not removed, unless the [!] is
                given, then they become hidden.  But modified buffers are
                never abandoned, so changes cannot get lost. >
                    :tabonly        " close all tab pages except the current
                                    " one

:{count}tabo[nly][!]
:tabo[nly][!] {count}
                Close all tab pages except {count} one. >
                    :.tabonly       " as above
                    :-tabonly       " close all tab pages except the previous
                                    " one
                    :+tabonly       " close all tab pages except the next one
                    :1tabonly       " close all tab pages except the first one
                    :$tabonly       " close all tab pages except the last one
                    :tabonly -      " close all tab pages except the previous
                                    " one
                    :tabonly +2     " close all tab pages except the two next
                                    " one
                    :tabonly 1      " close all tab pages except the first one
                    :tabonly $      " close all tab pages except the last one


SWITCHING TO ANOTHER TAB PAGE:
```

```
Using the mouse: If the tab page line is displayed you can click in a tab page
label to switch to that tab page.  Click where there is no label to go to the
next tab page.  |'tabline'|

:tabn[ext]                              *:tabn* *:tabnext* *gt*
<C-PageDown>                            *CTRL-<PageDown>* *<C-PageDown>*
gt                                      *i_CTRL-<PageDown>* *i_<C-PageDown>*
                Go to the next tab page.  Wraps around from the last to the
                first one.

:{count}tabn[ext]
:tabn[ext] {count}
                Go to tab page {count}.  The first tab page has number one. >
                    :-tabnext   " go to the previous tab page
                    :+tabnext   " go to the next tab page
                    :+2tabnext  " go to the two next tab page
                    :1tabnext   " go to the first tab page
                    :$tabnext   " go to the last tab page
                    :tabnext $  " as above
                    :tabnext -  " go to the previous tab page
                    :tabnext -1 " as above
                    :tabnext +  " go to the next tab page
                    :tabnext +1 " as above

{count}<C-PageDown>
{count}gt       Go to tab page {count}.  The first tab page has number one.


:tabp[revious]                          *:tabp* *:tabprevious* *gT* *:tabN*
:tabN[ext]                              *:tabNext* *CTRL-<PageUp>*
<C-PageUp>                        *<C-PageUp>* *i_CTRL-<PageUp>* *i_<C-PageUp>*
gT              Go to the previous tab page.  Wraps around from the first one
                to the last one.

:tabp[revious] {count}
:tabN[ext] {count}
{count}<C-PageUp>
{count}gT       Go {count} tab pages back.  Wraps around from the first one
                to the last one.

:tabr[ewind]                    *:tabfir* *:tabfirst* *:tabr* *:tabrewind*
:tabfir[st]     Go to the first tab page.

                                                        *:tabl* *:tablast*
:tabl[ast]      Go to the last tab page.


Other commands:
                                                        *:tabs*
:tabs           List the tab pages and the windows they contain.
                Shows a ">" for the current window.
                Shows a "+" for modified buffers.
                For example:
                        Tab page 1 ~
                          + tabpage.txt ~
                            ex_docmd.c ~
                        Tab page 2 ~
                        >   main.c ~


REORDERING TAB PAGES:
```

```
:tabm[ove] [N]                                          *:tabm* *:tabmove*
:[N]tabm[ove]
                Move the current tab page to after tab page N.  Use zero to
                make the current tab page the first one.  Without N the tab
                page is made the last one. >
                    :.tabmove   " do nothing
                    :-tabmove   " move the tab page to the left
                    :+tabmove   " move the tab page to the right
                    :0tabmove   " move the tab page to the beginning of the tab
                                " list
                    :tabmove 0  " as above
                    :tabmove    " move the tab page to the last
                    :$tabmove   " as above
                    :tabmove $  " as above

:tabm[ove] +[N]
:tabm[ove] -[N]
                Move the current tab page N places to the right (with +) or to
                the left (with -). >
                    :tabmove -  " move the tab page to the left
                    :tabmove -1 " as above
                    :tabmove +  " move the tab page to the right
                    :tabmove +1 " as above


Note that although it is possible to move a tab behind the N-th one by using
:Ntabmove. And move it by N places by using :+Ntabmove. For clarification what
+N means in this context see |[range]|.


LOOPING OVER TAB PAGES:

                                                        *:tabd* *:tabdo*
:[range]tabd[o] {cmd}
                Execute {cmd} in each tab page or if [range] is given only in
                tab pages which tab page number is in the [range].  It works
                like doing this: >
                        :tabfirst
                        :{cmd}
                        :tabnext
                        :{cmd}
                        etc.
<               This only operates in the current window of each tab page.
                When an error is detected on one tab page, further tab pages
                will not be visited.
                The last tab page (or where an error occurred) becomes the
                current tab page.
                {cmd} can contain '|' to concatenate several commands.
                {cmd} must not open or close tab pages or reorder them.
                {not in Vi} {not available when compiled without the
                |+listcmds| feature}
                Also see |:windo|, |:argdo|, |:bufdo|, |:cdo|, |:ldo|, |:cfdo|
                and |:lfdo|

==============================================================================
3. Other items                                          *tab-page-other*

                                                        *tabline-menu*
The GUI tab pages line has a popup menu.  It is accessed with a right click.
The entries are:
        Close           Close the tab page under the mouse pointer.  The
                        current one if there is no label under the mouse
```

```
                        pointer.
        New Tab         Open a tab page, editing an empty buffer.  It appears
                        to the left of the mouse pointer.
        Open Tab...     Like "New Tab" and additionally use a file selector to
                        select a file to edit.

Diff mode works per tab page.  You can see the diffs between several files
within one tab page.  Other tab pages can show differences between other
files.

Variables local to a tab page start with "t:". |tabpage-variable|

Currently there is only one option local to a tab page: 'cmdheight'.

                                                    *tabnew-autocmd*
The TabLeave and TabEnter autocommand events can be used to do something when
switching from one tab page to another.  The exact order depends on what you
are doing.  When creating a new tab page this works as if you create a new
window on the same buffer and then edit another buffer.  Thus ":tabnew"
triggers:
        WinLeave                leave current window
        TabLeave                leave current tab page
        WinEnter                enter window in new tab page
        TabEnter                enter new tab page
        BufLeave                leave current buffer
        BufEnter                enter new empty buffer

When switching to another tab page the order is:
        BufLeave
        WinLeave
        TabLeave
        TabEnter
        WinEnter
        BufEnter

==============================================================================
4. Setting 'tabline'                                    *setting-tabline*

The 'tabline' option specifies what the line with tab pages labels looks like.
It is only used when there is no GUI tab line.

You can use the 'showtabline' option to specify when you want the line with
tab page labels to appear: never, when there is more than one tab page or
always.

The highlighting of the tab pages line is set with the groups TabLine
TabLineSel and TabLineFill.  |hl-TabLine| |hl-TabLineSel| |hl-TabLineFill|

A "+" will be shown for a tab page that has a modified window.  The number of
windows in a tabpage is also shown.  Thus "3+" means three windows and one of
them has a modified buffer.

The 'tabline' option allows you to define your preferred way to tab pages
labels.  This isn't easy, thus an example will be given here.

For basics see the 'statusline' option.  The same items can be used in the
'tabline' option.  Additionally, the |tabpagebuflist()|, |tabpagenr()| and
|tabpagewinnr()| functions are useful.

Since the number of tab labels will vary, you need to use an expression for
the whole option.  Something like: >
        :set tabline=%!MyTabLine()
```

Then define the MyTabLine() function to list all the tab pages labels. A convenient method is to split it in two parts: First go over all the tab pages and define labels for them. Then get the label for each tab page. >

```
        function MyTabLine()
          let s = ''
          for i in range(tabpagenr('$'))
            " select the highlighting
            if i + 1 == tabpagenr()
              let s .= '%#TabLineSel#'
            else
              let s .= '%#TabLine#'
            endif

            " set the tab page number (for mouse clicks)
            let s .= '%' . (i + 1) . 'T'

            " the label is made by MyTabLabel()
            let s .= ' %{MyTabLabel(' . (i + 1) . ')} '
          endfor

          " after the last tab fill with TabLineFill and reset tab page nr
          let s .= '%#TabLineFill#%T'

          " right-align the label to close the current tab page
          if tabpagenr('$') > 1
            let s .= '%=%#TabLine#%999Xclose'
          endif

          return s
        endfunction
```

Now the MyTabLabel() function is called for each tab page to get its label. >

```
        function MyTabLabel(n)
          let buflist = tabpagebuflist(a:n)
          let winnr = tabpagewinnr(a:n)
          return bufname(buflist[winnr - 1])
        endfunction
```

This is just a simplistic example that results in a tab pages line that resembles the default, but without adding a + for a modified buffer or truncating the names. You will want to reduce the width of labels in a clever way when there is not enough room. Check the 'columns' option for the space available.

==

5. Setting 'guitablabel' *setting-guitablabel*

When the GUI tab pages line is displayed, 'guitablabel' can be used to specify the label to display for each tab page. Unlike 'tabline', which specifies the whole tab pages line at once, 'guitablabel' is used for each label separately.

'guitabtooltip' is very similar and is used for the tooltip of the same label. This only appears when the mouse pointer hovers over the label, thus it usually is longer. Only supported on some systems though.

See the 'statusline' option for the format of the value.

The "%N" item can be used for the current tab page number. The |v:lnum|

```
variable is also set to this number when the option is evaluated.
The items that use a file name refer to the current window of the tab page.

Note that syntax highlighting is not used for the option.  The %T and %X
items are also ignored.

A simple example that puts the tab page number and the buffer name in the
label: >
        :set guitablabel=%N\ %f

An example that resembles the default 'guitablabel': Show the number of
windows in the tab page and a '+' if there is a modified buffer: >

        function GuiTabLabel()
          let label = ''
          let bufnrlist = tabpagebuflist(v:lnum)

          " Add '+' if one of the buffers in the tab page is modified
          for bufnr in bufnrlist
            if getbufvar(bufnr, "&modified")
              let label = '+'
              break
            endif
          endfor

          " Append the number of windows in the tab page if more than one
          let wincount = tabpagewinnr(v:lnum, '$')
          if wincount > 1
            let label .= wincount
          endif
          if label != ''
            let label .= ' '
          endif

          " Append the buffer name
          return label . bufname(bufnrlist[tabpagewinnr(v:lnum) - 1])
        endfunction

        set guitablabel=%{GuiTabLabel()}

Note that the function must be defined before setting the option, otherwise
you get an error message for the function not being known.

If you want to fall back to the default label, return an empty string.

If you want to show something specific for a tab page, you might want to use a
tab page local variable. |t:var|


 vim:tw=78:ts=8:ft=help:norl:
*syntax.txt*    For Vim version 8.0.  Last change: 2017 Aug 12


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Syntax highlighting             *syntax* *syntax-highlighting* *coloring*

Syntax highlighting enables Vim to show parts of the text in another font or
color.  Those parts can be specific keywords or text matching a pattern.  Vim
doesn't parse the whole file (to keep it fast), so the highlighting has its
limitations.  Lexical highlighting might be a better name, but since everybody
```

```
calls it syntax highlighting we'll stick with that.

Vim supports syntax highlighting on all terminals.  But since most ordinary
terminals have very limited highlighting possibilities, it works best in the
GUI version, gvim.

In the User Manual:
|usr_06.txt| introduces syntax highlighting.
|usr_44.txt| introduces writing a syntax file.

1.  Quick start                 |:syn-qstart|
2.  Syntax files                |:syn-files|
3.  Syntax loading procedure    |syntax-loading|
4.  Syntax file remarks         |:syn-file-remarks|
5.  Defining a syntax           |:syn-define|
6.  :syntax arguments           |:syn-arguments|
7.  Syntax patterns             |:syn-pattern|
8.  Syntax clusters             |:syn-cluster|
9.  Including syntax files      |:syn-include|
10. Synchronizing               |:syn-sync|
11. Listing syntax items        |:syntax|
12. Highlight command           |:highlight|
13. Linking groups              |:highlight-link|
14. Cleaning up                 |:syn-clear|
15. Highlighting tags           |tag-highlight|
16. Window-local syntax         |:ownsyntax|
17. Color xterms                |xterm-color|
18. When syntax is slow         |:syntime|

{Vi does not have any of these commands}

Syntax highlighting is not available when the |+syntax| feature has been
disabled at compile time.

==============================================================================
1. Quick start                                          *:syn-qstart*

                                                *:syn-enable* *:syntax-enable*
This command switches on syntax highlighting: >

        :syntax enable

What this command actually does is to execute the command >
        :source $VIMRUNTIME/syntax/syntax.vim

If the VIM environment variable is not set, Vim will try to find
the path in another way (see |$VIMRUNTIME|).  Usually this works just
fine.  If it doesn't, try setting the VIM environment variable to the
directory where the Vim stuff is located.  For example, if your syntax files
are in the "/usr/vim/vim50/syntax" directory, set $VIMRUNTIME to
"/usr/vim/vim50".  You must do this in the shell, before starting Vim.

                                                        *:syn-on* *:syntax-on*
The ":syntax enable" command will keep your current color settings.  This
allows using ":highlight" commands to set your preferred colors before or
after using this command.  If you want Vim to overrule your settings with the
defaults, use: >
        :syntax on
<
                                            *:hi-normal* *:highlight-normal*
If you are running in the GUI, you can get white text on a black background
with: >
```

```
        :highlight Normal guibg=Black guifg=White
For a color terminal see |:hi-normal-cterm|.
For setting up your own colors syntax highlighting see |syncolor|.

NOTE: The syntax files on MS-DOS and Windows have lines that end in <CR><NL>.
The files for Unix end in <NL>.  This means you should use the right type of
file for your system.  Although on MS-DOS and Windows the right format is
automatically selected if the 'fileformats' option is not empty.

NOTE: When using reverse video ("gvim -fg white -bg black"), the default value
of 'background' will not be set until the GUI window is opened, which is after
reading the |gvimrc|.  This will cause the wrong default highlighting to be
used.  To set the default value of 'background' before switching on
highlighting, include the ":gui" command in the |gvimrc|: >

   :gui         " open window and set default for 'background'
   :syntax on   " start highlighting, use 'background' to set colors

NOTE: Using ":gui" in the |gvimrc| means that "gvim -f" won't start in the
foreground!  Use ":gui -f" then.

                                                        *g:syntax_on*
You can toggle the syntax on/off with this command: >
   :if exists("g:syntax_on") | syntax off | else | syntax enable | endif

To put this into a mapping, you can use: >
   :map <F7> :if exists("g:syntax_on") <Bar>
        \   syntax off <Bar>
        \ else <Bar>
        \   syntax enable <Bar>
        \ endif <CR>
[using the |<>| notation, type this literally]

Details:
The ":syntax" commands are implemented by sourcing a file.  To see exactly how
this works, look in the file:
    command             file ~
    :syntax enable      $VIMRUNTIME/syntax/syntax.vim
    :syntax on          $VIMRUNTIME/syntax/syntax.vim
    :syntax manual      $VIMRUNTIME/syntax/manual.vim
    :syntax off         $VIMRUNTIME/syntax/nosyntax.vim
Also see |syntax-loading|.

NOTE: If displaying long lines is slow and switching off syntax highlighting
makes it fast, consider setting the 'synmaxcol' option to a lower value.

==============================================================================
2. Syntax files                                         *:syn-files*

The syntax and highlighting commands for one language are normally stored in
a syntax file.  The name convention is: "{name}.vim".  Where {name} is the
name of the language, or an abbreviation (to fit the name in 8.3 characters,
a requirement in case the file is used on a DOS filesystem).
Examples:
        c.vim           perl.vim        java.vim        html.vim
        cpp.vim         sh.vim          csh.vim

The syntax file can contain any Ex commands, just like a vimrc file.  But
the idea is that only commands for a specific language are included.  When a
language is a superset of another language, it may include the other one,
for example, the cpp.vim file could include the c.vim file: >
   :so $VIMRUNTIME/syntax/c.vim
```

```
The .vim files are normally loaded with an autocommand.  For example: >
   :au Syntax c     runtime! syntax/c.vim
   :au Syntax cpp   runtime! syntax/cpp.vim
These commands are normally in the file $VIMRUNTIME/syntax/synload.vim.


MAKING YOUR OWN SYNTAX FILES                              *mysyntaxfile*

When you create your own syntax files, and you want to have Vim use these
automatically with ":syntax enable", do this:

1. Create your user runtime directory.  You would normally use the first item
   of the 'runtimepath' option.  Example for Unix: >
        mkdir ~/.vim

2. Create a directory in there called "syntax".  For Unix: >
        mkdir ~/.vim/syntax

3. Write the Vim syntax file.  Or download one from the internet.  Then write
   it in your syntax directory.  For example, for the "mine" syntax: >
        :w ~/.vim/syntax/mine.vim

Now you can start using your syntax file manually: >
        :set syntax=mine
You don't have to exit Vim to use this.

If you also want Vim to detect the type of file, see |new-filetype|.

If you are setting up a system with many users and you don't want each user
to add the same syntax file, you can use another directory from 'runtimepath'.


ADDING TO AN EXISTING SYNTAX FILE               *mysyntaxfile-add*

If you are mostly satisfied with an existing syntax file, but would like to
add a few items or change the highlighting, follow these steps:

1. Create your user directory from 'runtimepath', see above.

2. Create a directory in there called "after/syntax".  For Unix: >
        mkdir ~/.vim/after
        mkdir ~/.vim/after/syntax

3. Write a Vim script that contains the commands you want to use.  For
   example, to change the colors for the C syntax: >
        highlight cComment ctermfg=Green guifg=Green

4. Write that file in the "after/syntax" directory.  Use the name of the
   syntax, with ".vim" added.  For our C syntax: >
        :w ~/.vim/after/syntax/c.vim

That's it.  The next time you edit a C file the Comment color will be
different.  You don't even have to restart Vim.

If you have multiple files, you can use the filetype as the directory name.
All the "*.vim" files in this directory will be used, for example:
        ~/.vim/after/syntax/c/one.vim
        ~/.vim/after/syntax/c/two.vim


REPLACING AN EXISTING SYNTAX FILE                       *mysyntaxfile-replace*
```

If you don't like a distributed syntax file, or you have downloaded a new
version, follow the same steps as for |mysyntaxfile| above. Just make sure
that you write the syntax file in a directory that is early in 'runtimepath'.
Vim will only load the first syntax file found, assuming that it sets
b:current_syntax.

NAMING CONVENTIONS *group-name* *{group-name}* *E669* *W18*

A syntax group name is to be used for syntax items that match the same kind of
thing. These are then linked to a highlight group that specifies the color.
A syntax group name doesn't specify any color or attributes itself.

The name for a highlight or syntax group must consist of ASCII letters, digits
and the underscore. As a regexp: "[a-zA-Z0-9_]*". However, Vim does not give
an error when using other characters.

To be able to allow each user to pick his favorite set of colors, there must
be preferred names for highlight groups that are common for many languages.
These are the suggested group names (if syntax highlighting works properly
you can see the actual color, except for "Ignore"):

```
        *Comment        any comment

        *Constant       any constant
         String         a string constant: "this is a string"
         Character      a character constant: 'c', '\n'
         Number         a number constant: 234, 0xff
         Boolean        a boolean constant: TRUE, false
         Float          a floating point constant: 2.3e10

        *Identifier     any variable name
         Function       function name (also: methods for classes)

        *Statement      any statement
         Conditional    if, then, else, endif, switch, etc.
         Repeat         for, do, while, etc.
         Label          case, default, etc.
         Operator       "sizeof", "+", "*", etc.
         Keyword        any other keyword
         Exception      try, catch, throw

        *PreProc        generic Preprocessor
         Include        preprocessor #include
         Define         preprocessor #define
         Macro          same as Define
         PreCondit      preprocessor #if, #else, #endif, etc.

        *Type           int, long, char, etc.
         StorageClass   static, register, volatile, etc.
         Structure      struct, union, enum, etc.
         Typedef        A typedef

        *Special        any special symbol
         SpecialChar    special character in a constant
         Tag            you can use CTRL-] on this
         Delimiter      character that needs attention
         SpecialComment special things inside a comment
         Debug          debugging statements

        *Underlined     text that stands out, HTML links
```

```
        *Ignore         left blank, hidden  |hl-Ignore|

        *Error          any erroneous construct

        *Todo           anything that needs extra attention; mostly the
                        keywords TODO FIXME and XXX

The names marked with * are the preferred groups; the others are minor groups.
For the preferred groups, the "syntax.vim" file contains default highlighting.
The minor groups are linked to the preferred groups, so they get the same
highlighting.  You can override these defaults by using ":highlight" commands
after sourcing the "syntax.vim" file.

Note that highlight group names are not case sensitive.  "String" and "string"
can be used for the same group.

The following names are reserved and cannot be used as a group name:
        NONE   ALL   ALLBUT   contains   contained

                                                        *hl-Ignore*
When using the Ignore group, you may also consider using the conceal
mechanism.  See |conceal|.

==============================================================================
3. Syntax loading procedure                             *syntax-loading*

This explains the details that happen when the command ":syntax enable" is
issued.  When Vim initializes itself, it finds out where the runtime files are
located.  This is used here as the variable |$VIMRUNTIME|.

":syntax enable" and ":syntax on" do the following:

    Source $VIMRUNTIME/syntax/syntax.vim
    |
    +-  Clear out any old syntax by sourcing $VIMRUNTIME/syntax/nosyntax.vim
    |
    +-  Source first syntax/synload.vim in 'runtimepath'
    |   |
    |   +-  Setup the colors for syntax highlighting.  If a color scheme is
    |   |   defined it is loaded again with ":colors {name}".  Otherwise
    |   |   ":runtime! syntax/syncolor.vim" is used.  ":syntax on" overrules
    |   |   existing colors, ":syntax enable" only sets groups that weren't
    |   |   set yet.
    |   |
    |   +-  Set up syntax autocmds to load the appropriate syntax file when
    |   |   the 'syntax' option is set. *synload-1*
    |   |
    |   +-  Source the user's optional file, from the |mysyntaxfile| variable.
    |       This is for backwards compatibility with Vim 5.x only. *synload-2*
    |
    +-  Do ":filetype on", which does ":runtime! filetype.vim".  It loads any
    |   filetype.vim files found.  It should always Source
    |   $VIMRUNTIME/filetype.vim, which does the following.
    |   |
    |   +-  Install autocmds based on suffix to set the 'filetype' option
    |   |   This is where the connection between file name and file type is
    |   |   made for known file types. *synload-3*
    |   |
    |   +-  Source the user's optional file, from the *myfiletypefile*
    |   |   variable.  This is for backwards compatibility with Vim 5.x only.
    |   |   *synload-4*
```

```
    |   |
    |   +-  Install one autocommand which sources scripts.vim when no file
    |   |   type was detected yet. *synload-5*
    |   |
    |   +-  Source $VIMRUNTIME/menu.vim, to setup the Syntax menu. |menu.vim|
    |
    +-  Install a FileType autocommand to set the 'syntax' option when a file
    |   type has been detected. *synload-6*
    |
    +-  Execute syntax autocommands to start syntax highlighting for each
        already loaded buffer.


Upon loading a file, Vim finds the relevant syntax file as follows:

    Loading the file triggers the BufReadPost autocommands.
    |
    +-  If there is a match with one of the autocommands from |synload-3|
    |   (known file types) or |synload-4| (user's file types), the 'filetype'
    |   option is set to the file type.
    |
    +-  The autocommand at |synload-5| is triggered.  If the file type was not
    |   found yet, then scripts.vim is searched for in 'runtimepath'.  This
    |   should always load $VIMRUNTIME/scripts.vim, which does the following.
    |   |
    |   +-  Source the user's optional file, from the *myscriptsfile*
    |   |   variable.  This is for backwards compatibility with Vim 5.x only.
    |   |
    |   +-  If the file type is still unknown, check the contents of the file,
    |       again with checks like "getline(1) =~ pattern" as to whether the
    |       file type can be recognized, and set 'filetype'.
    |
    +-  When the file type was determined and 'filetype' was set, this
    |   triggers the FileType autocommand |synload-6| above.  It sets
    |   'syntax' to the determined file type.
    |
    +-  When the 'syntax' option was set above, this triggers an autocommand
    |   from |synload-1| (and |synload-2|).  This find the main syntax file in
    |   'runtimepath', with this command:
    |           runtime! syntax/<name>.vim
    |
    +-  Any other user installed FileType or Syntax autocommands are
        triggered.  This can be used to change the highlighting for a specific
        syntax.

==============================================================================
4. Syntax file remarks                                  *:syn-file-remarks*

                                                    *b:current_syntax-variable*
Vim stores the name of the syntax that has been loaded in the
"b:current_syntax" variable.  You can use this if you want to load other
settings, depending on which syntax is active.  Example: >
   :au BufReadPost * if b:current_syntax == "csh"
   :au BufReadPost *   do-some-things
   :au BufReadPost * endif


2HTML                                               *2html.vim* *convert-to-HTML*

This is not a syntax file itself, but a script that converts the current
window into HTML. Vim opens a new window in which it builds the HTML file.
```

```
After you save the resulting file, you can view it with any browser. The
colors should be exactly the same as you see them in Vim.  With
|g:html_line_ids| you can jump to specific lines by adding (for example) #L123
or #123 to the end of the URL in your browser's address bar. And with
|g:html_dynamic_folds| enabled, you can show or hide the text that is folded
in Vim.

You are not supposed to set the 'filetype' or 'syntax' option to "2html"!
Source the script to convert the current file: >

        :runtime! syntax/2html.vim
<
Many variables affect the output of 2html.vim; see below. Any of the on/off
options listed below can be enabled or disabled by setting them explicitly to
the desired value, or restored to their default by removing the variable using
|:unlet|.

Remarks:
- Some truly ancient browsers may not show the background colors.
- From most browsers you can also print the file (in color)!
- The latest TOhtml may actually work with older versions of Vim, but some
  features such as conceal support will not function, and the colors may be
  incorrect for an old Vim without GUI support compiled in.

Here is an example how to run the script over all .c and .h files from a
Unix shell: >
   for f in *.[ch]; do gvim -f +"syn on" +"run! syntax/2html.vim" +"wq" +"q" $f; done
<
                                          *g:html_start_line* *g:html_end_line*
To restrict the conversion to a range of lines, use a range with the |:TOhtml|
command below, or set "g:html_start_line" and "g:html_end_line" to the first
and last line to be converted.  Example, using the last set Visual area: >

        :let g:html_start_line = line("'<")
        :let g:html_end_line = line("'>")
        :runtime! syntax/2html.vim
<
                                                              *:TOhtml*
:[range]TOhtml          The ":TOhtml" command is defined in a standard plugin.
                        This command will source |2html.vim| for you. When a
                        range is given, this command sets |g:html_start_line|
                        and |g:html_end_line| to the start and end of the
                        range, respectively. Default range is the entire
                        buffer.

                        If the current window is part of a |diff|, unless
                        |g:html_diff_one_file| is set, :TOhtml will convert
                        all windows which are part of the diff in the current
                        tab and place them side-by-side in a <table> element
                        in the generated HTML. With |g:html_line_ids| you can
                        jump to lines in specific windows with (for example)
                        #W1L42 for line 42 in the first diffed window, or
                        #W3L87 for line 87 in the third.

                        Examples: >

        :10,40TOhtml " convert lines 10-40 to html
        :'<,'>TOhtml " convert current/last visual selection
        :TOhtml      " convert entire buffer
<
                                                              *g:html_diff_one_file*
Default: 0.
```

When 0, and using |:TOhtml| all windows involved in a |diff| in the current tab page are converted to HTML and placed side-by-side in a <table> element. When 1, only the current buffer is converted.
Example: >

        let g:html_diff_one_file = 1
<
g:html_whole_filler
Default: 0.
When 0, if |g:html_diff_one_file| is 1, a sequence of more than 3 filler lines is displayed as three lines with the middle line mentioning the total number of inserted lines.
When 1, always display all inserted lines as if |g:html_diff_one_file| were not set.
>
    :let g:html_whole_filler = 1
<
TOhtml-performance *g:html_no_progress*
Default: 0.
When 0, display a progress bar in the statusline for each major step in the 2html.vim conversion process.
When 1, do not display the progress bar. This offers a minor speed improvement but you won't have any idea how much longer the conversion might take; for big files it can take a long time!
Example: >

        let g:html_no_progress = 1
<
You can obtain better performance improvements by also instructing Vim to not run interactively, so that too much time is not taken to redraw as the script moves through the buffer, switches windows, and the like: >

  vim -E -s -c "let g:html_no_progress=1" -c "syntax on" -c "set ft=c" -c "runtime syntax/2html.vim" -cwqa myfile.c
<
Note that the -s flag prevents loading your .vimrc and any plugins, so you need to explicitly source/enable anything that will affect the HTML conversion. See |-E| and |-s-ex| for details. It is probably best to create a script to replace all the -c commands and use it with the -u flag instead of specifying each command separately.

g:html_number_lines
Default: current 'number' setting.
When 0, buffer text is displayed in the generated HTML without line numbering.
When 1, a column of line numbers is added to the generated HTML with the same highlighting as the line number column in Vim (|hl-LineNr|).
Force line numbers even if 'number' is not set: >
   :let g:html_number_lines = 1
Force to omit the line numbers: >
   :let g:html_number_lines = 0
Go back to the default to use 'number' by deleting the variable: >
   :unlet g:html_number_lines
<
g:html_line_ids
Default: 1 if |g:html_number_lines| is set, 0 otherwise.
When 1, adds an HTML id attribute to each line number, or to an empty inserted for that purpose if no line numbers are shown. This ID attribute takes the form of L123 for single-buffer HTML pages, or W2L123 for diff-view pages, and is used to jump to a specific line (in a specific window of a diff view). Javascript is inserted to open any closed dynamic folds (|g:html_dynamic_folds|) containing the specified line before jumping. The javascript also allows omitting the window ID in the url, and the leading L.

```
For example: >

        page.html#L123  jumps to line 123 in a single-buffer file
        page.html#123   does the same

        diff.html#W1L42 jumps to line 42 in the first window in a diff
        diff.html#42    does the same
<
                                                                *g:html_use_css*
Default: 1.
When 1, generate valid HTML 4.01 markup with CSS1 styling, supported in all
modern browsers and most old browsers.
When 0, generate <font> tags and similar outdated markup. This is not
recommended but it may work better in really old browsers, email clients,
forum posts, and similar situations where basic CSS support is unavailable.
Example: >
   :let g:html_use_css = 0
<
                                                        *g:html_ignore_conceal*
Default: 0.
When 0, concealed text is removed from the HTML and replaced with a character
from |:syn-cchar| or 'listchars' as appropriate, depending on the current
value of 'conceallevel'.
When 1, include all text from the buffer in the generated HTML, even if it is
|conceal|ed.

Either of the following commands will ensure that all text in the buffer is
included in the generated HTML (unless it is folded): >
   :let g:html_ignore_conceal = 1
   :setl conceallevel=0
<
                                                        *g:html_ignore_folding*
Default: 0.
When 0, text in a closed fold is replaced by the text shown for the fold in
Vim (|fold-foldtext|). See |g:html_dynamic_folds| if you also want to allow
the user to expand the fold as in Vim to see the text inside.
When 1, include all text from the buffer in the generated HTML; whether the
text is in a fold has no impact at all. |g:html_dynamic_folds| has no effect.

Either of these commands will ensure that all text in the buffer is included
in the generated HTML (unless it is concealed): >
   zR
   :let g:html_ignore_folding = 1
<
                                                         *g:html_dynamic_folds*
Default: 0.
When 0, text in a closed fold is not included at all in the generated HTML.
When 1, generate javascript to open a fold and show the text within, just like
in Vim.

Setting this variable to 1 causes 2html.vim to always use CSS for styling,
regardless of what |g:html_use_css| is set to.

This variable is ignored when |g:html_ignore_folding| is set.
>
   :let g:html_dynamic_folds = 1
<
                                                        *g:html_no_foldcolumn*
Default: 0.
When 0, if |g:html_dynamic_folds| is 1, generate a column of text similar to
Vim's foldcolumn (|fold-foldcolumn|) the user can click on to toggle folds
open or closed. The minimum width of the generated text column is the current
```

```
'foldcolumn' setting.
When 1, do not generate this column; instead, hovering the mouse cursor over
folded text will open the fold as if |g:html_hover_unfold| were set.
>
   :let g:html_no_foldcolumn = 1
<
                                  *TOhtml-uncopyable-text* *g:html_prevent_copy*
Default: empty string.
This option prevents certain regions of the generated HTML from being copied,
when you select all text in document rendered in a browser and copy it. Useful
for allowing users to copy-paste only the source text even if a fold column or
line numbers are shown in the generated content. Specify regions to be
affected in this way as follows:
        f:      fold column
        n:      line numbers (also within fold text)
        t:      fold text
        d:      diff filler

Example, to make the fold column and line numbers uncopyable: >
        :let g:html_prevent_copy = "fn"
<
This feature is currently implemented by inserting read-only <input> elements
into the markup to contain the uncopyable areas. This does not work well in
all cases. When pasting to some applications which understand HTML, the
<input> elements also get pasted. But plain-text paste destinations should
always work.

                                                                *g:html_no_invalid*
Default: 0.
When 0, if |g:html_prevent_copy| is non-empty, an invalid attribute is
intentionally inserted into the <input> element for the uncopyable areas. This
increases the number of applications you can paste to without also pasting the
<input> elements. Specifically, Microsoft Word will not paste the <input>
elements if they contain this invalid attribute.
When 1, no invalid markup is ever intentionally inserted, and the generated
page should validate. However, be careful pasting into Microsoft Word when
|g:html_prevent_copy| is non-empty; it can be hard to get rid of the <input>
elements which get pasted.

                                                              *g:html_hover_unfold*
Default: 0.
When 0, the only way to open a fold generated by 2html.vim with
|g:html_dynamic_folds| set, is to click on the generated fold column.
When 1, use CSS 2.0 to allow the user to open a fold by moving the mouse
cursor over the displayed fold text. This is useful to allow users with
disabled javascript to view the folded text.

Note that old browsers (notably Internet Explorer 6) will not support this
feature.  Browser-specific markup for IE6 is included to fall back to the
normal CSS1 styling so that the folds show up correctly for this browser, but
they will not be openable without a foldcolumn.
>
   :let g:html_hover_unfold = 1
<
                                                                  *g:html_id_expr*
Default: ""
Dynamic folding and jumping to line IDs rely on unique IDs within the document
to work. If generated HTML is copied into a larger document, these IDs are no
longer guaranteed to be unique. Set g:html_id_expr to an expression Vim can
evaluate to get a unique string to append to each ID used in a given document,
so that the full IDs will be unique even when combined with other content in a
larger HTML document. Example, to append _ and the buffer number to each ID: >
```

```
        :let g:html_id_expr = '"_".bufnr("%")'
<
To append a string "_mystring" to the end of each ID: >

        :let g:html_id_expr = '"_mystring"'
<
Note, when converting a diff view to HTML, the expression will only be
evaluated for the first window in the diff, and the result used for all the
windows.

                                              *TOhtml-wrap-text* *g:html_pre_wrap*
Default: current 'wrap' setting.
When 0, if |g:html_no_pre| is 0 or unset, the text in the generated HTML does
not wrap at the edge of the browser window.
When 1, if |g:html_use_css| is 1, the CSS 2.0 "white-space:pre-wrap" value is
used, causing the text to wrap at whitespace at the edge of the browser
window.
Explicitly enable text wrapping: >
   :let g:html_pre_wrap = 1
Explicitly disable wrapping: >
   :let g:html_pre_wrap = 0
Go back to default, determine wrapping from 'wrap' setting: >
   :unset g:html_pre_wrap
<
                                                                  *g:html_no_pre*
Default: 0.
When 0, buffer text in the generated HTML is surrounded by <pre>...</pre>
tags. Series of whitespace is shown as in Vim without special markup, and tab
characters can be included literally (see |g:html_expand_tabs|).
When 1 (not recommended), the <pre> tags are omitted, and a plain <div> is
used instead. Whitespace is replaced by a series of   character
references, and <br> is used to end each line. This is another way to allow
text in the generated HTML is wrap (see |g:html_pre_wrap|) which also works in
old browsers, but may cause noticeable differences between Vim's display and
the rendered page generated by 2html.vim.
>
   :let g:html_no_pre = 1
<
                                                          *g:html_expand_tabs*
Default: 1 if 'tabstop' is 8, 'expandtab' is 0, and no fold column or line
                numbers occur in the generated HTML;
         0 otherwise.
When 0, <Tab> characters in the buffer text are replaced with an appropriate
number of space characters, or   references if |g:html_no_pre| is 1.
When 1, if |g:html_no_pre| is 0 or unset, <Tab> characters in the buffer text
are included as-is in the generated HTML. This is useful for when you want to
allow copy and paste from a browser without losing the actual whitespace in
the source document. Note that this can easily break text alignment and
indentation in the HTML, unless set by default.

Force |2html.vim| to keep <Tab> characters: >
   :let g:html_expand_tabs = 0
<
Force tabs to be expanded: >
   :let g:html_expand_tabs = 1
<
                                     *TOhtml-encoding-detect* *TOhtml-encoding*
It is highly recommended to set your desired encoding with
|g:html_use_encoding| for any content which will be placed on a web server.

If you do not specify an encoding, |2html.vim| uses the preferred IANA name
```

```
for the current value of 'fileencoding' if set, or 'encoding' if not.
'encoding' is always used for certain 'buftype' values. 'fileencoding' will be
set to match the chosen document encoding.

Automatic detection works for the encodings mentioned specifically by name in
|encoding-names|, but TOhtml will only automatically use those encodings with
wide browser support. However, you can override this to support specific
encodings that may not be automatically detected by default (see options
below). See http://www.iana.org/assignments/character-sets for the IANA names.

Note, by default all Unicode encodings are converted to UTF-8 with no BOM in
the generated HTML, as recommended by W3C:

        http://www.w3.org/International/questions/qa-choosing-encodings
        http://www.w3.org/International/questions/qa-byte-order-mark

                                                          *g:html_use_encoding*
Default: none, uses IANA name for current 'fileencoding' as above.
To overrule all automatic charset detection, set g:html_use_encoding to the
name of the charset to be used. It is recommended to set this variable to
something widely supported, like UTF-8, for anything you will be hosting on a
webserver: >
   :let g:html_use_encoding = "UTF-8"
You can also use this option to omit the line that specifies the charset
entirely, by setting g:html_use_encoding to an empty string (NOT recommended): >
   :let g:html_use_encoding = ""
To go back to the automatic mechanism, delete the |g:html_use_encoding|
variable: >
   :unlet g:html_use_encoding
<
                                                      *g:html_encoding_override*
Default: none, autoload/tohtml.vim contains default conversions for encodings
                mentioned by name at |encoding-names|.
This option allows |2html.vim| to detect the correct 'fileencoding' when you
specify an encoding with |g:html_use_encoding| which is not in the default
list of conversions.

This is a dictionary of charset-encoding pairs that will replace existing
pairs automatically detected by TOhtml, or supplement with new pairs.

Detect the HTML charset "windows-1252" as the encoding "8bit-cp1252": >
   :let g:html_encoding_override = {'windows-1252': '8bit-cp1252'}
<
                                                       *g:html_charset_override*
Default: none, autoload/tohtml.vim contains default conversions for encodings
                mentioned by name at |encoding-names| and which have wide
                browser support.
This option allows |2html.vim| to detect the HTML charset for any
'fileencoding' or 'encoding' which is not detected automatically. You can also
use it to override specific existing encoding-charset pairs. For example,
TOhtml will by default use UTF-8 for all Unicode/UCS encodings. To use UTF-16
and UTF-32 instead, use: >
   :let g:html_charset_override = {'ucs-4': 'UTF-32', 'utf-16': 'UTF-16'}

Note that documents encoded in either UTF-32 or UTF-16 have known
compatibility problems with some major browsers.

                                                                  *g:html_font*
Default: "monospace"
You can specify the font or fonts used in the converted document using
g:html_font. If this option is set to a string, then the value will be
surrounded with single quotes. If this option is set to a list then each list
```

```
item is surrounded by single quotes and the list is joined with commas. Either
way, "monospace" is added as the fallback generic family name and the entire
result used as the font family (using CSS) or font face (if not using CSS).
Examples: >

   " font-family: 'Consolas', monospace;
   :let g:html_font = "Consolas"

   " font-family: 'DejaVu Sans Mono', 'Consolas', monospace;
   :let g:html_font = ["DejaVu Sans Mono", "Consolas"]
<
                         *convert-to-XML* *convert-to-XHTML* *g:html_use_xhtml*
Default: 0.
When 0, generate standard HTML 4.01 (strict when possible).
When 1, generate XHTML 1.0 instead (XML compliant HTML).
>
    :let g:html_use_xhtml = 1
<

ABEL                                                *abel.vim* *ft-abel-syntax*

ABEL highlighting provides some user-defined options.  To enable them, assign
any value to the respective variable.  Example: >
        :let abel_obsolete_ok=1
To disable them use ":unlet".  Example: >
        :unlet abel_obsolete_ok

Variable                        Highlight ~
abel_obsolete_ok                obsolete keywords are statements, not errors
abel_cpp_comments_illegal       do not interpret '//' as inline comment leader


ADA

See |ft-ada-syntax|


ANT                                                 *ant.vim* *ft-ant-syntax*

The ant syntax file provides syntax highlighting for javascript and python
by default.  Syntax highlighting for other script languages can be installed
by the function AntSyntaxScript(), which takes the tag name as first argument
and the script syntax file name as second argument.  Example: >

        :call AntSyntaxScript('perl', 'perl.vim')

will install syntax perl highlighting for the following ant code >

        <script language = 'perl'><![CDATA[
            # everything inside is highlighted as perl
        ]]></script>

See |mysyntaxfile-add| for installing script languages permanently.


APACHE                                              *apache.vim* *ft-apache-syntax*

The apache syntax file provides syntax highlighting depending on Apache HTTP
server version, by default for 1.3.x.  Set "apache_version" to Apache version
(as a string) to get highlighting for another version.  Example: >

        :let apache_version = "2.0"
```

<

asm.vim *asmh8300.vim* *nasm.vim* *masm.vim* *asm68k*
ASSEMBLY *ft-asm-syntax* *ft-asmh8300-syntax* *ft-nasm-syntax*
ft-masm-syntax *ft-asm68k-syntax* *fasm.vim*

Files matching "*.i" could be Progress or Assembly. If the automatic detection doesn't work for you, or you don't edit Progress at all, use this in your startup vimrc: >
:let filetype_i = "asm"
Replace "asm" with the type of assembly you use.

There are many types of assembly languages that all use the same file name extensions. Therefore you will have to select the type yourself, or add a line in the assembly file that Vim will recognize. Currently these syntax files are included:

asm	GNU assembly (the default)
asm68k	Motorola 680x0 assembly
asmh8300	Hitachi H-8300 version of GNU assembly
ia64	Intel Itanium 64
fasm	Flat assembly (http://flatassembler.net)
masm	Microsoft assembly (probably works for any 80x86)
nasm	Netwide assembly
tasm	Turbo Assembly (with opcodes 80x86 up to Pentium, and MMX)
pic	PIC assembly (currently for PIC16F84)

The most flexible is to add a line in your assembly file containing: >
asmsyntax=nasm
Replace "nasm" with the name of the real assembly syntax. This line must be one of the first five lines in the file. No non-white text must be immediately before or after this text. Note that specifying asmsyntax=foo is equivalent to setting ft=foo in a |modeline|, and that in case of a conflict between the two settings the one from the modeline will take precedence (in particular, if you have ft=asm in the modeline, you will get the GNU syntax highlighting regardless of what is specified as asmsyntax).

The syntax type can always be overruled for a specific buffer by setting the b:asmsyntax variable: >
:let b:asmsyntax = "nasm"

If b:asmsyntax is not set, either automatically or by hand, then the value of the global variable asmsyntax is used. This can be seen as a default assembly language: >
:let asmsyntax = "nasm"

As a last resort, if nothing is defined, the "asm" syntax is used.

Netwide assembler (nasm.vim) optional highlighting ~

To enable a feature: >
:let {variable}=1|set syntax=nasm
To disable a feature: >
:unlet {variable} |set syntax=nasm

Variable	Highlight ~
nasm_loose_syntax	unofficial parser allowed syntax not as Error (parser dependent; not recommended)
nasm_ctx_outside_macro	contexts outside macro not as Error
nasm_no_warn	potentially risky syntax not as ToDo

ASPPERL and ASPVBS *ft-aspperl-syntax* *ft-aspvbs-syntax*

*.asp and *.asa files could be either Perl or Visual Basic script. Since it's hard to detect this you can set two global variables to tell Vim what you are using. For Perl script use: >
:let g:filetype_asa = "aspperl"
:let g:filetype_asp = "aspperl"
For Visual Basic use: >
:let g:filetype_asa = "aspvbs"
:let g:filetype_asp = "aspvbs"

BAAN *baan.vim* *baan-syntax*

The baan.vim gives syntax support for BaanC of release BaanIV upto SSA ERP LN for both 3 GL and 4 GL programming. Large number of standard defines/constants are supported.

Some special violation of coding standards will be signalled when one specify in ones |.vimrc|: >
let baan_code_stds=1

baan-folding

Syntax folding can be enabled at various levels through the variables mentioned below (Set those in your |.vimrc|). The more complex folding on source blocks and SQL can be CPU intensive.

To allow any folding and enable folding at function level use: >
let baan_fold=1
Folding can be enabled at source block level as if, while, for ,... The indentation preceding the begin/end keywords has to match (spaces are not considered equal to a tab). >
let baan_fold_block=1
Folding can be enabled for embedded SQL blocks as SELECT, SELECTDO, SELECTEMPTY, ... The indentation preceding the begin/end keywords has to match (spaces are not considered equal to a tab). >
let baan_fold_sql=1
Note: Block folding can result in many small folds. It is suggested to |:set| the options 'foldminlines' and 'foldnestmax' in |.vimrc| or use |:setlocal| in .../after/syntax/baan.vim (see |after-directory|). Eg: >
set foldminlines=5
set foldnestmax=6

BASIC *basic.vim* *vb.vim* *ft-basic-syntax* *ft-vb-syntax*

Both Visual Basic and "normal" basic use the extension ".bas". To detect which one should be used, Vim checks for the string "VB_Name" in the first five lines of the file. If it is not found, filetype will be "basic", otherwise "vb". Files with the ".frm" extension will always be seen as Visual Basic.

C *c.vim* *ft-c-syntax*

A few things in C highlighting are optional. To enable them assign any value to the respective variable. Example: >
:let c_comment_strings = 1
To disable them use ":unlet". Example: >
:unlet c_comment_strings

```
Variable                Highlight ~
*c_gnu*                 GNU gcc specific items
*c_comment_strings*     strings and numbers inside a comment
*c_space_errors*                trailing white space and spaces before a <Tab>
*c_no_trail_space_error*         ... but no trailing spaces
*c_no_tab_space_error*   ... but no spaces before a <Tab>
*c_no_bracket_error*    don't highlight {}; inside [] as errors
*c_no_curly_error*      don't highlight {}; inside [] and () as errors;
                                except { and } in first column
                                Default is to highlight them, otherwise you
                                can't spot a missing ")".
*c_curly_error*         highlight a missing }; this forces syncing from the
                        start of the file, can be slow
*c_no_ansi*             don't do standard ANSI types and constants
*c_ansi_typedefs*                ... but do standard ANSI types
*c_ansi_constants*       ... but do standard ANSI constants
*c_no_utf*              don't highlight \u and \U in strings
*c_syntax_for_h*                for *.h files use C syntax instead of C++ and use objc
                        syntax instead of objcpp
*c_no_if0*              don't highlight "#if 0" blocks as comments
*c_no_cformat*          don't highlight %-formats in strings
*c_no_c99*              don't highlight C99 standard items
*c_no_c11*              don't highlight C11 standard items
*c_no_bsd*              don't highlight BSD specific types

When 'foldmethod' is set to "syntax" then /* */ comments and { } blocks will
become a fold.  If you don't want comments to become a fold use: >
        :let c_no_comment_fold = 1
"#if 0" blocks are also folded, unless: >
        :let c_no_if0_fold = 1

If you notice highlighting errors while scrolling backwards, which are fixed
when redrawing with CTRL-L, try setting the "c_minlines" internal variable
to a larger number: >
        :let c_minlines = 100
This will make the syntax synchronization start 100 lines before the first
displayed line.  The default value is 50 (15 when c_no_if0 is set).  The
disadvantage of using a larger number is that redrawing can become slow.

When using the "#if 0" / "#endif" comment highlighting, notice that this only
works when the "#if 0" is within "c_minlines" from the top of the window.  If
you have a long "#if 0" construct it will not be highlighted correctly.

To match extra items in comments, use the cCommentGroup cluster.
Example: >
   :au Syntax c call MyCadd()
   :function MyCadd()
   :  syn keyword cMyItem contained Ni
   :  syn cluster cCommentGroup add=cMyItem
   :  hi link cMyItem Title
   :endfun

ANSI constants will be highlighted with the "cConstant" group.  This includes
"NULL", "SIG_IGN" and others.  But not "TRUE", for example, because this is
not in the ANSI standard.  If you find this confusing, remove the cConstant
highlighting: >
        :hi link cConstant NONE

If you see '{' and '}' highlighted as an error where they are OK, reset the
highlighting for cErrInParen and cErrInBracket.
```

```
If you want to use folding in your C files, you can add these lines in a file
in the "after" directory in 'runtimepath'.  For Unix this would be
~/.vim/after/syntax/c.vim. >
    syn sync fromstart
    set foldmethod=syntax

CH                                                  *ch.vim* *ft-ch-syntax*

C/C++ interpreter.  Ch has similar syntax highlighting to C and builds upon
the C syntax file.  See |c.vim| for all the settings that are available for C.

By setting a variable you can tell Vim to use Ch syntax for *.h files, instead
of C or C++: >
        :let ch_syntax_for_h = 1


CHILL                                               *chill.vim* *ft-chill-syntax*

Chill syntax highlighting is similar to C.  See |c.vim| for all the settings
that are available.  Additionally there is:

chill_space_errors      like c_space_errors
chill_comment_string    like c_comment_strings
chill_minlines          like c_minlines


CHANGELOG                                 *changelog.vim* *ft-changelog-syntax*

ChangeLog supports highlighting spaces at the start of a line.
If you do not like this, add following line to your .vimrc: >
        let g:changelog_spacing_errors = 0
This works the next time you edit a changelog file.  You can also use
"b:changelog_spacing_errors" to set this per buffer (before loading the syntax
file).

You can change the highlighting used, e.g., to flag the spaces as an error: >
        :hi link ChangelogError Error
Or to avoid the highlighting: >
        :hi link ChangelogError NONE
This works immediately.


CLOJURE                                                   *ft-clojure-syntax*

The default syntax groups can be augmented through the
*g:clojure_syntax_keywords* and *b:clojure_syntax_keywords* variables. The
value should be a |Dictionary| of syntax group names to a |List| of custom
identifiers:
>
        let g:clojure_syntax_keywords = {
            \ 'clojureMacro': ["defproject", "defcustom"],
            \ 'clojureFunc': ["string/join", "string/replace"]
            \ }
<
Refer to the Clojure syntax script for valid syntax group names.

If the |buffer-variable| *b:clojure_syntax_without_core_keywords* is set, only
language constants and special forms are matched.

Setting *g:clojure_fold* enables folding Clojure code via the syntax engine.
Any list, vector, or map that extends over more than one line can be folded
using the standard Vim |fold-commands|.
```

```
Please note that this option does not work with scripts that redefine the
bracket syntax regions, such as rainbow-parentheses plugins.

This option is off by default.
>
        " Default
        let g:clojure_fold = 0
<

COBOL                                                   *cobol.vim* *ft-cobol-syntax*

COBOL highlighting has different needs for legacy code than it does for fresh
development.  This is due to differences in what is being done (maintenance
versus development) and other factors.  To enable legacy code highlighting,
add this line to your .vimrc: >
        :let cobol_legacy_code = 1
To disable it again, use this: >
        :unlet cobol_legacy_code


COLD FUSION                     *coldfusion.vim* *ft-coldfusion-syntax*

The ColdFusion has its own version of HTML comments.  To turn on ColdFusion
comment highlighting, add the following line to your startup file: >

        :let html_wrong_comments = 1

The ColdFusion syntax file is based on the HTML syntax file.


CPP                                                     *cpp.vim* *ft-cpp-syntax*

Most of things are same as |ft-c-syntax|.

Variable                Highlight ~
cpp_no_cpp11            don't highlight C++11 standard items
cpp_no_cpp14            don't highlight C++14 standard items


CSH                                                     *csh.vim* *ft-csh-syntax*

This covers the shell named "csh".  Note that on some systems tcsh is actually
used.

Detecting whether a file is csh or tcsh is notoriously hard.  Some systems
symlink /bin/csh to /bin/tcsh, making it almost impossible to distinguish
between csh and tcsh.  In case VIM guesses wrong you can set the
"filetype_csh" variable.  For using csh:  *g:filetype_csh*
>
        :let g:filetype_csh = "csh"

For using tcsh: >

        :let g:filetype_csh = "tcsh"

Any script with a tcsh extension or a standard tcsh filename (.tcshrc,
tcsh.tcshrc, tcsh.login) will have filetype tcsh.  All other tcsh/csh scripts
will be classified as tcsh, UNLESS the "filetype_csh" variable exists.  If the
"filetype_csh" variable exists, the filetype will be set to the value of the
variable.
```

CYNLIB *cynlib.vim* *ft-cynlib-syntax*

Cynlib files are C++ files that use the Cynlib class library to enable
hardware modelling and simulation using C++. Typically Cynlib files have a .cc
or a .cpp extension, which makes it very difficult to distinguish them from a
normal C++ file. Thus, to enable Cynlib highlighting for .cc files, add this
line to your .vimrc file: >

```
:let cynlib_cyntax_for_cc=1
```

Similarly for cpp files (this extension is only usually used in Windows) >

```
:let cynlib_cyntax_for_cpp=1
```

To disable these again, use this: >

```
:unlet cynlib_cyntax_for_cc
:unlet cynlib_cyntax_for_cpp
```
<

CWEB *cweb.vim* *ft-cweb-syntax*

Files matching "*.w" could be Progress or cweb. If the automatic detection
doesn't work for you, or you don't edit Progress at all, use this in your
startup vimrc: >
```
:let filetype_w = "cweb"
```

DESKTOP *desktop.vim* *ft-desktop-syntax*

Primary goal of this syntax file is to highlight .desktop and .directory files
according to freedesktop.org standard:
http://standards.freedesktop.org/desktop-entry-spec/latest/
But actually almost none implements this standard fully. Thus it will
highlight all Unix ini files. But you can force strict highlighting according
to standard by placing this in your vimrc file: >
```
:let enforce_freedesktop_standard = 1
```

DIFF *diff.vim*

The diff highlighting normally finds translated headers. This can be slow if
there are very long lines in the file. To disable translations: >

```
:let diff_translations = 0
```

Also see |diff-slow|.

DIRCOLORS *dircolors.vim* *ft-dircolors-syntax*

The dircolors utility highlighting definition has one option. It exists to
provide compatibility with the Slackware GNU/Linux distributions version of
the command. It adds a few keywords that are generally ignored by most
versions. On Slackware systems, however, the utility accepts the keywords and
uses them for processing. To enable the Slackware keywords add the following
line to your startup file: >
```
let dircolors_is_slackware = 1
```

DOCBOOK *docbk.vim* *ft-docbk-syntax* *docbook*

```
DOCBOOK XML                             *docbkxml.vim* *ft-docbkxml-syntax*
DOCBOOK SGML                            *docbksgml.vim* *ft-docbksgml-syntax*

There are two types of DocBook files: SGML and XML.  To specify what type you
are using the "b:docbk_type" variable should be set.  Vim does this for you
automatically if it can recognize the type.  When Vim can't guess it the type
defaults to XML.
You can set the type manually: >
        :let docbk_type = "sgml"
or: >
        :let docbk_type = "xml"
You need to do this before loading the syntax file, which is complicated.
Simpler is setting the filetype to "docbkxml" or "docbksgml": >
        :set filetype=docbksgml
or: >
        :set filetype=docbkxml

You can specify the DocBook version: >
        :let docbk_ver = 3
When not set 4 is used.


DOSBATCH                                *dosbatch.vim* *ft-dosbatch-syntax*

There is one option with highlighting DOS batch files.  This covers new
extensions to the Command Interpreter introduced with Windows 2000 and
is controlled by the variable dosbatch_cmdextversion.  For Windows NT
this should have the value 1, and for Windows 2000 it should be 2.
Select the version you want with the following line: >

   :let dosbatch_cmdextversion = 1

If this variable is not defined it defaults to a value of 2 to support
Windows 2000.

A second option covers whether *.btm files should be detected as type
"dosbatch" (MS-DOS batch files) or type "btm" (4DOS batch files).  The latter
is used by default.  You may select the former with the following line: >

   :let g:dosbatch_syntax_for_btm = 1

If this variable is undefined or zero, btm syntax is selected.


DOXYGEN                                         *doxygen.vim* *doxygen-syntax*

Doxygen generates code documentation using a special documentation format
(similar to Javadoc).  This syntax script adds doxygen highlighting to c, cpp,
idl and php files, and should also work with java.

There are a few of ways to turn on doxygen formatting. It can be done
explicitly or in a modeline by appending '.doxygen' to the syntax of the file.
Example: >
        :set syntax=c.doxygen
or >
        // vim:syntax=c.doxygen

It can also be done automatically for C, C++, C#, IDL and PHP files by setting
the global or buffer-local variable load_doxygen_syntax.  This is done by
adding the following to your .vimrc. >
        :let g:load_doxygen_syntax=1
```

There are a couple of variables that have an effect on syntax highlighting, and
are to do with non-standard highlighting options.

Variable	Default	Effect ~
g:doxygen_enhanced_color g:doxygen_enhanced_colour	0	Use non-standard highlighting for doxygen comments.
doxygen_my_rendering	0	Disable rendering of HTML bold, italic and html_my_rendering underline.
doxygen_javadoc_autobrief	1	Set to 0 to disable javadoc autobrief colour highlighting.
doxygen_end_punctuation	'[.]'	Set to regexp match for the ending punctuation of brief

There are also some hilight groups worth mentioning as they can be useful in
configuration.

Highlight	Effect ~
doxygenErrorComment	The colour of an end-comment when missing punctuation in a code, verbatim or dot section
doxygenLinkError	The colour of an end-comment when missing the \endlink from a \link section.

DTD *dtd.vim* *ft-dtd-syntax*

The DTD syntax highlighting is case sensitive by default. To disable
case-sensitive highlighting, add the following line to your startup file: >

```
:let dtd_ignore_case=1
```

The DTD syntax file will highlight unknown tags as errors. If
this is annoying, it can be turned off by setting: >

```
:let dtd_no_tag_errors=1
```

before sourcing the dtd.vim syntax file.
Parameter entity names are highlighted in the definition using the
'Type' highlighting group and 'Comment' for punctuation and '%'.
Parameter entity instances are highlighted using the 'Constant'
highlighting group and the 'Type' highlighting group for the
delimiters % and ;. This can be turned off by setting: >

```
:let dtd_no_param_entities=1
```

The DTD syntax file is also included by xml.vim to highlight included dtd's.

EIFFEL *eiffel.vim* *ft-eiffel-syntax*

While Eiffel is not case-sensitive, its style guidelines are, and the
syntax highlighting file encourages their use. This also allows to
highlight class names differently. If you want to disable case-sensitive
highlighting, add the following line to your startup file: >

```
:let eiffel_ignore_case=1
```

Case still matters for class names and TODO marks in comments.

```
Conversely, for even stricter checks, add one of the following lines: >

        :let eiffel_strict=1
        :let eiffel_pedantic=1

Setting eiffel_strict will only catch improper capitalization for the
five predefined words "Current", "Void", "Result", "Precursor", and
"NONE", to warn against their accidental use as feature or class names.

Setting eiffel_pedantic will enforce adherence to the Eiffel style
guidelines fairly rigorously (like arbitrary mixes of upper- and
lowercase letters as well as outdated ways to capitalize keywords).

If you want to use the lower-case version of "Current", "Void",
"Result", and "Precursor", you can use >

        :let eiffel_lower_case_predef=1

instead of completely turning case-sensitive highlighting off.

Support for ISE's proposed new creation syntax that is already
experimentally handled by some compilers can be enabled by: >

        :let eiffel_ise=1

Finally, some vendors support hexadecimal constants.  To handle them, add >

        :let eiffel_hex_constants=1

to your startup file.


EUPHORIA            *euphoria3.vim* *euphoria4.vim* *ft-euphoria-syntax*

Two syntax highlighting files exists for Euphoria. One for Euphoria
version 3.1.1, which is the default syntax highlighting file, and one for
Euphoria version 4.0.5 or later.

Euphoria version 3.1.1 (http://www.rapideuphoria.com/) is still necessary
for developing applications for the DOS platform, which Euphoria version 4
(http://www.openeuphoria.org/) does not support.

The following file extensions are auto-detected as Euphoria file type:

        *.e, *.eu, *.ew, *.ex, *.exu, *.exw
        *.E, *.EU, *.EW, *.EX, *.EXU, *.EXW

To select syntax highlighting file for Euphoria, as well as for
auto-detecting the *.e and *.E file extensions as Euphoria file type,
add the following line to your startup file: >

        :let filetype_euphoria="euphoria3"

        or

        :let filetype_euphoria="euphoria4"


ERLANG                                          *erlang.vim* *ft-erlang-syntax*

Erlang is a functional programming language developed by Ericsson.  Files with
the following extensions are recognized as Erlang files: erl, hrl, yaws.
```

The BIFs (built-in functions) are highlighted by default. To disable this,
put the following line in your vimrc: >

```
:let g:erlang_highlight_bifs = 0
```

To enable highlighting some special atoms, put this in your vimrc: >

```
:let g:erlang_highlight_special_atoms = 1
```

FLEXWIKI *flexwiki.vim* *ft-flexwiki-syntax*

FlexWiki is an ASP.NET-based wiki package available at http://www.flexwiki.com
NOTE: this site currently doesn't work, on Wikipedia is mentioned that
development stopped in 2009.

Syntax highlighting is available for the most common elements of FlexWiki
syntax. The associated ftplugin script sets some buffer-local options to make
editing FlexWiki pages more convenient. FlexWiki considers a newline as the
start of a new paragraph, so the ftplugin sets 'tw'=0 (unlimited line length),
'wrap' (wrap long lines instead of using horizontal scrolling), 'linebreak'
(to wrap at a character in 'breakat' instead of at the last char on screen),
and so on. It also includes some keymaps that are disabled by default.

If you want to enable the keymaps that make "j" and "k" and the cursor keys
move up and down by display lines, add this to your .vimrc: >

```
:let flexwiki_maps = 1
```

FORM *form.vim* *ft-form-syntax*

The coloring scheme for syntax elements in the FORM file uses the default
modes Conditional, Number, Statement, Comment, PreProc, Type, and String,
following the language specifications in 'Symbolic Manipulation with FORM' by
J.A.M. Vermaseren, CAN, Netherlands, 1991.

If you want include your own changes to the default colors, you have to
redefine the following syntax groups:

- formConditional
- formNumber
- formStatement
- formHeaderStatement
- formComment
- formPreProc
- formDirective
- formType
- formString

Note that the form.vim syntax file implements FORM preprocessor commands and
directives per default in the same syntax group.

A predefined enhanced color mode for FORM is available to distinguish between
header statements and statements in the body of a FORM program. To activate
this mode define the following variable in your vimrc file >

```
:let form_enhanced_color=1
```

The enhanced mode also takes advantage of additional color features for a dark
gvim display. Here, statements are colored LightYellow instead of Yellow, and
conditionals are LightBlue for better distinction.

```
FORTRAN                                   *fortran.vim* *ft-fortran-syntax*

Default highlighting and dialect ~
Highlighting appropriate for Fortran 2008 is used by default.  This choice
should be appropriate for most users most of the time because Fortran 2008 is
almost a superset of previous versions (Fortran 2003, 95, 90, and 77).

Fortran source code form ~
Fortran code can be in either fixed or free source form.  Note that the
syntax highlighting will not be correct if the form is incorrectly set.

When you create a new fortran file, the syntax script assumes fixed source
form.  If you always use free source form, then >
    :let fortran_free_source=1
in your .vimrc prior to the :syntax on command.  If you always use fixed source
form, then >
    :let fortran_fixed_source=1
in your .vimrc prior to the :syntax on command.

If the form of the source code depends, in a non-standard way, upon the file
extension, then it is most convenient to set fortran_free_source in a ftplugin
file.  For more information on ftplugin files, see |ftplugin|. Note that this
will work only if the "filetype plugin indent on" command precedes the "syntax
on" command in your .vimrc file.

When you edit an existing fortran file, the syntax script will assume free
source form if the fortran_free_source variable has been set, and assumes
fixed source form if the fortran_fixed_source variable has been set.  If
neither of these variables have been set, the syntax script attempts to
determine which source form has been used by examining the file extension
using conventions common to the ifort, gfortran, Cray, NAG, and PathScale
compilers (.f, .for, .f77 for fixed-source, .f90, .f95, .f03, .f08 for
free-source). If none of this works, then the script examines the first five
columns of the first 500 lines of your file.  If no signs of free source form
are detected, then the file is assumed to be in fixed source form.  The
algorithm should work in the vast majority of cases.  In some cases, such as a
file that begins with 500 or more full-line comments, the script may
incorrectly decide that the fortran code is in fixed form.  If that happens,
just add a non-comment statement beginning anywhere in the first five columns
of the first twenty-five lines, save (:w) and then reload (:e!) the file.

Tabs in fortran files ~
Tabs are not recognized by the Fortran standards.  Tabs are not a good idea in
fixed format fortran source code which requires fixed column boundaries.
Therefore, tabs are marked as errors.  Nevertheless, some programmers like
using tabs.  If your fortran files contain tabs, then you should set the
variable fortran_have_tabs in your .vimrc with a command such as >
    :let fortran_have_tabs=1
placed prior to the :syntax on command.  Unfortunately, the use of tabs will
mean that the syntax file will not be able to detect incorrect margins.

Syntax folding of fortran files ~
If you wish to use foldmethod=syntax, then you must first set the variable
fortran_fold with a command such as >
    :let fortran_fold=1
to instruct the syntax script to define fold regions for program units, that
is main programs starting with a program statement, subroutines, function
subprograms, block data subprograms, interface blocks, and modules.  If you
also set the variable fortran_fold_conditionals with a command such as >
    :let fortran_fold_conditionals=1
```

```
then fold regions will also be defined for do loops, if blocks, and select
case constructs.  If you also set the variable
fortran_fold_multilinecomments with a command such as >
    :let fortran_fold_multilinecomments=1
then fold regions will also be defined for three or more consecutive comment
lines.  Note that defining fold regions can be slow for large files.

If fortran_fold, and possibly fortran_fold_conditionals and/or
fortran_fold_multilinecomments, have been set, then vim will fold your file if
you set foldmethod=syntax.  Comments or blank lines placed between two program
units are not folded because they are seen as not belonging to any program
unit.

More precise fortran syntax ~
If you set the variable fortran_more_precise with a command such as >
    :let fortran_more_precise=1
then the syntax coloring will be more precise but slower.  In particular,
statement labels used in do, goto and arithmetic if statements will be
recognized, as will construct names at the end of a do, if, select or forall
construct.

Non-default fortran dialects ~
The syntax script supports two Fortran dialects: f08 and F. You will probably
find the default highlighting (f08) satisfactory.  A few legacy constructs
deleted or declared obsolescent in the 2008 standard are highlighted as todo
items.

If you use F, the advantage of setting the dialect appropriately is that
other legacy features excluded from F will be highlighted as todo items and
that free source form will be assumed.

The dialect can be selected in various ways.  If all your fortran files use
the same dialect, set the global variable fortran_dialect in your .vimrc prior
to your syntax on statement.  The case-sensitive, permissible values of
fortran_dialect are "f08" or "F".  Invalid values of fortran_dialect are
ignored.

If the dialect depends upon the file extension, then it is most convenient to
set a buffer-local variable in a ftplugin file.  For more information on
ftplugin files, see |ftplugin|.  For example, if all your fortran files with
an .f90 extension are written in the F subset, your ftplugin file should
contain the code >
    let s:extfname = expand("%:e")
    if s:extfname ==? "f90"
        let b:fortran_dialect="F"
    else
        unlet! b:fortran_dialect
    endif
Note that this will work only if the "filetype plugin indent on" command
precedes the "syntax on" command in your .vimrc file.

Finer control is necessary if the file extension does not uniquely identify
the dialect.  You can override the default dialect, on a file-by-file basis,
by including a comment with the directive "fortran_dialect=xx" (where xx=F or
f08) in one of the first three lines in your file.  For example, your older .f
files may be legacy code but your newer ones may be F codes, and you would
identify the latter by including in the first three lines of those files a
Fortran comment of the form >
  ! fortran_dialect=F

For previous versions of the syntax, you may have set fortran_dialect to the
now-obsolete values "f77", "f90", "f95", or "elf". Such settings will be
```

silently handled as "f08". Users of "elf" may wish to experiment with "F"
instead.

The syntax/fortran.vim script contains embedded comments that tell you how to
comment and/or uncomment some lines to (a) activate recognition of some
non-standard, vendor-supplied intrinsics and (b) to prevent features deleted
or declared obsolescent in the 2008 standard from being highlighted as todo
items.

Limitations ~
Parenthesis checking does not catch too few closing parentheses. Hollerith
strings are not recognized. Some keywords may be highlighted incorrectly
because Fortran90 has no reserved words.

For further information related to fortran, see |ft-fortran-indent| and
|ft-fortran-plugin|.

FVWM CONFIGURATION FILES *fvwm.vim* *ft-fvwm-syntax*

In order for Vim to recognize Fvwm configuration files that do not match
the patterns *fvwmrc* or *fvwm2rc* , you must put additional patterns
appropriate to your system in your myfiletypes.vim file. For these
patterns, you must set the variable "b:fvwm_version" to the major version
number of Fvwm, and the 'filetype' option to fvwm.

For example, to make Vim identify all files in /etc/X11/fvwm2/
as Fvwm2 configuration files, add the following: >

```
  :au! BufNewFile,BufRead /etc/X11/fvwm2/*  let b:fvwm_version = 2 |
                                          \ set filetype=fvwm
```

If you'd like Vim to highlight all valid color names, tell it where to
find the color database (rgb.txt) on your system. Do this by setting
"rgb_file" to its location. Assuming your color database is located
in /usr/X11/lib/X11/, you should add the line >

```
        :let rgb_file = "/usr/X11/lib/X11/rgb.txt"
```

to your .vimrc file.

GSP *gsp.vim* *ft-gsp-syntax*

The default coloring style for GSP pages is defined by |html.vim|, and
the coloring for java code (within java tags or inline between backticks)
is defined by |java.vim|. The following HTML groups defined in |html.vim|
are redefined to incorporate and highlight inline java code:

```
    htmlString
    htmlValue
    htmlEndTag
    htmlTag
    htmlTagN
```

Highlighting should look fine most of the places where you'd see inline
java code, but in some special cases it may not. To add another HTML
group where you will have inline java code where it does not highlight
correctly, just copy the line you want from |html.vim| and add gspJava
to the contains clause.

The backticks for inline java are highlighted according to the htmlError

group to make them easier to see.

GROFF *groff.vim* *ft-groff-syntax*

The groff syntax file is a wrapper for |nroff.vim|, see the notes
under that heading for examples of use and configuration. The purpose
of this wrapper is to set up groff syntax extensions by setting the
filetype from a |modeline| or in a personal filetype definitions file
(see |filetype.txt|).

HASKELL *haskell.vim* *lhaskell.vim* *ft-haskell-syntax*

The Haskell syntax files support plain Haskell code as well as literate
Haskell code, the latter in both Bird style and TeX style. The Haskell
syntax highlighting will also highlight C preprocessor directives.

If you want to highlight delimiter characters (useful if you have a
light-coloured background), add to your .vimrc: >
        :let hs_highlight_delimiters = 1
To treat True and False as keywords as opposed to ordinary identifiers,
add: >
        :let hs_highlight_boolean = 1
To also treat the names of primitive types as keywords: >
        :let hs_highlight_types = 1
And to treat the names of even more relatively common types as keywords: >
        :let hs_highlight_more_types = 1
If you want to highlight the names of debugging functions, put in
your .vimrc: >
        :let hs_highlight_debug = 1

The Haskell syntax highlighting also highlights C preprocessor
directives, and flags lines that start with # but are not valid
directives as erroneous. This interferes with Haskell's syntax for
operators, as they may start with #. If you want to highlight those
as operators as opposed to errors, put in your .vimrc: >
        :let hs_allow_hash_operator = 1

The syntax highlighting for literate Haskell code will try to
automatically guess whether your literate Haskell code contains
TeX markup or not, and correspondingly highlight TeX constructs
or nothing at all. You can override this globally by putting
in your .vimrc >
        :let lhs_markup = none
for no highlighting at all, or >
        :let lhs_markup = tex
to force the highlighting to always try to highlight TeX markup.
For more flexibility, you may also use buffer local versions of
this variable, so e.g. >
        :let b:lhs_markup = tex
will force TeX highlighting for a particular buffer. It has to be
set before turning syntax highlighting on for the buffer or
loading a file.

HTML *html.vim* *ft-html-syntax*

The coloring scheme for tags in the HTML file works as follows.

The <> of opening tags are colored differently than the </> of a closing tag.
This is on purpose! For opening tags the 'Function' color is used, while for

closing tags the 'Type' color is used (See syntax.vim to check how those are
defined for you)

Known tag names are colored the same way as statements in C. Unknown tag
names are colored with the same color as the <> or </> respectively which
makes it easy to spot errors

Note that the same is true for argument (or attribute) names. Known attribute
names are colored differently than unknown ones.

Some HTML tags are used to change the rendering of text. The following tags
are recognized by the html.vim syntax coloring file and change the way normal
text is shown: <B> <I> <U> <EM> <STRONG> (<EM> is used as an alias for <I>,
while <STRONG> as an alias for <B>), <H1> - <H6>, <HEAD>, <TITLE> and <A>, but
only if used as a link (that is, it must include a href as in
<A href="somefile.html">).

If you want to change how such text is rendered, you must redefine the
following syntax groups:

    - htmlBold
    - htmlBoldUnderline
    - htmlBoldUnderlineItalic
    - htmlUnderline
    - htmlUnderlineItalic
    - htmlItalic
    - htmlTitle for titles
    - htmlH1 - htmlH6 for headings

To make this redefinition work you must redefine them all with the exception
of the last two (htmlTitle and htmlH[1-6], which are optional) and define the
following variable in your vimrc (this is due to the order in which the files
are read during initialization) >
        :let html_my_rendering=1

If you'd like to see an example download mysyntax.vim at
http://www.fleiner.com/vim/download.html

You can also disable this rendering by adding the following line to your
vimrc file: >
        :let html_no_rendering=1

HTML comments are rather special (see an HTML reference document for the
details), and the syntax coloring scheme will highlight all errors.
However, if you prefer to use the wrong style (starts with <!-- and
ends with -->) you can define >
        :let html_wrong_comments=1

JavaScript and Visual Basic embedded inside HTML documents are highlighted as
'Special' with statements, comments, strings and so on colored as in standard
programming languages. Note that only JavaScript and Visual Basic are currently
supported, no other scripting language has been added yet.

Embedded and inlined cascading style sheets (CSS) are highlighted too.

There are several html preprocessor languages out there. html.vim has been
written such that it should be trivial to include it. To do so add the
following two lines to the syntax coloring file for that language
(the example comes from the asp.vim file):

    runtime! syntax/html.vim
    syn cluster htmlPreproc add=asp

Now you just need to make sure that you add all regions that contain
the preprocessor language to the cluster htmlPreproc.

HTML/OS (by Aestiva) *htmlos.vim* *ft-htmlos-syntax*

The coloring scheme for HTML/OS works as follows:

Functions and variable names are the same color by default, because VIM
doesn't specify different colors for Functions and Identifiers. To change
this (which is recommended if you want function names to be recognizable in a
different color) you need to add the following line to either your ~/.vimrc: >
  :hi Function term=underline cterm=bold ctermfg=LightGray

Of course, the ctermfg can be a different color if you choose.

Another issues that HTML/OS runs into is that there is no special filetype to
signify that it is a file with HTML/OS coding. You can change this by opening
a file and turning on HTML/OS syntax by doing the following: >
  :set syntax=htmlos

Lastly, it should be noted that the opening and closing characters to begin a
block of HTML/OS code can either be << or [[and >> or]], respectively.

IA64 *ia64.vim* *intel-itanium* *ft-ia64-syntax*

Highlighting for the Intel Itanium 64 assembly language. See |asm.vim| for
how to recognize this filetype.

To have *.inc files be recognized as IA64, add this to your .vimrc file: >
        :let g:filetype_inc = "ia64"

INFORM *inform.vim* *ft-inform-syntax*

Inform highlighting includes symbols provided by the Inform Library, as
most programs make extensive use of it. If do not wish Library symbols
to be highlighted add this to your vim startup: >
        :let inform_highlight_simple=1

By default it is assumed that Inform programs are Z-machine targeted,
and highlights Z-machine assembly language symbols appropriately. If
you intend your program to be targeted to a Glulx/Glk environment you
need to add this to your startup sequence: >
        :let inform_highlight_glulx=1

This will highlight Glulx opcodes instead, and also adds glk() to the
set of highlighted system functions.

The Inform compiler will flag certain obsolete keywords as errors when
it encounters them. These keywords are normally highlighted as errors
by Vim. To prevent such error highlighting, you must add this to your
startup sequence: >
        :let inform_suppress_obsolete=1

By default, the language features highlighted conform to Compiler
version 6.30 and Library version 6.11. If you are using an older
Inform development environment, you may with to add this to your
startup sequence: >
        :let inform_highlight_old=1

IDL *idl.vim* *idl-syntax*

IDL (Interface Definition Language) files are used to define RPC calls. In Microsoft land, this is also used for defining COM interfaces and calls.

IDL's structure is simple enough to permit a full grammar based approach to rather than using a few heuristics. The result is large and somewhat repetitive but seems to work.

There are some Microsoft extensions to idl files that are here. Some of them are disabled by defining idl_no_ms_extensions.

The more complex of the extensions are disabled by defining idl_no_extensions.

Variable	Effect ~
idl_no_ms_extensions	Disable some of the Microsoft specific extensions
idl_no_extensions	Disable complex extensions
idlsyntax_showerror	Show IDL errors (can be rather intrusive, but quite helpful)
idlsyntax_showerror_soft	Use softer colours by default for errors

JAVA *java.vim* *ft-java-syntax*

The java.vim syntax highlighting file offers several options:

In Java 1.0.2 it was never possible to have braces inside parens, so this was flagged as an error. Since Java 1.1 this is possible (with anonymous classes), and therefore is no longer marked as an error. If you prefer the old way, put the following line into your vim startup file: >

```
:let java_mark_braces_in_parens_as_errors=1
```

All identifiers in java.lang.* are always visible in all classes. To highlight them use: >

```
:let java_highlight_java_lang_ids=1
```

You can also highlight identifiers of most standard Java packages if you download the javaid.vim script at http://www.fleiner.com/vim/download.html. If you prefer to only highlight identifiers of a certain package, say java.io use the following: >

```
:let java_highlight_java_io=1
```

Check the javaid.vim file for a list of all the packages that are supported.

Function names are not highlighted, as the way to find functions depends on how you write Java code. The syntax file knows two possible ways to highlight functions:

If you write function declarations that are always indented by either a tab, 8 spaces or 2 spaces you may want to set >

```
:let java_highlight_functions="indent"
```

However, if you follow the Java guidelines about how functions and classes are supposed to be named (with respect to upper and lowercase), use >

```
:let java_highlight_functions="style"
```

If both options do not work for you, but you would still want function declarations to be highlighted create your own definitions by changing the definitions in java.vim or by creating your own java.vim which includes the original one and then adds the code to highlight functions.

In Java 1.1 the functions System.out.println() and System.err.println() should

```
only be used for debugging.  Therefore it is possible to highlight debugging
statements differently.  To do this you must add the following definition in
your startup file: >
        :let java_highlight_debug=1
The result will be that those statements are highlighted as 'Special'
characters.  If you prefer to have them highlighted differently you must define
new highlightings for the following groups.:
    Debug, DebugSpecial, DebugString, DebugBoolean, DebugType
which are used for the statement itself, special characters used in debug
strings, strings, boolean constants and types (this, super) respectively.  I
have opted to chose another background for those statements.

Javadoc is a program that takes special comments out of Java program files and
creates HTML pages.  The standard configuration will highlight this HTML code
similarly to HTML files (see |html.vim|).  You can even add Javascript
and CSS inside this code (see below).  There are four differences however:
  1. The title (all characters up to the first '.' which is followed by
     some white space or up to the first '@') is colored differently (to change
     the color change the group CommentTitle).
  2. The text is colored as 'Comment'.
  3. HTML comments are colored as 'Special'
  4. The special Javadoc tags (@see, @param, ...) are highlighted as specials
     and the argument (for @see, @param, @exception) as Function.
To turn this feature off add the following line to your startup file: >
        :let java_ignore_javadoc=1

If you use the special Javadoc comment highlighting described above you
can also turn on special highlighting for Javascript, visual basic
scripts and embedded CSS (stylesheets).  This makes only sense if you
actually have Javadoc comments that include either Javascript or embedded
CSS.  The options to use are >
        :let java_javascript=1
        :let java_css=1
        :let java_vb=1

In order to highlight nested parens with different colors define colors
for javaParen, javaParen1 and javaParen2, for example with >
        :hi link javaParen Comment
or >
        :hi javaParen ctermfg=blue guifg=#0000ff

If you notice highlighting errors while scrolling backwards, which are fixed
when redrawing with CTRL-L, try setting the "java_minlines" internal variable
to a larger number: >
        :let java_minlines = 50
This will make the syntax synchronization start 50 lines before the first
displayed line.  The default value is 10.  The disadvantage of using a larger
number is that redrawing can become slow.


LACE                                            *lace.vim* *ft-lace-syntax*

Lace (Language for Assembly of Classes in Eiffel) is case insensitive, but the
style guide lines are not.  If you prefer case insensitive highlighting, just
define the vim variable 'lace_case_insensitive' in your startup file: >
        :let lace_case_insensitive=1


LEX                                             *lex.vim* *ft-lex-syntax*

Lex uses brute-force synchronizing as the "^%%$" section delimiter
gives no clue as to what section follows.  Consequently, the value for >
```

```
        :syn sync minlines=300
may be changed by the user if s/he is experiencing synchronization
difficulties (such as may happen with large lex files).


LIFELINES                                *lifelines.vim* *ft-lifelines-syntax*

To highlight deprecated functions as errors, add in your .vimrc: >

        :let g:lifelines_deprecated = 1
<

LISP                                                *lisp.vim* *ft-lisp-syntax*

The lisp syntax highlighting provides two options: >

        g:lisp_instring : if it exists, then "(...)" strings are highlighted
                          as if the contents of the string were lisp.
                          Useful for AutoLisp.
        g:lisp_rainbow  : if it exists and is nonzero, then differing levels
                          of parenthesization will receive different
                          highlighting.
<
The g:lisp_rainbow option provides 10 levels of individual colorization for
the parentheses and backquoted parentheses.  Because of the quantity of
colorization levels, unlike non-rainbow highlighting, the rainbow mode
specifies its highlighting using ctermfg and guifg, thereby bypassing the
usual colorscheme control using standard highlighting groups.  The actual
highlighting used depends on the dark/bright setting  (see |'bg'|).


LITE                                                *lite.vim* *ft-lite-syntax*

There are two options for the lite syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this: >

        :let lite_sql_query = 1

For syncing, minlines defaults to 100.  If you prefer another value, you can
set "lite_minlines" to the value you desire.  Example: >

        :let lite_minlines = 200


LPC                                                *lpc.vim* *ft-lpc-syntax*

LPC stands for a simple, memory-efficient language: Lars Pensj| C.  The
file name of LPC is usually *.c.  Recognizing these files as LPC would bother
users writing only C programs.  If you want to use LPC syntax in Vim, you
should set a variable in your .vimrc file: >

        :let lpc_syntax_for_c = 1

If it doesn't work properly for some particular C or LPC files, use a
modeline.  For a LPC file:

        // vim:set ft=lpc:

For a C file that is recognized as LPC:

        // vim:set ft=c:
```

If you don't want to set the variable, use the modeline in EVERY LPC file.

There are several implementations for LPC, we intend to support most widely
used ones. Here the default LPC syntax is for MudOS series, for MudOS v22
and before, you should turn off the sensible modifiers, and this will also
assert the new efuns after v22 to be invalid, don't set this variable when
you are using the latest version of MudOS: >

```
        :let lpc_pre_v22 = 1
```

For LpMud 3.2 series of LPC: >

```
        :let lpc_compat_32 = 1
```

For LPC4 series of LPC: >

```
        :let lpc_use_lpc4_syntax = 1
```

For uLPC series of LPC:
uLPC has been developed to Pike, so you should use Pike syntax
instead, and the name of your source file should be *.pike

LUA *lua.vim* *ft-lua-syntax*

The Lua syntax file can be used for versions 4.0, 5.0, 5.1 and 5.2 (5.2 is
the default). You can select one of these versions using the global variables
lua_version and lua_subversion. For example, to activate Lua
5.1 syntax highlighting, set the variables like this:

```
        :let lua_version = 5
        :let lua_subversion = 1
```

MAIL *mail.vim* *ft-mail.vim*

Vim highlights all the standard elements of an email (headers, signatures,
quoted text and URLs / email addresses). In keeping with standard conventions,
signatures begin in a line containing only "--" followed optionally by
whitespaces and end with a newline.

Vim treats lines beginning with ']', '}', '|', '>' or a word followed by '>'
as quoted text. However Vim highlights headers and signatures in quoted text
only if the text is quoted with '>' (optionally followed by one space).

By default mail.vim synchronises syntax to 100 lines before the first
displayed line. If you have a slow machine, and generally deal with emails
with short headers, you can change this to a smaller value: >

```
    :let mail_minlines = 30
```

MAKE *make.vim* *ft-make-syntax*

In makefiles, commands are usually highlighted to make it easy for you to spot
errors. However, this may be too much coloring for you. You can turn this
feature off by using: >

```
        :let make_no_commands = 1
```

MAPLE *maple.vim* *ft-maple-syntax*

Maple V, by Waterloo Maple Inc, supports symbolic algebra. The language supports many packages of functions which are selectively loaded by the user. The standard set of packages' functions as supplied in Maple V release 4 may be highlighted at the user's discretion. Users may place in their .vimrc file: >

```
:let mvpkg_all= 1
```

to get all package functions highlighted, or users may select any subset by choosing a variable/package from the table below and setting that variable to 1, also in their .vimrc file (prior to sourcing $VIMRUNTIME/syntax/syntax.vim).

Table of Maple V Package Function Selectors >

```
mv_DEtools     mv_genfunc     mv_networks     mv_process
mv_Galois      mv_geometry    mv_numapprox    mv_simplex
mv_GaussInt    mv_grobner     mv_numtheory    mv_stats
mv_LREtools    mv_group       mv_orthopoly    mv_student
mv_combinat    mv_inttrans    mv_padic        mv_sumtools
mv_combstruct mv_liesymm      mv_plots        mv_tensor
mv_difforms    mv_linalg      mv_plottools    mv_totorder
mv_finance     mv_logic       mv_powseries
```

MATHEMATICA *mma.vim* *ft-mma-syntax* *ft-mathematica-syntax*

Empty *.m files will automatically be presumed to be Matlab files unless you have the following in your .vimrc: >

```
let filetype_m = "mma"
```

MOO *moo.vim* *ft-moo-syntax*

If you use C-style comments inside expressions and find it mangles your highlighting, you may want to use extended (slow!) matches for C-style comments: >

```
:let moo_extended_cstyle_comments = 1
```

To disable highlighting of pronoun substitution patterns inside strings: >

```
:let moo_no_pronoun_sub = 1
```

To disable highlighting of the regular expression operator '%|', and matching '%(' and '%)' inside strings: >

```
:let moo_no_regexp = 1
```

Unmatched double quotes can be recognized and highlighted as errors: >

```
:let moo_unmatched_quotes = 1
```

To highlight builtin properties (.name, .location, .programmer etc.): >

```
:let moo_builtin_properties = 1
```

Unknown builtin functions can be recognized and highlighted as errors. If you use this option, add your own extensions to the mooKnownBuiltinFunction group. To enable this option: >

```
        :let moo_unknown_builtin_functions = 1

An example of adding sprintf() to the list of known builtin functions: >

        :syn keyword mooKnownBuiltinFunction sprintf contained


MSQL                                                *msql.vim* *ft-msql-syntax*

There are two options for the msql syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this: >

        :let msql_sql_query = 1

For syncing, minlines defaults to 100.  If you prefer another value, you can
set "msql_minlines" to the value you desire.  Example: >

        :let msql_minlines = 200


N1QL                                                *n1ql.vim* *ft-n1ql-syntax*

N1QL is a SQL-like declarative language for manipulating JSON documents in
Couchbase Server databases.

Vim syntax highlights N1QL statements, keywords, operators, types, comments,
and special values.  Vim ignores syntactical elements specific to SQL or its
many dialects, like COLUMN or CHAR, that don't exist in N1QL.


NCF                                                 *ncf.vim* *ft-ncf-syntax*

There is one option for NCF syntax highlighting.

If you want to have unrecognized (by ncf.vim) statements highlighted as
errors, use this: >

        :let ncf_highlight_unknowns = 1

If you don't want to highlight these errors, leave it unset.


NROFF                                               *nroff.vim* *ft-nroff-syntax*

The nroff syntax file works with AT&T n/troff out of the box.  You need to
activate the GNU groff extra features included in the syntax file before you
can use them.

For example, Linux and BSD distributions use groff as their default text
processing package.  In order to activate the extra syntax highlighting
features for groff, add the following option to your start-up files: >

  :let b:nroff_is_groff = 1

Groff is different from the old AT&T n/troff that you may still find in
Solaris.  Groff macro and request names can be longer than 2 characters and
there are extensions to the language primitives.  For example, in AT&T troff
you access the year as a 2-digit number with the request \(yr.  In groff you
can use the same request, recognized for compatibility, or you can use groff's
native syntax, \[yr].  Furthermore, you can use a 4-digit year directly:
\[year].  Macro requests can be longer than 2 characters, for example, GNU mm
```

accepts the requests ".VERBON" and ".VERBOFF" for creating verbatim
environments.

In order to obtain the best formatted output g/troff can give you, you should
follow a few simple rules about spacing and punctuation.

1. Do not leave empty spaces at the end of lines.

2. Leave one space and one space only after an end-of-sentence period,
   exclamation mark, etc.

3. For reasons stated below, it is best to follow all period marks with a
   carriage return.

The reason behind these unusual tips is that g/n/troff have a line breaking
algorithm that can be easily upset if you don't follow the rules given above.

Unlike TeX, troff fills text line-by-line, not paragraph-by-paragraph and,
furthermore, it does not have a concept of glue or stretch, all horizontal and
vertical space input will be output as is.

Therefore, you should be careful about not using more space between sentences
than you intend to have in your final document. For this reason, the common
practice is to insert a carriage return immediately after all punctuation
marks. If you want to have "even" text in your final processed output, you
need to maintain regular spacing in the input text. To mark both trailing
spaces and two or more spaces after a punctuation as an error, use: >

```
  :let nroff_space_errors = 1
```

Another technique to detect extra spacing and other errors that will interfere
with the correct typesetting of your file, is to define an eye-catching
highlighting definition for the syntax groups "nroffDefinition" and
"nroffDefSpecial" in your configuration files. For example: >

```
  hi def nroffDefinition term=italic cterm=italic gui=reverse
  hi def nroffDefSpecial term=italic,bold cterm=italic,bold
                        \ gui=reverse,bold
```

If you want to navigate preprocessor entries in your source file as easily as
with section markers, you can activate the following option in your .vimrc
file: >

```
        let b:preprocs_as_sections = 1
```

As well, the syntax file adds an extra paragraph marker for the extended
paragraph macro (.XP) in the ms package.

Finally, there is a |groff.vim| syntax file that can be used for enabling
groff syntax highlighting either on a file basis or globally by default.

OCAML   *ocaml.vim* *ft-ocaml-syntax*

The OCaml syntax file handles files having the following prefixes: .ml,
.mli, .mll and .mly. By setting the following variable >

```
        :let ocaml_revised = 1
```

you can switch from standard OCaml-syntax to revised syntax as supported
by the camlp4 preprocessor. Setting the variable >

```
        :let ocaml_noend_error = 1

prevents highlighting of "end" as error, which is useful when sources
contain very long structures that Vim does not synchronize anymore.


PAPP                                                *papp.vim* *ft-papp-syntax*

The PApp syntax file handles .papp files and, to a lesser extend, .pxml
and .pxsl files which are all a mixture of perl/xml/html/other using xml
as the top-level file format.  By default everything inside phtml or pxml
sections is treated as a string with embedded preprocessor commands.  If
you set the variable: >

        :let papp_include_html=1

in your startup file it will try to syntax-hilight html code inside phtml
sections, but this is relatively slow and much too colourful to be able to
edit sensibly. ;)

The newest version of the papp.vim syntax file can usually be found at
http://papp.plan9.de.


PASCAL                                          *pascal.vim* *ft-pascal-syntax*

Files matching "*.p" could be Progress or Pascal.  If the automatic detection
doesn't work for you, or you don't edit Progress at all, use this in your
startup vimrc: >

   :let filetype_p = "pascal"

The Pascal syntax file has been extended to take into account some extensions
provided by Turbo Pascal, Free Pascal Compiler and GNU Pascal Compiler.
Delphi keywords are also supported.  By default, Turbo Pascal 7.0 features are
enabled.  If you prefer to stick with the standard Pascal keywords, add the
following line to your startup file: >

   :let pascal_traditional=1

To switch on Delphi specific constructions (such as one-line comments,
keywords, etc): >

   :let pascal_delphi=1


The option pascal_symbol_operator controls whether symbol operators such as +,
*, .., etc. are displayed using the Operator color or not.  To colorize symbol
operators, add the following line to your startup file: >

   :let pascal_symbol_operator=1

Some functions are highlighted by default.  To switch it off: >

   :let pascal_no_functions=1

Furthermore, there are specific variables for some compilers.  Besides
pascal_delphi, there are pascal_gpc and pascal_fpc.  Default extensions try to
match Turbo Pascal. >

   :let pascal_gpc=1
```

```
or >

   :let pascal_fpc=1

To ensure that strings are defined on a single line, you can define the
pascal_one_line_string variable. >

   :let pascal_one_line_string=1

If you dislike <Tab> chars, you can set the pascal_no_tabs variable.  Tabs
will be highlighted as Error. >

   :let pascal_no_tabs=1



PERL                                              *perl.vim* *ft-perl-syntax*

There are a number of possible options to the perl syntax highlighting.

Inline POD highlighting is now turned on by default.  If you don't wish
to have the added complexity of highlighting POD embedded within Perl
files, you may set the 'perl_include_pod' option to 0: >

        :let perl_include_pod = 0

To reduce the complexity of parsing (and increase performance) you can switch
off two elements in the parsing of variable names and contents. >

To handle package references in variable and function names not differently
from the rest of the name (like 'PkgName::' in '$PkgName::VarName'): >

        :let perl_no_scope_in_variables = 1

(In Vim 6.x it was the other way around: "perl_want_scope_in_variables"
enabled it.)

If you do not want complex things like '@{${"foo"}}' to be parsed: >

        :let perl_no_extended_vars = 1

(In Vim 6.x it was the other way around: "perl_extended_vars" enabled it.)

The coloring strings can be changed.  By default strings and qq friends will be
highlighted like the first line.  If you set the variable
perl_string_as_statement, it will be highlighted as in the second line.

   "hello world!"; qq|hello world|;
   ^^^^^^^^^^^^^^NN^^^^^^^^^^^^^^^N       (unlet perl_string_as_statement)
   S^^^^^^^^^^^^^SNNSSS^^^^^^^^^^^SN      (let perl_string_as_statement)

(^ = perlString, S = perlStatement, N = None at all)

The syncing has 3 options.  The first two switch off some triggering of
synchronization and should only be needed in case it fails to work properly.
If while scrolling all of a sudden the whole screen changes color completely
then you should try and switch off one of those.  Let me know if you can figure
out the line that causes the mistake.

One triggers on "^\s*sub\s*" and the other on "^[$@%]" more or less. >

        :let perl_no_sync_on_sub
```

```
        :let perl_no_sync_on_global_var

Below you can set the maximum distance VIM should look for starting points for
its attempts in syntax highlighting. >

        :let perl_sync_dist = 100

If you want to use folding with perl, set perl_fold: >

        :let perl_fold = 1

If you want to fold blocks in if statements, etc. as well set the following: >

        :let perl_fold_blocks = 1

Subroutines are folded by default if 'perl_fold' is set.  If you do not want
this, you can set 'perl_nofold_subs': >

        :let perl_nofold_subs = 1

Anonymous subroutines are not folded by default; you may enable their folding
via 'perl_fold_anonymous_subs': >

        :let perl_fold_anonymous_subs = 1

Packages are also folded by default if 'perl_fold' is set.  To disable this
behavior, set 'perl_nofold_packages': >

        :let perl_nofold_packages = 1

PHP3 and PHP4           *php.vim* *php3.vim* *ft-php-syntax* *ft-php3-syntax*

[note: previously this was called "php3", but since it now also supports php4
it has been renamed to "php"]

There are the following options for the php syntax highlighting.

If you like SQL syntax highlighting inside Strings: >

  let php_sql_query = 1

For highlighting the Baselib methods: >

  let php_baselib = 1

Enable HTML syntax highlighting inside strings: >

  let php_htmlInStrings = 1

Using the old colorstyle: >

  let php_oldStyle = 1

Enable highlighting ASP-style short tags: >

  let php_asp_tags = 1

Disable short tags: >

  let php_noShortTags = 1

For highlighting parent error ] or ): >
```

```
  let php_parent_error_close = 1

For skipping a php end tag, if there exists an open ( or [ without a closing
one: >

  let php_parent_error_open = 1

Enable folding for classes and functions: >

  let php_folding = 1

Selecting syncing method: >

  let php_sync_method = x

x = -1 to sync by search (default),
x > 0 to sync at least x lines backwards,
x = 0 to sync from start.


PLAINTEX                                *plaintex.vim* *ft-plaintex-syntax*

TeX is a typesetting language, and plaintex is the file type for the "plain"
variant of TeX.  If you never want your *.tex files recognized as plain TeX,
see |ft-tex-plugin|.

This syntax file has the option >

        let g:plaintex_delimiters = 1

if you want to highlight brackets "[]" and braces "{}".


PPWIZARD                                          *ppwiz.vim* *ft-ppwiz-syntax*

PPWizard is a preprocessor for HTML and OS/2 INF files

This syntax file has the options:

- ppwiz_highlight_defs : determines highlighting mode for PPWizard's
  definitions.  Possible values are

  ppwiz_highlight_defs = 1 : PPWizard #define statements retain the
    colors of their contents (e.g. PPWizard macros and variables)

  ppwiz_highlight_defs = 2 : preprocessor #define and #evaluate
    statements are shown in a single color with the exception of line
    continuation symbols

  The default setting for ppwiz_highlight_defs is 1.

- ppwiz_with_html : If the value is 1 (the default), highlight literal
  HTML code; if 0, treat HTML code like ordinary text.


PHTML                                             *phtml.vim* *ft-phtml-syntax*

There are two options for the phtml syntax highlighting.

If you like SQL syntax highlighting inside Strings, use this: >
```

```
        :let phtml_sql_query = 1

For syncing, minlines defaults to 100.  If you prefer another value, you can
set "phtml_minlines" to the value you desire.  Example: >

        :let phtml_minlines = 200


POSTSCRIPT                              *postscr.vim* *ft-postscr-syntax*

There are several options when it comes to highlighting PostScript.

First which version of the PostScript language to highlight.  There are
currently three defined language versions, or levels.  Level 1 is the original
and base version, and includes all extensions prior to the release of level 2.
Level 2 is the most common version around, and includes its own set of
extensions prior to the release of level 3.  Level 3 is currently the highest
level supported.  You select which level of the PostScript language you want
highlighted by defining the postscr_level variable as follows: >

        :let postscr_level=2

If this variable is not defined it defaults to 2 (level 2) since this is
the most prevalent version currently.

Note, not all PS interpreters will support all language features for a
particular language level.  In particular the %!PS-Adobe-3.0 at the start of
PS files does NOT mean the PostScript present is level 3 PostScript!

If you are working with Display PostScript, you can include highlighting of
Display PS language features by defining the postscr_display variable as
follows: >

        :let postscr_display=1

If you are working with Ghostscript, you can include highlighting of
Ghostscript specific language features by defining the variable
postscr_ghostscript as follows: >

        :let postscr_ghostscript=1

PostScript is a large language, with many predefined elements.  While it
useful to have all these elements highlighted, on slower machines this can
cause Vim to slow down.  In an attempt to be machine friendly font names and
character encodings are not highlighted by default.  Unless you are working
explicitly with either of these this should be ok.  If you want them to be
highlighted you should set one or both of the following variables: >

        :let postscr_fonts=1
        :let postscr_encodings=1

There is a stylistic option to the highlighting of and, or, and not.  In
PostScript the function of these operators depends on the types of their
operands - if the operands are booleans then they are the logical operators,
if they are integers then they are binary operators.  As binary and logical
operators can be highlighted differently they have to be highlighted one way
or the other.  By default they are treated as logical operators.  They can be
highlighted as binary operators by defining the variable
postscr_andornot_binary as follows: >

        :let postscr_andornot_binary=1
<
```

```
                        *ptcap.vim* *ft-printcap-syntax*
PRINTCAP + TERMCAP      *ft-ptcap-syntax* *ft-termcap-syntax*

This syntax file applies to the printcap and termcap databases.

In order for Vim to recognize printcap/termcap files that do not match
the patterns *printcap*, or *termcap*, you must put additional patterns
appropriate to your system in your |myfiletypefile| file.  For these
patterns, you must set the variable "b:ptcap_type" to either "print" or
"term", and then the 'filetype' option to ptcap.

For example, to make Vim identify all files in /etc/termcaps/ as termcap
files, add the following: >

   :au BufNewFile,BufRead /etc/termcaps/* let b:ptcap_type = "term" |
                                       \ set filetype=ptcap

If you notice highlighting errors while scrolling backwards, which
are fixed when redrawing with CTRL-L, try setting the "ptcap_minlines"
internal variable to a larger number: >

   :let ptcap_minlines = 50

(The default is 20 lines.)


PROGRESS                                  *progress.vim* *ft-progress-syntax*

Files matching "*.w" could be Progress or cweb.  If the automatic detection
doesn't work for you, or you don't edit cweb at all, use this in your
startup vimrc: >
   :let filetype_w = "progress"
The same happens for "*.i", which could be assembly, and "*.p", which could be
Pascal.  Use this if you don't use assembly and Pascal: >
   :let filetype_i = "progress"
   :let filetype_p = "progress"


PYTHON                                               *python.vim* *ft-python-syntax*

There are six options to control Python syntax highlighting.

For highlighted numbers: >
        :let python_no_number_highlight = 1

For highlighted builtin functions: >
        :let python_no_builtin_highlight = 1

For highlighted standard exceptions: >
        :let python_no_exception_highlight = 1

For highlighted doctests and code inside: >
        :let python_no_doctest_highlight = 1
or >
        :let python_no_doctest_code_highlight = 1
(first option implies second one).

For highlighted trailing whitespace and mix of spaces and tabs: >
        :let python_space_error_highlight = 1

If you want all possible Python highlighting (the same as setting the
```

```
preceding last option and unsetting all other ones): >
        :let python_highlight_all = 1

Note: only existence of these options matter, not their value. You can replace
      1 above with anything.


QUAKE                                           *quake.vim* *ft-quake-syntax*

The Quake syntax definition should work for most any FPS (First Person
Shooter) based on one of the Quake engines.  However, the command names vary
a bit between the three games (Quake, Quake 2, and Quake 3 Arena) so the
syntax definition checks for the existence of three global variables to allow
users to specify what commands are legal in their files.  The three variables
can be set for the following effects:

set to highlight commands only available in Quake: >
        :let quake_is_quake1 = 1

set to highlight commands only available in Quake 2: >
        :let quake_is_quake2 = 1

set to highlight commands only available in Quake 3 Arena: >
        :let quake_is_quake3 = 1

Any combination of these three variables is legal, but might highlight more
commands than are actually available to you by the game.


READLINE                                *readline.vim* *ft-readline-syntax*

The readline library is primarily used by the BASH shell, which adds quite a
few commands and options to the ones already available.  To highlight these
items as well you can add the following to your |vimrc| or just type it in the
command line before loading a file with the readline syntax: >
        let readline_has_bash = 1

This will add highlighting for the commands that BASH (version 2.05a and
later, and part earlier) adds.


RESTRUCTURED TEXT                       *rst.vim* *ft-rst-syntax*

You may set what syntax definitions should be used for code blocks via >
        let rst_syntax_code_list = ['vim', 'lisp', ...]
<

REXX                                            *rexx.vim* *ft-rexx-syntax*

If you notice highlighting errors while scrolling backwards, which are fixed
when redrawing with CTRL-L, try setting the "rexx_minlines" internal variable
to a larger number: >
        :let rexx_minlines = 50
This will make the syntax synchronization start 50 lines before the first
displayed line.  The default value is 10.  The disadvantage of using a larger
number is that redrawing can become slow.

Vim tries to guess what type a ".r" file is.  If it can't be detected (from
comment lines), the default is "r".  To make the default rexx add this line to
your .vimrc:  *g:filetype_r*
>
        :let g:filetype_r = "r"
```

```
RUBY                                                *ruby.vim* *ft-ruby-syntax*

    Ruby: Operator highlighting         |ruby_operators|
    Ruby: Whitespace errors             |ruby_space_errors|
    Ruby: Folding                       |ruby_fold| |ruby_foldable_groups|
    Ruby: Reducing expensive operations |ruby_no_expensive| |ruby_minlines|
    Ruby: Spellchecking strings         |ruby_spellcheck_strings|

                                                    *ruby_operators*
 Ruby: Operator highlighting ~

Operators can be highlighted by defining "ruby_operators": >

        :let ruby_operators = 1
<
                                                    *ruby_space_errors*
 Ruby: Whitespace errors ~

Whitespace errors can be highlighted by defining "ruby_space_errors": >

        :let ruby_space_errors = 1
<
This will highlight trailing whitespace and tabs preceded by a space character
as errors.  This can be refined by defining "ruby_no_trail_space_error" and
"ruby_no_tab_space_error" which will ignore trailing whitespace and tabs after
spaces respectively.

                                          *ruby_fold* *ruby_foldable_groups*
 Ruby: Folding ~

Folding can be enabled by defining "ruby_fold": >

        :let ruby_fold = 1
<
This will set the value of 'foldmethod' to "syntax" locally to the current
buffer or window, which will enable syntax-based folding when editing Ruby
filetypes.

Default folding is rather detailed, i.e., small syntax units like "if", "do",
"%w[]" may create corresponding fold levels.

You can set "ruby_foldable_groups" to restrict which groups are foldable: >

        :let ruby_foldable_groups = 'if case %'
<
The value is a space-separated list of keywords:

    keyword       meaning ~
    --------  ------------------------------------ ~
    ALL        Most block syntax (default)
    NONE       Nothing
    if         "if" or "unless" block
    def        "def" block
    class      "class" block
    module     "module" block
    do         "do" block
    begin      "begin" block
    case       "case" block
    for        "for", "while", "until" loops
    {          Curly bracket block or hash literal
```

```
    [           Array literal
    %           Literal with "%" notation, e.g.: %w(STRING), %!STRING!
    /           Regexp
    string      String and shell command output (surrounded by ', ", `)
    :           Symbol
    #           Multiline comment
    <<          Here documents
    __END__     Source code after "__END__" directive
```

ruby_no_expensive

Ruby: Reducing expensive operations ~

By default, the "end" keyword is colorized according to the opening statement of the block it closes. While useful, this feature can be expensive; if you experience slow redrawing (or you are on a terminal with poor color support) you may want to turn it off by defining the "ruby_no_expensive" variable: >

```
        :let ruby_no_expensive = 1
<
```

In this case the same color will be used for all control keywords.

ruby_minlines

If you do want this feature enabled, but notice highlighting errors while scrolling backwards, which are fixed when redrawing with CTRL-L, try setting the "ruby_minlines" variable to a value larger than 50: >

```
        :let ruby_minlines = 100
<
```

Ideally, this value should be a number of lines large enough to embrace your largest class or module.

ruby_spellcheck_strings

Ruby: Spellchecking strings ~

Ruby syntax will perform spellchecking of strings if you define "ruby_spellcheck_strings": >

```
        :let ruby_spellcheck_strings = 1
<
```

SCHEME *scheme.vim* *ft-scheme-syntax*

By default only R5RS keywords are highlighted and properly indented.

MzScheme-specific stuff will be used if b:is_mzscheme or g:is_mzscheme variables are defined.

Also scheme.vim supports keywords of the Chicken Scheme->C compiler. Define b:is_chicken or g:is_chicken, if you need them.

SDL *sdl.vim* *ft-sdl-syntax*

The SDL highlighting probably misses a few keywords, but SDL has so many of them it's almost impossibly to cope.

The new standard, SDL-2000, specifies that all identifiers are case-sensitive (which was not so before), and that all keywords can be used either completely lowercase or completely uppercase. To have the highlighting reflect this, you can set the following variable: >

```
        :let sdl_2000=1
```

This also sets many new keywords. If you want to disable the old
keywords, which is probably a good idea, use: >
        :let SDL_no_96=1

The indentation is probably also incomplete, but right now I am very
satisfied with it for my own projects.

SED   *sed.vim* *ft-sed-syntax*

To make tabs stand out from regular blanks (accomplished by using Todo
highlighting on the tabs), define "highlight_sedtabs" by putting >

        :let highlight_sedtabs = 1

in the vimrc file. (This special highlighting only applies for tabs
inside search patterns, replacement texts, addresses or text included
by an Append/Change/Insert command.) If you enable this option, it is
also a good idea to set the tab width to one character; by doing that,
you can easily count the number of tabs in a string.

Bugs:

  The transform command (y) is treated exactly like the substitute
  command. This means that, as far as this syntax file is concerned,
  transform accepts the same flags as substitute, which is wrong.
  (Transform accepts no flags.) I tolerate this bug because the
  involved commands need very complex treatment (95 patterns, one for
  each plausible pattern delimiter).

SGML   *sgml.vim* *ft-sgml-syntax*

The coloring scheme for tags in the SGML file works as follows.

The <> of opening tags are colored differently than the </> of a closing tag.
This is on purpose! For opening tags the 'Function' color is used, while for
closing tags the 'Type' color is used (See syntax.vim to check how those are
defined for you)

Known tag names are colored the same way as statements in C. Unknown tag
names are not colored which makes it easy to spot errors.

Note that the same is true for argument (or attribute) names. Known attribute
names are colored differently than unknown ones.

Some SGML tags are used to change the rendering of text. The following tags
are recognized by the sgml.vim syntax coloring file and change the way normal
text is shown: <varname> <emphasis> <command> <function> <literal>
<replaceable> <ulink> and <link>.

If you want to change how such text is rendered, you must redefine the
following syntax groups:

    - sgmlBold
    - sgmlBoldItalic
    - sgmlUnderline
    - sgmlItalic
    - sgmlLink for links

```
To make this redefinition work you must redefine them all and define the
following variable in your vimrc (this is due to the order in which the files
are read during initialization) >
   let sgml_my_rendering=1

You can also disable this rendering by adding the following line to your
vimrc file: >
   let sgml_no_rendering=1

(Adapted from the html.vim help text by Claudio Fleiner <claudio@fleiner.com>)


                *ft-posix-synax* *ft-dash-syntax*
SH              *sh.vim*  *ft-sh-syntax*  *ft-bash-syntax*  *ft-ksh-syntax*

This covers syntax highlighting for the older Unix (Bourne) sh, and newer
shells such as bash, dash, posix, and the Korn shells.

Vim attempts to determine which shell type is in use by specifying that
various filenames are of specific types: >

    ksh : .kshrc* *.ksh
    bash: .bashrc* bashrc bash.bashrc .bash_profile* *.bash
<
If none of these cases pertain, then the first line of the file is examined
(ex. looking for /bin/sh  /bin/ksh  /bin/bash).  If the first line specifies a
shelltype, then that shelltype is used.  However some files (ex. .profile) are
known to be shell files but the type is not apparent.  Furthermore, on many
systems sh is symbolically linked to "bash" (Linux, Windows+cygwin) or "ksh"
(Posix).

One may specify a global default by instantiating one of the following
variables in your <.vimrc>:

   ksh: >
        let g:is_kornshell = 1
<   posix:  (using this is the nearly the same as setting g:is_kornshell to 1) >
        let g:is_posix     = 1
<   bash: >
        let g:is_bash      = 1
<   sh: (default) Bourne shell >
        let g:is_sh        = 1

<   (dash users should use posix)

If there's no "#! ..." line, and the user hasn't availed himself/herself of a
default sh.vim syntax setting as just shown, then syntax/sh.vim will assume
the Bourne shell syntax.  No need to quote RFCs or market penetration
statistics in error reports, please -- just select the default version of the
sh your system uses and install the associated "let..." in your <.vimrc>.

The syntax/sh.vim file provides several levels of syntax-based folding: >

        let g:sh_fold_enabled= 0     (default, no syntax folding)
        let g:sh_fold_enabled= 1     (enable function folding)
        let g:sh_fold_enabled= 2     (enable heredoc folding)
        let g:sh_fold_enabled= 4     (enable if/do/for folding)
>
then various syntax items (ie. HereDocuments and function bodies) become
syntax-foldable (see |:syn-fold|).  You also may add these together
to get multiple types of folding: >
```

```
        let g:sh_fold_enabled= 3     (enables function and heredoc folding)

If you notice highlighting errors while scrolling backwards which are fixed
when one redraws with CTRL-L, try setting the "sh_minlines" internal variable
to a larger number.  Example: >

        let sh_minlines = 500

This will make syntax synchronization start 500 lines before the first
displayed line.  The default value is 200.  The disadvantage of using a larger
number is that redrawing can become slow.

If you don't have much to synchronize on, displaying can be very slow.  To
reduce this, the "sh_maxlines" internal variable can be set.  Example: >

        let sh_maxlines = 100
<
The default is to use the twice sh_minlines.  Set it to a smaller number to
speed up displaying.  The disadvantage is that highlight errors may appear.

syntax/sh.vim tries to flag certain problems as errors; usually things like
extra ']'s, 'done's, 'fi's, etc.  If you find the error handling problematic
for your purposes, you may suppress such error highlighting by putting
the following line in your .vimrc: >

        let g:sh_no_error= 1
<

                                                *sh-embed*  *sh-awk*
 Sh: EMBEDDING LANGUAGES~

You may wish to embed languages into sh.  I'll give an example courtesy of
Lorance Stinson on how to do this with awk as an example. Put the following
file into $HOME/.vim/after/syntax/sh/awkembed.vim: >

    " AWK Embedding:
    " ==============
    " Shamelessly ripped from aspperl.vim by Aaron Hope.
    if exists("b:current_syntax")
      unlet b:current_syntax
    endif
    syn include @AWKScript syntax/awk.vim
    syn region AWKScriptCode matchgroup=AWKCommand start=+[=\\]\@<!'+ skip=+\\'+
end=+'+ contains=@AWKScript contained
    syn region AWKScriptEmbedded matchgroup=AWKCommand start=+\<awk\>+ skip=+\\$+
end=+[=\\]\@<!'+me=e-1 contains=@shIdList,@shExprList2 nextgroup=AWKScriptCode
    syn cluster shCommandSubList add=AWKScriptEmbedded
    hi def link AWKCommand Type
<
This code will then let the awk code in the single quotes: >
        awk '...awk code here...'
be highlighted using the awk highlighting syntax.  Clearly this may be
extended to other languages.


SPEEDUP                                         *spup.vim* *ft-spup-syntax*
(AspenTech plant simulator)

The Speedup syntax file has some options:

- strict_subsections : If this variable is defined, only keywords for
  sections and subsections will be highlighted as statements but not
```

```
  other keywords (like WITHIN in the OPERATION section).

- highlight_types : Definition of this variable causes stream types
  like temperature or pressure to be highlighted as Type, not as a
  plain Identifier.  Included are the types that are usually found in
  the DECLARE section; if you defined own types, you have to include
  them in the syntax file.

- oneline_comments : this value ranges from 1 to 3 and determines the
  highlighting of # style comments.

  oneline_comments = 1 : allow normal Speedup code after an even
  number of #s.

  oneline_comments = 2 : show code starting with the second # as
  error.  This is the default setting.

  oneline_comments = 3 : show the whole line as error if it contains
  more than one #.

Since especially OPERATION sections tend to become very large due to
PRESETting variables, syncing may be critical.  If your computer is
fast enough, you can increase minlines and/or maxlines near the end of
the syntax file.


SQL                                             *sql.vim* *ft-sql-syntax*
                                  *sqlinformix.vim* *ft-sqlinformix-syntax*
                                  *sqlanywhere.vim* *ft-sqlanywhere-syntax*

While there is an ANSI standard for SQL, most database engines add their own
custom extensions.  Vim currently supports the Oracle and Informix dialects of
SQL.  Vim assumes "*.sql" files are Oracle SQL by default.

Vim currently has SQL support for a variety of different vendors via syntax
scripts.  You can change Vim's default from Oracle to any of the current SQL
supported types.  You can also easily alter the SQL dialect being used on a
buffer by buffer basis.

For more detailed instructions see |ft_sql.txt|.


TCSH                                            *tcsh.vim* *ft-tcsh-syntax*

This covers the shell named "tcsh".  It is a superset of csh.  See |csh.vim|
for how the filetype is detected.

Tcsh does not allow \" in strings unless the "backslash_quote" shell variable
is set.  If you want VIM to assume that no backslash quote constructs exist add
this line to your .vimrc: >

        :let tcsh_backslash_quote = 0

If you notice highlighting errors while scrolling backwards, which are fixed
when redrawing with CTRL-L, try setting the "tcsh_minlines" internal variable
to a larger number: >

        :let tcsh_minlines = 1000

This will make the syntax synchronization start 1000 lines before the first
displayed line.  If you set "tcsh_minlines" to "fromstart", then
synchronization is done from the start of the file. The default value for
```

tcsh_minlines is 100. The disadvantage of using a larger number is that
redrawing can become slow.

TEX *tex.vim* *ft-tex-syntax* *latex-syntax*

Tex Contents~

tex-folding *g:tex_fold_enabled*

Tex: Want Syntax Folding? ~

As of version 28 of <syntax/tex.vim>, syntax-based folding of parts, chapters,
sections, subsections, etc are supported. Put >
```
let g:tex_fold_enabled=1
```
in your <.vimrc>, and :set fdm=syntax. I suggest doing the latter via a
modeline at the end of your LaTeX file: >
```
% vim: fdm=syntax
```
If your system becomes too slow, then you might wish to look into >
```
https://vimhelp.appspot.com/vim_faq.txt.html#faq-29.7
```
<

g:tex_nospell

Tex: No Spell Checking Wanted~

If you don't want spell checking anywhere in your LaTeX document, put >
```
let g:tex_nospell=1
```
into your .vimrc. If you merely wish to suppress spell checking inside
comments only, see |g:tex_comment_nospell|.

tex-nospell *g:tex_comment_nospell*

Tex: Don't Want Spell Checking In Comments? ~

Some folks like to include things like source code in comments and so would
prefer that spell checking be disabled in comments in LaTeX files. To do
this, put the following in your <.vimrc>: >
```
let g:tex_comment_nospell= 1
```
If you want to suppress spell checking everywhere inside your LaTeX document,
see |g:tex_nospell|.

tex-verb *g:tex_verbspell*

Tex: Want Spell Checking in Verbatim Zones?~

Often verbatim regions are used for things like source code; seldom does
one want source code spell-checked. However, for those of you who do
want your verbatim zones spell-checked, put the following in your <.vimrc>: >
```
let g:tex_verbspell= 1
```
<

tex-runon *tex-stopzone*

Tex: Run-on Comments or MathZones ~

The <syntax/tex.vim> highlighting supports TeX, LaTeX, and some AmsTeX. The highlighting supports three primary zones/regions: normal, texZone, and texMathZone. Although considerable effort has been made to have these zones terminate properly, zones delineated by $..$ and $$..$$ cannot be synchronized as there's no difference between start and end patterns. Consequently, a special "TeX comment" has been provided >

```
%stopzone
```

which will forcibly terminate the highlighting of either a texZone or a texMathZone.

tex-slow *tex-sync*

Tex: Slow Syntax Highlighting? ~

If you have a slow computer, you may wish to reduce the values for >

```
:syn sync maxlines=200
:syn sync minlines=50
```

(especially the latter). If your computer is fast, you may wish to increase them. This primarily affects synchronizing (i.e. just what group, if any, is the text at the top of the screen supposed to be in?).

Another cause of slow highlighting is due to syntax-driven folding; see |tex-folding| for a way around this.

g:tex_fast

Finally, if syntax highlighting is still too slow, you may set >

```
:let g:tex_fast= ""
```

in your .vimrc. Used this way, the g:tex_fast variable causes the syntax highlighting script to avoid defining any regions and associated synchronization. The result will be much faster syntax highlighting; the price: you will no longer have as much highlighting or any syntax-based folding, and you will be missing syntax-based error checking.

You may decide that some syntax is acceptable; you may use the following table selectively to enable just some syntax highlighting: >

```
    b : allow bold and italic syntax
    c : allow texComment syntax
    m : allow texMatcher syntax (ie. {...} and [...])
    M : allow texMath syntax
    p : allow parts, chapter, section, etc syntax
    r : allow texRefZone syntax (nocite, bibliography, label, pageref, eqref)
    s : allow superscript/subscript regions
    S : allow texStyle syntax
    v : allow verbatim syntax
    V : allow texNewEnv and texNewCmd syntax
<
```

As an example, let g:tex_fast= "M" will allow math-associated highlighting but suppress all the other region-based syntax highlighting.
(also see: |g:tex_conceal| and |tex-supersub|)

tex-morecommands *tex-package*

Tex: Want To Highlight More Commands? ~

LaTeX is a programmable language, and so there are thousands of packages full of specialized LaTeX commands, syntax, and fonts. If you're using such a package you'll often wish that the distributed syntax/tex.vim would support it. However, clearly this is impractical. So please consider using the techniques in |mysyntaxfile-add| to extend or modify the highlighting provided

by syntax/tex.vim. Please consider uploading any extensions that you write,
which typically would go in $HOME/after/syntax/tex/[pkgname].vim, to
http://vim.sf.net/.

tex-error *g:tex_no_error*

Tex: Excessive Error Highlighting? ~

The <tex.vim> supports lexical error checking of various sorts. Thus,
although the error checking is ofttimes very useful, it can indicate
errors where none actually are. If this proves to be a problem for you,
you may put in your <.vimrc> the following statement: >
	let g:tex_no_error=1
and all error checking by <syntax/tex.vim> will be suppressed.

tex-math

Tex: Need a new Math Group? ~

If you want to include a new math group in your LaTeX, the following
code shows you an example as to how you might do so: >
	call TexNewMathZone(sfx,mathzone,starform)
You'll want to provide the new math group with a unique suffix
(currently, A-L and V-Z are taken by <syntax/tex.vim> itself).
As an example, consider how eqnarray is set up by <syntax/tex.vim>: >
	call TexNewMathZone("D","eqnarray",1)
You'll need to change "mathzone" to the name of your new math group,
and then to the call to it in .vim/after/syntax/tex.vim.
The "starform" variable, if true, implies that your new math group
has a starred form (ie. eqnarray*).

tex-style *b:tex_stylish*

Tex: Starting a New Style? ~

One may use "\makeatletter" in *.tex files, thereby making the use of "@" in
commands available. However, since the *.tex file doesn't have one of the
following suffices: sty cls clo dtx ltx, the syntax highlighting will flag
such use of @ as an error. To solve this: >

	:let b:tex_stylish = 1
	:set ft=tex

Putting "let g:tex_stylish=1" into your <.vimrc> will make <syntax/tex.vim>
always accept such use of @.

tex-cchar *tex-cole* *tex-conceal*

Tex: Taking Advantage of Conceal Mode~

If you have |'conceallevel'| set to 2 and if your encoding is utf-8, then a
number of character sequences can be translated into appropriate utf-8 glyphs,
including various accented characters, Greek characters in MathZones, and
superscripts and subscripts in MathZones. Not all characters can be made into
superscripts or subscripts; the constraint is due to what utf-8 supports.
In fact, only a few characters are supported as subscripts.

One way to use this is to have vertically split windows (see |CTRL-W_v|); one
with |'conceallevel'| at 0 and the other at 2; and both using |'scrollbind'|.

g:tex_conceal

Tex: Selective Conceal Mode~

You may selectively use conceal mode by setting g:tex_conceal in your
<.vimrc>. By default, g:tex_conceal is set to "admgs" to enable concealment
for the following sets of characters: >

```
        a = accents/ligatures
        b = bold and italic
        d = delimiters
        m = math symbols
        g = Greek
        s = superscripts/subscripts
<
By leaving one or more of these out, the associated conceal-character
substitution will not be made.

                                                *g:tex_isk* *g:tex_stylish*
 Tex: Controlling iskeyword~

Normally, LaTeX keywords support 0-9, a-z, A-z, and 192-255 only. Latex
keywords don't support the underscore - except when in *.sty files.  The
syntax highlighting script handles this with the following logic:

        * If g:tex_stylish exists and is 1
                then the file will be treated as a "sty" file, so the "_"
                will be allowed as part of keywords
                (regardless of g:tex_isk)
        * Else if the file's suffix is sty, cls, clo, dtx, or ltx,
                then the file will be treated as a "sty" file, so the "_"
                will be allowed as part of keywords
                (regardless of g:tex_isk)

        * If g:tex_isk exists, then it will be used for the local 'iskeyword'
        * Else the local 'iskeyword' will be set to 48-57,a-z,A-Z,192-255

                        *tex-supersub* *g:tex_superscripts* *g:tex_subscripts*
 Tex: Fine Subscript and Superscript Control~

        See |tex-conceal| for how to enable concealed character replacement.

        See |g:tex_conceal| for selectively concealing accents, bold/italic,
        math, Greek, and superscripts/subscripts.

        One may exert fine control over which superscripts and subscripts one
        wants syntax-based concealment for (see |:syn-cchar|).  Since not all
        fonts support all characters, one may override the
        concealed-replacement lists; by default these lists are given by: >

            let g:tex_superscripts= "[0-9a-zA-W.,:;+-<>/()=]"
            let g:tex_subscripts= "[0-9aehijklmnoprstuvx,+-/().]"
<
        For example, I use Luxi Mono Bold; it doesn't support subscript
        characters for "hklmnpst", so I put >
                let g:tex_subscripts= "[0-9aeijoruvx,+-/().]"
<       in ~/.vim/ftplugin/tex/tex.vim in order to avoid having inscrutable
        utf-8 glyphs appear.


TF                                              *tf.vim* *ft-tf-syntax*

There is one option for the tf syntax highlighting.

For syncing, minlines defaults to 100.  If you prefer another value, you can
set "tf_minlines" to the value you desire.  Example: >

        :let tf_minlines = your choice
<
```

```
VIM                     *vim.vim*               *ft-vim-syntax*
                        *g:vimsyn_minlines*     *g:vimsyn_maxlines*
There is a trade-off between more accurate syntax highlighting versus screen
updating speed.  To improve accuracy, you may wish to increase the
g:vimsyn_minlines variable.  The g:vimsyn_maxlines variable may be used to
improve screen updating rates (see |:syn-sync| for more on this). >

        g:vimsyn_minlines : used to set synchronization minlines
        g:vimsyn_maxlines : used to set synchronization maxlines
<
        (g:vim_minlines and g:vim_maxlines are deprecated variants of
        these two options)

                                                    *g:vimsyn_embed*
The g:vimsyn_embed option allows users to select what, if any, types of
embedded script highlighting they wish to have. >

   g:vimsyn_embed == 0   : don't support any embedded scripts
   g:vimsyn_embed =~ 'l' : support embedded lua
   g:vimsyn_embed =~ 'm' : support embedded mzscheme
   g:vimsyn_embed =~ 'p' : support embedded perl
   g:vimsyn_embed =~ 'P' : support embedded python
   g:vimsyn_embed =~ 'r' : support embedded ruby
   g:vimsyn_embed =~ 't' : support embedded tcl
<
By default, g:vimsyn_embed is a string supporting interpreters that your vim
itself supports.  Concatenate multiple characters to support multiple types
of embedded interpreters; ie. g:vimsyn_embed= "mp" supports embedded mzscheme
and embedded perl.
                                                    *g:vimsyn_folding*

Some folding is now supported with syntax/vim.vim: >

   g:vimsyn_folding == 0 or doesn't exist: no syntax-based folding
   g:vimsyn_folding =~ 'a' : augroups
   g:vimsyn_folding =~ 'f' : fold functions
   g:vimsyn_folding =~ 'l' : fold lua      script
   g:vimsyn_folding =~ 'm' : fold mzscheme script
   g:vimsyn_folding =~ 'p' : fold perl     script
   g:vimsyn_folding =~ 'P' : fold python   script
   g:vimsyn_folding =~ 'r' : fold ruby     script
   g:vimsyn_folding =~ 't' : fold tcl      script
<
                                                            *g:vimsyn_noerror*
Not all error highlighting that syntax/vim.vim does may be correct; Vim script
is a difficult language to highlight correctly.  A way to suppress error
highlighting is to put the following line in your |vimrc|: >

        let g:vimsyn_noerror = 1
<


XF86CONFIG                              *xf86conf.vim* *ft-xf86conf-syntax*

The syntax of XF86Config file differs in XFree86 v3.x and v4.x.  Both
variants are supported.  Automatic detection is used, but is far from perfect.
You may need to specify the version manually.  Set the variable
xf86conf_xfree86_version to 3 or 4 according to your XFree86 version in
your .vimrc.  Example: >
        :let xf86conf_xfree86_version=3
When using a mix of versions, set the b:xf86conf_xfree86_version variable.
```

```
Note that spaces and underscores in option names are not supported.  Use
"SyncOnGreen" instead of "__s yn con gr_e_e_n" if you want the option name
highlighted.


XML                                                 *xml.vim* *ft-xml-syntax*

Xml namespaces are highlighted by default.  This can be inhibited by
setting a global variable: >

        :let g:xml_namespace_transparent=1
<
                                                                *xml-folding*
The xml syntax file provides syntax |folding| (see |:syn-fold|) between
start and end tags.  This can be turned on by >

        :let g:xml_syntax_folding = 1
        :set foldmethod=syntax

Note: syntax folding might slow down syntax highlighting significantly,
especially for large files.


X Pixmaps (XPM)                                     *xpm.vim* *ft-xpm-syntax*

xpm.vim creates its syntax items dynamically based upon the contents of the
XPM file.  Thus if you make changes e.g. in the color specification strings,
you have to source it again e.g. with ":set syn=xpm".

To copy a pixel with one of the colors, yank a "pixel" with "yl" and insert it
somewhere else with "P".

Do you want to draw with the mouse?  Try the following: >
   :function! GetPixel()
   :   let c = getline(".")[col(".") - 1]
   :   echo c
   :   exe "noremap <LeftMouse> <LeftMouse>r".c
   :   exe "noremap <LeftDrag>  <LeftMouse>r".c
   :endfunction
   :noremap <RightMouse> <LeftMouse>:call GetPixel()<CR>
   :set guicursor=n:hor20          " to see the color beneath the cursor
This turns the right button into a pipette and the left button into a pen.
It will work with XPM files that have one character per pixel only and you
must not click outside of the pixel strings, but feel free to improve it.

It will look much better with a font in a quadratic cell size, e.g. for X: >
        :set guifont=-*-clean-medium-r-*-*-8-*-*-*-*-80-*


YAML                                                *yaml.vim* *ft-yaml-syntax*

                                          *g:yaml_schema* *b:yaml_schema*
A YAML schema is a combination of a set of tags and a mechanism for resolving
non-specific tags. For user this means that YAML parser may, depending on
plain scalar contents, treat plain scalar (which can actually be only string
and nothing else) as a value of the other type: null, boolean, floating-point,
integer. `g:yaml_schema` option determines according to which schema values
will be highlighted specially. Supported schemas are

Schema          Description ~
failsafe        No additional highlighting.
json            Supports JSON-style numbers, booleans and null.
```

```
core            Supports more number, boolean and null styles.
pyyaml          In addition to core schema supports highlighting timestamps,
                but there are some differences in what is recognized as
                numbers and many additional boolean values not present in core
                schema.

Default schema is `core`.

Note that schemas are not actually limited to plain scalars, but this is the
only difference between schemas defined in YAML specification and the only
difference defined in the syntax file.


ZSH                                                  *zsh.vim* *ft-zsh-syntax*

The syntax script for zsh allows for syntax-based folding: >

        :let g:zsh_fold_enable = 1

==============================================================================
5. Defining a syntax                                  *:syn-define* *E410*

Vim understands three types of syntax items:

1. Keyword
   It can only contain keyword characters, according to the 'iskeyword'
   option.  It cannot contain other syntax items.  It will only match with a
   complete word (there are no keyword characters before or after the match).
   The keyword "if" would match in "if(a=b)", but not in "ifdef x", because
   "(" is not a keyword character and "d" is.

2. Match
   This is a match with a single regexp pattern.

3. Region
   This starts at a match of the "start" regexp pattern and ends with a match
   with the "end" regexp pattern.  Any other text can appear in between.  A
   "skip" regexp pattern can be used to avoid matching the "end" pattern.

Several syntax ITEMs can be put into one syntax GROUP.  For a syntax group
you can give highlighting attributes.  For example, you could have an item
to define a "/* .. */" comment and another one that defines a "// .." comment,
and put them both in the "Comment" group.  You can then specify that a
"Comment" will be in bold font and have a blue color.  You are free to make
one highlight group for one syntax item, or put all items into one group.
This depends on how you want to specify your highlighting attributes.  Putting
each item in its own group results in having to specify the highlighting
for a lot of groups.

Note that a syntax group and a highlight group are similar.  For a highlight
group you will have given highlight attributes.  These attributes will be used
for the syntax group with the same name.

In case more than one item matches at the same position, the one that was
defined LAST wins.  Thus you can override previously defined syntax items by
using an item that matches the same text.  But a keyword always goes before a
match or region.  And a keyword with matching case always goes before a
keyword with ignoring case.


PRIORITY                                              *:syn-priority*
```

```
When several syntax items may match, these rules are used:

1. When multiple Match or Region items start in the same position, the item
   defined last has priority.
2. A Keyword has priority over Match and Region items.
3. An item that starts in an earlier position has priority over items that
   start in later positions.


DEFINING CASE                                           *:syn-case* *E390*

:sy[ntax] case [match | ignore]
        This defines if the following ":syntax" commands will work with
        matching case, when using "match", or with ignoring case, when using
        "ignore".  Note that any items before this are not affected, and all
        items until the next ":syntax case" command are affected.

:sy[ntax] case
        Show either "syntax case match" or "syntax case ignore" (translated).

SPELL CHECKING                                          *:syn-spell*

:sy[ntax] spell [toplevel | notoplevel | default]
        This defines where spell checking is to be done for text that is not
        in a syntax item:

        toplevel:       Text is spell checked.
        notoplevel:     Text is not spell checked.
        default:        When there is a @Spell cluster no spell checking.

        For text in syntax items use the @Spell and @NoSpell clusters
        |spell-syntax|.  When there is no @Spell and no @NoSpell cluster then
        spell checking is done for "default" and "toplevel".

        To activate spell checking the 'spell' option must be set.

:sy[ntax] spell
        Show either "syntax spell toplevel", "syntax spell notoplevel" or
        "syntax spell default" (translated).


SYNTAX ISKEYWORD SETTING                                *:syn-iskeyword*

:sy[ntax] iskeyword [clear | {option}]
        This defines the keyword characters.  It's like the 'iskeyword' option
        for but only applies to syntax highlighting.

        clear:          Syntax specific iskeyword setting is disabled and the
                        buffer-local 'iskeyword' setting is used.
        {option}        Set the syntax 'iskeyword' option to a new value.

        Example: >
  :syntax iskeyword @,48-57,192-255,$,_
<
        This would set the syntax specific iskeyword option to include all
        alphabetic characters, plus the numeric characters, all accented
        characters and also includes the "_" and the "$".

        If no argument is given, the current value will be output.

        Setting this option influences what |/\k| matches in syntax patterns
        and also determines where |:syn-keyword| will be checked for a new
```

```
        match.

        It is recommended when writing syntax files, to use this command to
        set the correct value for the specific syntax language and not change
        the 'iskeyword' option.

DEFINING KEYWORDS                                       *:syn-keyword*

:sy[ntax] keyword {group-name} [{options}] {keyword} .. [{options}]

        This defines a number of keywords.

        {group-name}    Is a syntax group name such as "Comment".
        [{options}]     See |:syn-arguments| below.
        {keyword} ..    Is a list of keywords which are part of this group.

        Example: >
  :syntax keyword   Type   int long char
<
        The {options} can be given anywhere in the line.  They will apply to
        all keywords given, also for options that come after a keyword.
        These examples do exactly the same: >
  :syntax keyword   Type   contained int long char
  :syntax keyword   Type   int long contained char
  :syntax keyword   Type   int long char contained
<                                                           *E789* *E890*
        When you have a keyword with an optional tail, like Ex commands in
        Vim, you can put the optional characters inside [], to define all the
        variations at once: >
  :syntax keyword   vimCommand   ab[breviate] n[ext]
<
        Don't forget that a keyword can only be recognized if all the
        characters are included in the 'iskeyword' option.  If one character
        isn't, the keyword will never be recognized.
        Multi-byte characters can also be used.  These do not have to be in
        'iskeyword'.
        See |:syn-iskeyword| for defining syntax specific iskeyword settings.

        A keyword always has higher priority than a match or region, the
        keyword is used if more than one item matches.  Keywords do not nest
        and a keyword can't contain anything else.

        Note that when you have a keyword that is the same as an option (even
        one that isn't allowed here), you can not use it.  Use a match
        instead.

        The maximum length of a keyword is 80 characters.

        The same keyword can be defined multiple times, when its containment
        differs.  For example, you can define the keyword once not contained
        and use one highlight group, and once contained, and use a different
        highlight group.  Example: >
  :syn keyword vimCommand tag
  :syn keyword vimSetting contained tag
<       When finding "tag" outside of any syntax item, the "vimCommand"
        highlight group is used.  When finding "tag" in a syntax item that
        contains "vimSetting", the "vimSetting" group is used.


DEFINING MATCHES                                        *:syn-match*

:sy[ntax] match {group-name} [{options}]
```

```
                [excludenl]
                [keepend]
                {pattern}
                [{options}]

        This defines one match.

        {group-name}            A syntax group name such as "Comment".
        [{options}]             See |:syn-arguments| below.
        [excludenl]             Don't make a pattern with the end-of-line "$"
                                extend a containing match or region.  Must be
                                given before the pattern. |:syn-excludenl|
        keepend                 Don't allow contained matches to go past a
                                match with the end pattern.  See
                                |:syn-keepend|.
        {pattern}               The search pattern that defines the match.
                                See |:syn-pattern| below.
                                Note that the pattern may match more than one
                                line, which makes the match depend on where
                                Vim starts searching for the pattern.  You
                                need to make sure syncing takes care of this.

        Example (match a character constant): >
  :syntax match Character /'.'/hs=s+1,he=e-1
<

DEFINING REGIONS        *:syn-region* *:syn-start* *:syn-skip* *:syn-end*
                                                        *E398* *E399*
:sy[ntax] region {group-name} [{options}]
                [matchgroup={group-name}]
                [keepend]
                [extend]
                [excludenl]
                start={start_pattern} ..
                [skip={skip_pattern}]
                end={end_pattern} ..
                [{options}]

        This defines one region.  It may span several lines.

        {group-name}            A syntax group name such as "Comment".
        [{options}]             See |:syn-arguments| below.
        [matchgroup={group-name}]  The syntax group to use for the following
                                start or end pattern matches only.  Not used
                                for the text in between the matched start and
                                end patterns.  Use NONE to reset to not using
                                a different group for the start or end match.
                                See |:syn-matchgroup|.
        keepend                 Don't allow contained matches to go past a
                                match with the end pattern.  See
                                |:syn-keepend|.
        extend                  Override a "keepend" for an item this region
                                is contained in.  See |:syn-extend|.
        excludenl               Don't make a pattern with the end-of-line "$"
                                extend a containing match or item.  Only
                                useful for end patterns.  Must be given before
                                the patterns it applies to. |:syn-excludenl|
        start={start_pattern}   The search pattern that defines the start of
                                the region.  See |:syn-pattern| below.
        skip={skip_pattern}     The search pattern that defines text inside
                                the region where not to look for the end
                                pattern.  See |:syn-pattern| below.
```

```
        end={end_pattern}       The search pattern that defines the end of
                                the region.  See |:syn-pattern| below.

        Example: >
  :syntax region String   start=+"+  skip=+\\"+  end=+"+
<
        The start/skip/end patterns and the options can be given in any order.
        There can be zero or one skip pattern.  There must be one or more
        start and end patterns.  This means that you can omit the skip
        pattern, but you must give at least one start and one end pattern.  It
        is allowed to have white space before and after the equal sign
        (although it mostly looks better without white space).

        When more than one start pattern is given, a match with one of these
        is sufficient.  This means there is an OR relation between the start
        patterns.  The last one that matches is used.  The same is true for
        the end patterns.

        The search for the end pattern starts right after the start pattern.
        Offsets are not used for this.  This implies that the match for the
        end pattern will never overlap with the start pattern.

        The skip and end pattern can match across line breaks, but since the
        search for the pattern can start in any line it often does not do what
        you want.  The skip pattern doesn't avoid a match of an end pattern in
        the next line.  Use single-line patterns to avoid trouble.

        Note: The decision to start a region is only based on a matching start
        pattern.  There is no check for a matching end pattern.  This does NOT
        work: >
                :syn region First  start="("  end=":"
                :syn region Second start="("  end=";"
<       The Second always matches before the First (last defined pattern has
        higher priority).  The Second region then continues until the next
        ';', no matter if there is a ':' before it.  Using a match does work: >
                :syn match First  "(\_.\{-}:"
                :syn match Second "(\_.\{-};"
<       This pattern matches any character or line break with "\_." and
        repeats that with "\{-}" (repeat as few as possible).

                                                        *:syn-keepend*
        By default, a contained match can obscure a match for the end pattern.
        This is useful for nesting.  For example, a region that starts with
        "{" and ends with "}", can contain another region.  An encountered "}"
        will then end the contained region, but not the outer region:
            {           starts outer "{}" region
                {       starts contained "{}" region
                }       ends contained "{}" region
            }           ends outer "{} region
        If you don't want this, the "keepend" argument will make the matching
        of an end pattern of the outer region also end any contained item.
        This makes it impossible to nest the same region, but allows for
        contained items to highlight parts of the end pattern, without causing
        that to skip the match with the end pattern.  Example: >
  :syn match  vimComment +"[^"]\+$+
  :syn region vimCommand start="set" end="$" contains=vimComment keepend
<       The "keepend" makes the vimCommand always end at the end of the line,
        even though the contained vimComment includes a match with the <EOL>.

        When "keepend" is not used, a match with an end pattern is retried
        after each contained match.  When "keepend" is included, the first
        encountered match with an end pattern is used, truncating any
```

```
        contained matches.
                                                        *:syn-extend*
        The "keepend" behavior can be changed by using the "extend" argument.
        When an item with "extend" is contained in an item that uses
        "keepend", the "keepend" is ignored and the containing region will be
        extended.
        This can be used to have some contained items extend a region while
        others don't.  Example: >

   :syn region htmlRef start=+<a>+ end=+</a>+ keepend contains=htmlItem,htmlScript
   :syn match htmlItem +<[^>]*>+ contained
   :syn region htmlScript start=+<script+ end=+</script[^>]*>+ contained extend

<       Here the htmlItem item does not make the htmlRef item continue
        further, it is only used to highlight the <> items.  The htmlScript
        item does extend the htmlRef item.

        Another example: >
   :syn region xmlFold start="<a>" end="</a>" fold transparent keepend extend
<       This defines a region with "keepend", so that its end cannot be
        changed by contained items, like when the "</a>" is matched to
        highlight it differently.  But when the xmlFold region is nested (it
        includes itself), the "extend" applies, so that the "</a>" of a nested
        region only ends that region, and not the one it is contained in.

                                                        *:syn-excludenl*
        When a pattern for a match or end pattern of a region includes a '$'
        to match the end-of-line, it will make a region item that it is
        contained in continue on the next line.  For example, a match with
        "\\$" (backslash at the end of the line) can make a region continue
        that would normally stop at the end of the line.  This is the default
        behavior.  If this is not wanted, there are two ways to avoid it:
        1. Use "keepend" for the containing item.  This will keep all
           contained matches from extending the match or region.  It can be
           used when all contained items must not extend the containing item.
        2. Use "excludenl" in the contained item.  This will keep that match
           from extending the containing match or region.  It can be used if
           only some contained items must not extend the containing item.
           "excludenl" must be given before the pattern it applies to.

                                                        *:syn-matchgroup*
        "matchgroup" can be used to highlight the start and/or end pattern
        differently than the body of the region.  Example: >
  :syntax region String matchgroup=Quote start=+"+  skip=+\\"+  end=+"+
<       This will highlight the quotes with the "Quote" group, and the text in
        between with the "String" group.
        The "matchgroup" is used for all start and end patterns that follow,
        until the next "matchgroup".  Use "matchgroup=NONE" to go back to not
        using a matchgroup.

        In a start or end pattern that is highlighted with "matchgroup" the
        contained items of the region are not used.  This can be used to avoid
        that a contained item matches in the start or end pattern match.  When
        using "transparent", this does not apply to a start or end pattern
        match that is highlighted with "matchgroup".

        Here is an example, which highlights three levels of parentheses in
        different colors: >
   :sy region par1 matchgroup=par1 start=/(/ end=/)/ contains=par2
   :sy region par2 matchgroup=par2 start=/(/ end=/)/ contains=par3 contained
   :sy region par3 matchgroup=par3 start=/(/ end=/)/ contains=par1 contained
   :hi par1 ctermfg=red guifg=red
```

```
   :hi par2 ctermfg=blue guifg=blue
   :hi par3 ctermfg=darkgreen guifg=darkgreen
<
                                                    *E849*
The maximum number of syntax groups is 19999.

==============================================================================
6. :syntax arguments                                    *:syn-arguments*

The :syntax commands that define syntax items take a number of arguments.
The common ones are explained here.  The arguments may be given in any order
and may be mixed with patterns.

Not all commands accept all arguments.  This table shows which arguments
can not be used for all commands:
                                                        *E395*
                     contains  oneline   fold  display  extend concealends~
:syntax keyword          -       -       -       -       -      -
:syntax match           yes      -      yes     yes     yes     -
:syntax region          yes     yes     yes     yes     yes    yes

These arguments can be used for all three commands:
        conceal
        cchar
        contained
        containedin
        nextgroup
        transparent
        skipwhite
        skipnl
        skipempty

conceal                                         *conceal* *:syn-conceal*

When the "conceal" argument is given, the item is marked as concealable.
Whether or not it is actually concealed depends on the value of the
'conceallevel' option.  The 'concealcursor' option is used to decide whether
concealable items in the current line are displayed unconcealed to be able to
edit the line.
Another way to conceal text is with |matchadd()|.

concealends                                             *:syn-concealends*

When the "concealends" argument is given, the start and end matches of
the region, but not the contents of the region, are marked as concealable.
Whether or not they are actually concealed depends on the setting on the
'conceallevel' option. The ends of a region can only be concealed separately
in this way when they have their own highlighting via "matchgroup"

cchar                                                   *:syn-cchar*
                                                        *E844*
The "cchar" argument defines the character shown in place of the item
when it is concealed (setting "cchar" only makes sense when the conceal
argument is given.) If "cchar" is not set then the default conceal
character defined in the 'listchars' option is used.  The character cannot be
a control character such as Tab.  Example: >
   :syntax match Entity "&amp;" conceal cchar=&
See |hl-Conceal| for highlighting.

contained                                               *:syn-contained*

When the "contained" argument is given, this item will not be recognized at
```

```
the top level, but only when it is mentioned in the "contains" field of
another match.  Example: >
   :syntax keyword Todo    TODO    contained
   :syntax match   Comment "//.*"  contains=Todo


display                                                 *:syn-display*

If the "display" argument is given, this item will be skipped when the
detected highlighting will not be displayed.  This will speed up highlighting,
by skipping this item when only finding the syntax state for the text that is
to be displayed.

Generally, you can use "display" for match and region items that meet these
conditions:
- The item does not continue past the end of a line.  Example for C: A region
  for a "/*" comment can't contain "display", because it continues on the next
  line.
- The item does not contain items that continue past the end of the line or
  make it continue on the next line.
- The item does not change the size of any item it is contained in.  Example
  for C: A match with "\\$" in a preprocessor match can't have "display",
  because it may make that preprocessor match shorter.
- The item does not allow other items to match that didn't match otherwise,
  and that item may extend the match too far.  Example for C: A match for a
  "//" comment can't use "display", because a "/*" inside that comment would
  match then and start a comment which extends past the end of the line.

Examples, for the C language, where "display" can be used:
- match with a number
- match with a label


transparent                                             *:syn-transparent*

If the "transparent" argument is given, this item will not be highlighted
itself, but will take the highlighting of the item it is contained in.  This
is useful for syntax items that don't need any highlighting but are used
only to skip over a part of the text.

The "contains=" argument is also inherited from the item it is contained in,
unless a "contains" argument is given for the transparent item itself.  To
avoid that unwanted items are contained, use "contains=NONE".  Example, which
highlights words in strings, but makes an exception for "vim": >
        :syn match myString /'[^']*'/ contains=myWord,myVim
        :syn match myWord   /\<[a-z]*\>/ contained
        :syn match myVim    /\<vim\>/ transparent contained contains=NONE
        :hi link myString String
        :hi link myWord   Comment
Since the "myVim" match comes after "myWord" it is the preferred match (last
match in the same position overrules an earlier one).  The "transparent"
argument makes the "myVim" match use the same highlighting as "myString".  But
it does not contain anything.  If the "contains=NONE" argument would be left
out, then "myVim" would use the contains argument from myString and allow
"myWord" to be contained, which will be highlighted as a Constant.  This
happens because a contained match doesn't match inside itself in the same
position, thus the "myVim" match doesn't overrule the "myWord" match here.

When you look at the colored text, it is like looking at layers of contained
items.  The contained item is on top of the item it is contained in, thus you
see the contained item.  When a contained item is transparent, you can look
through, thus you see the item it is contained in.  In a picture:
```

```
               look from here

            |   |   |   |   |   |
            V   V   V   V   V   V

               xxxx       yyy           more contained items
            ...................         contained item (transparent)
         ============================   first item
```

The 'x', 'y' and '=' represent a highlighted syntax item. The '.' represent a transparent group.

What you see is:

```
        =======xxxx=======yyy========
```

Thus you look through the transparent "....".

oneline *:syn-oneline*

The "oneline" argument indicates that the region does not cross a line boundary. It must match completely in the current line. However, when the region has a contained item that does cross a line boundary, it continues on the next line anyway. A contained item can be used to recognize a line continuation pattern. But the "end" pattern must still match in the first line, otherwise the region doesn't even start.

When the start pattern includes a "\n" to match an end-of-line, the end pattern must be found in the same line as where the start pattern ends. The end pattern may also include an end-of-line. Thus the "oneline" argument means that the end of the start pattern and the start of the end pattern must be within one line. This can't be changed by a skip pattern that matches a line break.

fold *:syn-fold*

The "fold" argument makes the fold level increase by one for this item. Example: >

```
   :syn region myFold start="{" end="}" transparent fold
   :syn sync fromstart
   :set foldmethod=syntax
```

This will make each {} block form one fold.

The fold will start on the line where the item starts, and end where the item ends. If the start and end are within the same line, there is no fold. The 'foldnestmax' option limits the nesting of syntax folds. {not available when Vim was compiled without |+folding| feature}

:syn-contains *E405* *E406* *E407* *E408* *E409*

contains={group-name},..

The "contains" argument is followed by a list of syntax group names. These groups will be allowed to begin inside the item (they may extend past the containing group's end). This allows for recursive nesting of matches and regions. If there is no "contains" argument, no groups will be contained in this item. The group names do not need to be defined before they can be used here.

```
contains=ALL
                If the only item in the contains list is "ALL", then all
                groups will be accepted inside the item.

contains=ALLBUT,{group-name},..
                If the first item in the contains list is "ALLBUT", then all
                groups will be accepted inside the item, except the ones that
                are listed.  Example: >
  :syntax region Block start="{" end="}" ... contains=ALLBUT,Function

contains=TOP
                If the first item in the contains list is "TOP", then all
                groups will be accepted that don't have the "contained"
                argument.
contains=TOP,{group-name},..
                Like "TOP", but excluding the groups that are listed.

contains=CONTAINED
                If the first item in the contains list is "CONTAINED", then
                all groups will be accepted that have the "contained"
                argument.
contains=CONTAINED,{group-name},..
                Like "CONTAINED", but excluding the groups that are
                listed.


The {group-name} in the "contains" list can be a pattern.  All group names
that match the pattern will be included (or excluded, if "ALLBUT" is used).
The pattern cannot contain white space or a ','.  Example: >
   ... contains=Comment.*,Keyw[0-3]
The matching will be done at moment the syntax command is executed.  Groups
that are defined later will not be matched.  Also, if the current syntax
command defines a new group, it is not matched.  Be careful: When putting
syntax commands in a file you can't rely on groups NOT being defined, because
the file may have been sourced before, and ":syn clear" doesn't remove the
group names.

The contained groups will also match in the start and end patterns of a
region.  If this is not wanted, the "matchgroup" argument can be used
|:syn-matchgroup|.  The "ms=" and "me=" offsets can be used to change the
region where contained items do match.  Note that this may also limit the
area that is highlighted


containedin={group-name}...                              *:syn-containedin*

The "containedin" argument is followed by a list of syntax group names.  The
item will be allowed to begin inside these groups.  This works as if the
containing item has a "contains=" argument that includes this item.

The {group-name}... can be used just like for "contains", as explained above.

This is useful when adding a syntax item afterwards.  An item can be told to
be included inside an already existing item, without changing the definition
of that item.  For example, to highlight a word in a C comment after loading
the C syntax: >
        :syn keyword myword HELP containedin=cComment contained
Note that "contained" is also used, to avoid that the item matches at the top
level.

Matches for "containedin" are added to the other places where the item can
appear.  A "contains" argument may also be added as usual.  Don't forget that
```

```
keywords never contain another item, thus adding them to "containedin" won't
work.


nextgroup={group-name},..                                   *:syn-nextgroup*

The "nextgroup" argument is followed by a list of syntax group names,
separated by commas (just like with "contains", so you can also use patterns).

If the "nextgroup" argument is given, the mentioned syntax groups will be
tried for a match, after the match or region ends.  If none of the groups have
a match, highlighting continues normally.  If there is a match, this group
will be used, even when it is not mentioned in the "contains" field of the
current group.  This is like giving the mentioned group priority over all
other groups.  Example: >
   :syntax match  ccFoobar  "Foo.\{-}Bar"  contains=ccFoo
   :syntax match  ccFoo     "Foo"           contained nextgroup=ccFiller
   :syntax region ccFiller  start="."  matchgroup=ccBar  end="Bar"  contained

This will highlight "Foo" and "Bar" differently, and only when there is a
"Bar" after "Foo".  In the text line below, "f" shows where ccFoo is used for
highlighting, and "bbb" where ccBar is used. >

   Foo asdfasd Bar asdf Foo asdf Bar asdf
   fff         bbb      fff      bbb

Note the use of ".\{-}" to skip as little as possible until the next Bar.
when ".*" would be used, the "asdf" in between "Bar" and "Foo" would be
highlighted according to the "ccFoobar" group, because the ccFooBar match
would include the first "Foo" and the last "Bar" in the line (see |pattern|).


skipwhite                                                   *:syn-skipwhite*
skipnl                                                      *:syn-skipnl*
skipempty                                                   *:syn-skipempty*

These arguments are only used in combination with "nextgroup".  They can be
used to allow the next group to match after skipping some text:
        skipwhite       skip over space and tab characters
        skipnl          skip over the end of a line
        skipempty       skip over empty lines (implies a "skipnl")

When "skipwhite" is present, the white space is only skipped if there is no
next group that matches the white space.

When "skipnl" is present, the match with nextgroup may be found in the next
line.  This only happens when the current item ends at the end of the current
line!  When "skipnl" is not present, the nextgroup will only be found after
the current item in the same line.

When skipping text while looking for a next group, the matches for other
groups are ignored.  Only when no next group matches, other items are tried
for a match again.  This means that matching a next group and skipping white
space and <EOL>s has a higher priority than other items.

Example: >
  :syn match ifstart "\<if.*"     nextgroup=ifline skipwhite skipempty
  :syn match ifline  "[^ \t].*" nextgroup=ifline skipwhite skipempty contained
  :syn match ifline  "endif"    contained
Note that the "[^ \t].*" match matches all non-white text.  Thus it would also
match "endif".  Therefore the "endif" match is put last, so that it takes
precedence.
```

```
Note that this example doesn't work for nested "if"s.  You need to add
"contains" arguments to make that work (omitted for simplicity of the
example).

IMPLICIT CONCEAL                                        *:syn-conceal-implicit*

:sy[ntax] conceal [on|off]
        This defines if the following ":syntax" commands will define keywords,
        matches or regions with the "conceal" flag set. After ":syn conceal
        on", all subsequent ":syn keyword", ":syn match" or ":syn region"
        defined will have the "conceal" flag set implicitly. ":syn conceal
        off" returns to the normal state where the "conceal" flag must be
        given explicitly.

:sy[ntax] conceal
        Show either "syntax conceal on" or "syntax conceal off" (translated).

==============================================================================
7. Syntax patterns                                *:syn-pattern* *E401* *E402*

In the syntax commands, a pattern must be surrounded by two identical
characters.  This is like it works for the ":s" command.  The most common to
use is the double quote.  But if the pattern contains a double quote, you can
use another character that is not used in the pattern.  Examples: >
  :syntax region Comment  start="/\*"  end="\*/"
  :syntax region String   start=+"+    end=+"+   skip=+\\"+

See |pattern| for the explanation of what a pattern is.  Syntax patterns are
always interpreted like the 'magic' option is set, no matter what the actual
value of 'magic' is.  And the patterns are interpreted like the 'l' flag is
not included in 'cpoptions'.  This was done to make syntax files portable and
independent of 'compatible' and 'magic' settings.

Try to avoid patterns that can match an empty string, such as "[a-z]*".
This slows down the highlighting a lot, because it matches everywhere.

                                                  *:syn-pattern-offset*
The pattern can be followed by a character offset.  This can be used to
change the highlighted part, and to change the text area included in the
match or region (which only matters when trying to match other items).  Both
are relative to the matched pattern.  The character offset for a skip
pattern can be used to tell where to continue looking for an end pattern.

The offset takes the form of "{what}={offset}"
The {what} can be one of seven strings:

ms      Match Start     offset for the start of the matched text
me      Match End       offset for the end of the matched text
hs      Highlight Start offset for where the highlighting starts
he      Highlight End   offset for where the highlighting ends
rs      Region Start    offset for where the body of a region starts
re      Region End      offset for where the body of a region ends
lc      Leading Context offset past "leading context" of pattern

The {offset} can be:

s       start of the matched pattern
s+{nr}  start of the matched pattern plus {nr} chars to the right
s-{nr}  start of the matched pattern plus {nr} chars to the left
e       end of the matched pattern
e+{nr}  end of the matched pattern plus {nr} chars to the right
e-{nr}  end of the matched pattern plus {nr} chars to the left
```

```
{nr}    (for "lc" only): start matching {nr} chars right of the start

Examples: "ms=s+1", "hs=e-2", "lc=3".

Although all offsets are accepted after any pattern, they are not always
meaningful.  This table shows which offsets are actually used:

                    ms   me   hs   he   rs   re   lc ~
match item          yes  yes  yes  yes  -    -    yes
region item start   yes  -    yes  -    yes  -    yes
region item skip    -    yes  -    -    -    -    yes
region item end     -    yes  -    yes  -    yes  yes

Offsets can be concatenated, with a ',' in between.  Example: >
  :syn match String  /"[^"]*"/hs=s+1,he=e-1
<
    some "string" text
         ^^^^^^               highlighted

Notes:
- There must be no white space between the pattern and the character
  offset(s).
- The highlighted area will never be outside of the matched text.
- A negative offset for an end pattern may not always work, because the end
  pattern may be detected when the highlighting should already have stopped.
- Before Vim 7.2 the offsets were counted in bytes instead of characters.
  This didn't work well for multi-byte characters, so it was changed with the
  Vim 7.2 release.
- The start of a match cannot be in a line other than where the pattern
  matched.  This doesn't work: "a\nb"ms=e.  You can make the highlighting
  start in another line, this does work: "a\nb"hs=e.

Example (match a comment but don't highlight the /* and */): >
  :syntax region Comment start="/\*"hs=e+1 end="\*/"he=s-1
<
        /* this is a comment */
          ^^^^^^^^^^^^^^^^^^    highlighted

A more complicated Example: >
  :syn region Exa matchgroup=Foo start="foo"hs=s+2,rs=e+2 matchgroup=Bar
end="bar"me=e-1,he=e-1,re=s-1
<
         abcfoostringbarabc
            mmmmmmmmmm     match
              sssrrreee    highlight start/region/end ("Foo", "Exa" and "Bar")

Leading context                 *:syn-lc* *:syn-leading* *:syn-context*

Note: This is an obsolete feature, only included for backwards compatibility
with previous Vim versions.  It's now recommended to use the |/\@<=| construct
in the pattern.

The "lc" offset specifies leading context -- a part of the pattern that must
be present, but is not considered part of the match.  An offset of "lc=n" will
cause Vim to step back n columns before attempting the pattern match, allowing
characters which have already been matched in previous patterns to also be
used as leading context for this match.  This can be used, for instance, to
specify that an "escaping" character must not precede the match: >

  :syn match ZNoBackslash "[^\\]z"ms=s+1
  :syn match WNoBackslash "[^\\]w"lc=1
  :syn match Underline "_\+"
```

<
```
         ___zzzz ___wwww
         ^^^     ^^^     matches Underline
             ^ ^         matches ZNoBackslash
                   ^^^^ matches WNoBackslash
```

The "ms" offset is automatically set to the same value as the "lc" offset,
unless you set "ms" explicitly.

Multi-line patterns *:syn-multi-line*

The patterns can include "\n" to match an end-of-line. Mostly this works as
expected, but there are a few exceptions.

When using a start pattern with an offset, the start of the match is not
allowed to start in a following line. The highlighting can start in a
following line though. Using the "\zs" item also requires that the start of
the match doesn't move to another line.

The skip pattern can include the "\n", but the search for an end pattern will
continue in the first character of the next line, also when that character is
matched by the skip pattern. This is because redrawing may start in any line
halfway a region and there is no check if the skip pattern started in a
previous line. For example, if the skip pattern is "a\nb" and an end pattern
is "b", the end pattern does match in the second line of this: >
```
        x x a
        b x x
```
Generally this means that the skip pattern should not match any characters
after the "\n".

External matches *:syn-ext-match*

These extra regular expression items are available in region patterns:

/\z(*/\z(\)* *E50* *E52* *E879*
\z(\) Marks the sub-expression as "external", meaning that it can be
accessed from another pattern match. Currently only usable in
defining a syntax region start pattern.

/\z1 */\z2* */\z3* */\z4* */\z5*
\z1 ... \z9 */\z6* */\z7* */\z8* */\z9* *E66* *E67*
Matches the same string that was matched by the corresponding
sub-expression in a previous start pattern match.

Sometimes the start and end patterns of a region need to share a common
sub-expression. A common example is the "here" document in Perl and many Unix
shells. This effect can be achieved with the "\z" special regular expression
items, which marks a sub-expression as "external", in the sense that it can be
referenced from outside the pattern in which it is defined. The here-document
example, for instance, can be done like this: >
```
  :syn region hereDoc start="<<\z(\I\i*\)" end="^\z1$"
```

As can be seen here, the \z actually does double duty. In the start pattern,
it marks the "\(\I\i*\)" sub-expression as external; in the end pattern, it
changes the \z1 back-reference into an external reference referring to the
first external sub-expression in the start pattern. External references can
also be used in skip patterns: >
```
  :syn region foo start="start \(\I\i*\)" skip="not end \z1" end="end \z1"
```

Note that normal and external sub-expressions are completely orthogonal and

```
indexed separately; for instance, if the pattern "\z(..\)\(..\)" is applied
to the string "aabb", then \1 will refer to "bb" and \z1 will refer to "aa".
Note also that external sub-expressions cannot be accessed as back-references
within the same pattern like normal sub-expressions.  If you want to use one
sub-expression as both a normal and an external sub-expression, you can nest
the two, as in "\(\z(...\)\)".

Note that only matches within a single line can be used.  Multi-line matches
cannot be referred to.

==============================================================================
8. Syntax clusters                                      *:syn-cluster* *E400*

:sy[ntax] cluster {cluster-name} [contains={group-name}..]
                                 [add={group-name}..]
                                 [remove={group-name}..]

This command allows you to cluster a list of syntax groups together under a
single name.

        contains={group-name}..
                The cluster is set to the specified list of groups.
        add={group-name}..
                The specified groups are added to the cluster.
        remove={group-name}..
                The specified groups are removed from the cluster.

A cluster so defined may be referred to in a contains=.., containedin=..,
nextgroup=.., add=..  or remove=.. list with a "@" prefix.  You can also use
this notation to implicitly declare a cluster before specifying its contents.

Example: >
   :syntax match Thing "# [^#]\+ #" contains=@ThingMembers
   :syntax cluster ThingMembers contains=ThingMember1,ThingMember2

As the previous example suggests, modifications to a cluster are effectively
retroactive; the membership of the cluster is checked at the last minute, so
to speak: >
   :syntax keyword A aaa
   :syntax keyword B bbb
   :syntax cluster AandB contains=A
   :syntax match Stuff "( aaa bbb )" contains=@AandB
   :syntax cluster AandB add=B    " now both keywords are matched in Stuff

This also has implications for nested clusters: >
   :syntax keyword A aaa
   :syntax keyword B bbb
   :syntax cluster SmallGroup contains=B
   :syntax cluster BigGroup contains=A,@SmallGroup
   :syntax match Stuff "( aaa bbb )" contains=@BigGroup
   :syntax cluster BigGroup remove=B    " no effect, since B isn't in BigGroup
   :syntax cluster SmallGroup remove=B  " now bbb isn't matched within Stuff
<
                                                        *E848*
The maximum number of clusters is 9767.

==============================================================================
9. Including syntax files                              *:syn-include* *E397*

It is often useful for one language's syntax file to include a syntax file for
a related language.  Depending on the exact relationship, this can be done in
two different ways:
```

```
        - If top-level syntax items in the included syntax file are to be
          allowed at the top level in the including syntax, you can simply use
          the |:runtime| command: >

  " In cpp.vim:
  :runtime! syntax/c.vim
  :unlet b:current_syntax

<       - If top-level syntax items in the included syntax file are to be
          contained within a region in the including syntax, you can use the
          ":syntax include" command:

:sy[ntax] include [@{grouplist-name}] {file-name}

          All syntax items declared in the included file will have the
          "contained" flag added.  In addition, if a group list is specified,
          all top-level syntax items in the included file will be added to
          that list. >

   " In perl.vim:
   :syntax include @Pod <sfile>:p:h/pod.vim
   :syntax region perlPOD start="^=head" end="^=cut" contains=@Pod
<
          When {file-name} is an absolute path (starts with "/", "c:", "$VAR"
          or "<sfile>") that file is sourced.  When it is a relative path
          (e.g., "syntax/pod.vim") the file is searched for in 'runtimepath'.
          All matching files are loaded.  Using a relative path is
          recommended, because it allows a user to replace the included file
          with his own version, without replacing the file that does the ":syn
          include".

                                                *E847*
The maximum number of includes is 999.

==============================================================================
10. Synchronizing                               *:syn-sync* *E403* *E404*

Vim wants to be able to start redrawing in any position in the document.  To
make this possible it needs to know the syntax state at the position where
redrawing starts.

:sy[ntax] sync [ccomment [group-name] | minlines={N} | ...]

There are four ways to synchronize:
1. Always parse from the start of the file.
   |:syn-sync-first|
2. Based on C-style comments.  Vim understands how C-comments work and can
   figure out if the current line starts inside or outside a comment.
   |:syn-sync-second|
3. Jumping back a certain number of lines and start parsing there.
   |:syn-sync-third|
4. Searching backwards in the text for a pattern to sync on.
   |:syn-sync-fourth|

                                *:syn-sync-maxlines* *:syn-sync-minlines*
For the last three methods, the line range where the parsing can start is
limited by "minlines" and "maxlines".

If the "minlines={N}" argument is given, the parsing always starts at least
that many lines backwards.  This can be used if the parsing may take a few
lines before it's correct, or when it's not possible to use syncing.
```

If the "maxlines={N}" argument is given, the number of lines that are searched
for a comment or syncing pattern is restricted to N lines backwards (after
adding "minlines"). This is useful if you have few things to sync on and a
slow machine. Example: >
   :syntax sync maxlines=500 ccomment
<
:syn-sync-linebreaks
When using a pattern that matches multiple lines, a change in one line may
cause a pattern to no longer match in a previous line. This means has to
start above where the change was made. How many lines can be specified with
the "linebreaks" argument. For example, when a pattern may include one line
break use this: >
   :syntax sync linebreaks=1
The result is that redrawing always starts at least one line before where a
change was made. The default value for "linebreaks" is zero. Usually the
value for "minlines" is bigger than "linebreaks".

First syncing method: *:syn-sync-first*
>
   :syntax sync fromstart

The file will be parsed from the start. This makes syntax highlighting
accurate, but can be slow for long files. Vim caches previously parsed text,
so that it's only slow when parsing the text for the first time. However,
when making changes some part of the text needs to be parsed again (worst
case: to the end of the file).

Using "fromstart" is equivalent to using "minlines" with a very large number.

Second syncing method: *:syn-sync-second* *:syn-sync-ccomment*

For the second method, only the "ccomment" argument needs to be given.
Example: >
   :syntax sync ccomment

When Vim finds that the line where displaying starts is inside a C-style
comment, the last region syntax item with the group-name "Comment" will be
used. This requires that there is a region with the group-name "Comment"!
An alternate group name can be specified, for example: >
   :syntax sync ccomment javaComment
This means that the last item specified with "syn region javaComment" will be
used for the detected C comment region. This only works properly if that
region does have a start pattern "\/*" and an end pattern "*\/".

The "maxlines" argument can be used to restrict the search to a number of
lines. The "minlines" argument can be used to at least start a number of
lines back (e.g., for when there is some construct that only takes a few
lines, but it hard to sync on).

Note: Syncing on a C comment doesn't work properly when strings are used
that cross a line and contain a "*/". Since letting strings cross a line
is a bad programming habit (many compilers give a warning message), and the
chance of a "*/" appearing inside a comment is very small, this restriction
is hardly ever noticed.

Third syncing method: *:syn-sync-third*

For the third method, only the "minlines={N}" argument needs to be given.

```
Vim will subtract {N} from the line number and start parsing there.  This
means {N} extra lines need to be parsed, which makes this method a bit slower.
Example: >
   :syntax sync minlines=50

"lines" is equivalent to "minlines" (used by older versions).


Fourth syncing method:                          *:syn-sync-fourth*

The idea is to synchronize on the end of a few specific regions, called a
sync pattern.  Only regions can cross lines, so when we find the end of some
region, we might be able to know in which syntax item we are.  The search
starts in the line just above the one where redrawing starts.  From there
the search continues backwards in the file.

This works just like the non-syncing syntax items.  You can use contained
matches, nextgroup, etc.  But there are a few differences:
- Keywords cannot be used.
- The syntax items with the "sync" keyword form a completely separated group
  of syntax items.  You can't mix syncing groups and non-syncing groups.
- The matching works backwards in the buffer (line by line), instead of
  forwards.
- A line continuation pattern can be given.  It is used to decide which group
  of lines need to be searched like they were one line.  This means that the
  search for a match with the specified items starts in the first of the
  consecutive that contain the continuation pattern.
- When using "nextgroup" or "contains", this only works within one line (or
  group of continued lines).
- When using a region, it must start and end in the same line (or group of
  continued lines).  Otherwise the end is assumed to be at the end of the
  line (or group of continued lines).
- When a match with a sync pattern is found, the rest of the line (or group of
  continued lines) is searched for another match.  The last match is used.
  This is used when a line can contain both the start end the end of a region
  (e.g., in a C-comment like /* this */, the last "*/" is used).

There are two ways how a match with a sync pattern can be used:
1. Parsing for highlighting starts where redrawing starts (and where the
   search for the sync pattern started).  The syntax group that is expected
   to be valid there must be specified.  This works well when the regions
   that cross lines cannot contain other regions.
2. Parsing for highlighting continues just after the match.  The syntax group
   that is expected to be present just after the match must be specified.
   This can be used when the previous method doesn't work well.  It's much
   slower, because more text needs to be parsed.
Both types of sync patterns can be used at the same time.

Besides the sync patterns, other matches and regions can be specified, to
avoid finding unwanted matches.

[The reason that the sync patterns are given separately, is that mostly the
search for the sync point can be much simpler than figuring out the
highlighting.  The reduced number of patterns means it will go (much)
faster.]

                                          *syn-sync-grouphere* *E393* *E394*
    :syntax sync match {sync-group-name} grouphere {group-name} "pattern" ..

        Define a match that is used for syncing.  {group-name} is the
        name of a syntax group that follows just after the match.  Parsing
        of the text for highlighting starts just after the match.  A region
```

must exist for this {group-name}. The first one defined will be used.
"NONE" can be used for when there is no syntax group after the match.

syn-sync-groupthere
:syntax sync match {sync-group-name} groupthere {group-name} "pattern" ..

Like "grouphere", but {group-name} is the name of a syntax group that is to be used at the start of the line where searching for the sync point started. The text between the match and the start of the sync pattern searching is assumed not to change the syntax highlighting. For example, in C you could search backwards for "/*" and "*/". If "/*" is found first, you know that you are inside a comment, so the "groupthere" is "cComment". If "*/" is found first, you know that you are not in a comment, so the "groupthere" is "NONE". (in practice it's a bit more complicated, because the "/*" and "*/" could appear inside a string. That's left as an exercise to the reader...).

:syntax sync match ..
:syntax sync region ..

Without a "groupthere" argument. Define a region or match that is skipped while searching for a sync point.

syn-sync-linecont
:syntax sync linecont {pattern}

When {pattern} matches in a line, it is considered to continue in the next line. This means that the search for a sync point will consider the lines to be concatenated.

If the "maxlines={N}" argument is given too, the number of lines that are searched for a match is restricted to N. This is useful if you have very few things to sync on and a slow machine. Example: >
:syntax sync maxlines=100

You can clear all sync settings with: >
:syntax sync clear

You can clear specific sync patterns with: >
:syntax sync clear {sync-group-name} ..

==
11. Listing syntax items *:syntax* *:sy* *:syn* *:syn-list*

This command lists all the syntax items: >

:sy[ntax] [list]

To show the syntax items for one syntax group: >

:sy[ntax] list {group-name}

To list the syntax groups in one cluster: *E392* >

:sy[ntax] list @{cluster-name}

See above for other arguments for the ":syntax" command.

Note that the ":syntax" command can be abbreviated to ":sy", although ":syn" is mostly used, because it looks better.

==

12. Highlight command *:highlight* *:hi* *E28* *E411* *E415*

There are three types of highlight groups:
- The ones used for specific languages. For these the name starts with the
  name of the language. Many of these don't have any attributes, but are
  linked to a group of the second type.
- The ones used for all syntax languages.
- The ones used for the 'highlight' option.

hitest.vim

You can see all the groups currently active with this command: >
    :so $VIMRUNTIME/syntax/hitest.vim
This will open a new window containing all highlight group names, displayed
in their own color.

:colo *:colorscheme* *E185*

:colo[rscheme] Output the name of the currently active color scheme.
This is basically the same as >
:echo g:colors_name
< In case g:colors_name has not been defined :colo will
output "default". When compiled without the |+eval|
feature it will output "unknown".

:colo[rscheme] {name} Load color scheme {name}. This searches 'runtimepath'
for the file "colors/{name}.vim". The first one that
is found is loaded.
Also searches all plugins in 'packpath', first below
"start" and then under "opt".

Doesn't work recursively, thus you can't use
":colorscheme" in a color scheme script.

To customize a colorscheme use another name, e.g.
"~/.vim/colors/mine.vim", and use `:runtime` to load
the original colorscheme: >
runtime colors/evening.vim
hi Statement ctermfg=Blue guifg=Blue

< After the color scheme has been loaded the
|ColorScheme| autocommand event is triggered.
For info about writing a colorscheme file: >
:edit $VIMRUNTIME/colors/README.txt

:hi[ghlight] List all the current highlight groups that have
attributes set.

:hi[ghlight] {group-name}
List one highlight group.

:hi[ghlight] clear Reset all highlighting to the defaults. Removes all
highlighting for groups added by the user!
Uses the current value of 'background' to decide which
default colors to use.

:hi[ghlight] clear {group-name}
:hi[ghlight] {group-name} NONE
Disable the highlighting for one highlight group. It
is _not_ set back to the default colors.

:hi[ghlight] [default] {group-name} {key}={arg} ..
Add a highlight group, or change the highlighting for
an existing group.
See |highlight-args| for the {key}={arg} arguments.

```
                        See |:highlight-default| for the optional [default]
                        argument.

Normally a highlight group is added once when starting up.  This sets the
default values for the highlighting.  After that, you can use additional
highlight commands to change the arguments that you want to set to non-default
values.  The value "NONE" can be used to switch the value off or go back to
the default value.

A simple way to change colors is with the |:colorscheme| command.  This loads
a file with ":highlight" commands such as this: >

   :hi Comment  gui=bold

Note that all settings that are not included remain the same, only the
specified field is used, and settings are merged with previous ones.  So, the
result is like this single command has been used: >
   :hi Comment  term=bold ctermfg=Cyan guifg=#80a0ff gui=bold
<
                                                          *:highlight-verbose*
When listing a highlight group and 'verbose' is non-zero, the listing will
also tell where it was last set.  Example: >
        :verbose hi Comment
<       Comment        xxx term=bold ctermfg=4 guifg=Blue ~
           Last set from /home/mool/vim/vim7/runtime/syntax/syncolor.vim ~

When ":hi clear" is used then the script where this command is used will be
mentioned for the default values. See |:verbose-cmd| for more information.

                                        *highlight-args* *E416* *E417* *E423*
There are three types of terminals for highlighting:
term    a normal terminal (vt100, xterm)
cterm   a color terminal (MS-DOS console, color-xterm, these have the "Co"
        termcap entry)
gui     the GUI

For each type the highlighting can be given.  This makes it possible to use
the same syntax file on all terminals, and use the optimal highlighting.

1. highlight arguments for normal terminals

                                        *bold* *underline* *undercurl*
                                        *inverse* *italic* *standout*
                                        *nocombine* *strikethrough*
term={attr-list}                        *attr-list* *highlight-term* *E418*
        attr-list is a comma separated list (without spaces) of the
        following items (in any order):
                bold
                underline
                undercurl       not always available
                strikethrough   not always available
                reverse
                inverse         same as reverse
                italic
                standout
                nocombine       override attributes instead of combining them
                NONE            no attributes used (used to reset it)

        Note that "bold" can be used here and by using a bold font.  They
        have the same effect.
        "undercurl" is a curly underline.  When "undercurl" is not possible
        then "underline" is used.  In general "undercurl" and "strikethrough"
```

```
        is only available in the GUI.  The color is set with |highlight-guisp|.

start={term-list}                               *highlight-start* *E422*
stop={term-list}                                *term-list* *highlight-stop*
        These lists of terminal codes can be used to get
        non-standard attributes on a terminal.

        The escape sequence specified with the "start" argument
        is written before the characters in the highlighted
        area.  It can be anything that you want to send to the
        terminal to highlight this area.  The escape sequence
        specified with the "stop" argument is written after the
        highlighted area.  This should undo the "start" argument.
        Otherwise the screen will look messed up.

        The {term-list} can have two forms:

        1. A string with escape sequences.
           This is any string of characters, except that it can't start with
           "t_" and blanks are not allowed.  The <> notation is recognized
           here, so you can use things like "<Esc>" and "<Space>".  Example:
                start=<Esc>[27h;<Esc>[<Space>r;

        2. A list of terminal codes.
           Each terminal code has the form "t_xx", where "xx" is the name of
           the termcap entry.  The codes have to be separated with commas.
           White space is not allowed.  Example:
                start=t_C1,t_BL
           The terminal codes must exist for this to work.


2. highlight arguments for color terminals

cterm={attr-list}                                       *highlight-cterm*
        See above for the description of {attr-list} |attr-list|.
        The "cterm" argument is likely to be different from "term", when
        colors are used.  For example, in a normal terminal comments could
        be underlined, in a color terminal they can be made Blue.
        Note: Many terminals (e.g., DOS console) can't mix these attributes
        with coloring.  Use only one of "cterm=" OR "ctermfg=" OR "ctermbg=".

ctermfg={color-nr}                              *highlight-ctermfg* *E421*
ctermbg={color-nr}                              *highlight-ctermbg*
        The {color-nr} argument is a color number.  Its range is zero to
        (not including) the number given by the termcap entry "Co".
        The actual color with this number depends on the type of terminal
        and its settings.  Sometimes the color also depends on the settings of
        "cterm".  For example, on some systems "cterm=bold ctermfg=3" gives
        another color, on others you just get color 3.

        For an xterm this depends on your resources, and is a bit
        unpredictable.  See your xterm documentation for the defaults.  The
        colors for a color-xterm can be changed from the .Xdefaults file.
        Unfortunately this means that it's not possible to get the same colors
        for each user.  See |xterm-color| for info about color xterms.

        The MSDOS standard colors are fixed (in a console window), so these
        have been used for the names.  But the meaning of color names in X11
        are fixed, so these color settings have been used, to make the
        highlighting settings portable (complicated, isn't it?).  The
        following names are recognized, with the color number used:
```

cterm-colors

NR-16	NR-8	COLOR NAME ~
0	0	Black
1	4	DarkBlue
2	2	DarkGreen
3	6	DarkCyan
4	1	DarkRed
5	5	DarkMagenta
6	3	Brown, DarkYellow
7	7	LightGray, LightGrey, Gray, Grey
8	0*	DarkGray, DarkGrey
9	4*	Blue, LightBlue
10	2*	Green, LightGreen
11	6*	Cyan, LightCyan
12	1*	Red, LightRed
13	5*	Magenta, LightMagenta
14	3*	Yellow, LightYellow
15	7*	White

The number under "NR-16" is used for 16-color terminals ('t_Co' greater than or equal to 16). The number under "NR-8" is used for 8-color terminals ('t_Co' less than 16). The '*' indicates that the bold attribute is set for ctermfg. In many 8-color terminals (e.g., "linux"), this causes the bright colors to appear. This doesn't work for background colors! Without the '*' the bold attribute is removed. If you want to set the bold attribute in a different way, put a "cterm=" argument AFTER the "ctermfg=" or "ctermbg=" argument. Or use a number instead of a color name.

The case of the color names is ignored.
Note that for 16 color ansi style terminals (including xterms), the numbers in the NR-8 column is used. Here '*' means 'add 8' so that Blue is 12, DarkGray is 8 etc.

Note that for some color terminals these names may result in the wrong colors!

You can also use "NONE" to remove the color.

:hi-normal-cterm

When setting the "ctermfg" or "ctermbg" colors for the Normal group, these will become the colors used for the non-highlighted text.
Example: >

```
:highlight Normal ctermfg=grey ctermbg=darkblue
```

<

When setting the "ctermbg" color for the Normal group, the 'background' option will be adjusted automatically, under the condition that the color is recognized and 'background' was not set explicitly. This causes the highlight groups that depend on 'background' to change! This means you should set the colors for Normal first, before setting other colors.
When a colorscheme is being used, changing 'background' causes it to be reloaded, which may reset all colors (including Normal). First delete the "g:colors_name" variable when you don't want this.

When you have set "ctermfg" or "ctermbg" for the Normal group, Vim needs to reset the color when exiting. This is done with the "op" termcap entry |t_op|. If this doesn't work correctly, try setting the 't_op' option in your .vimrc.

E419 *E420*

When Vim knows the normal foreground and background colors, "fg" and "bg" can be used as color names. This only works after setting the colors for the Normal group and for the MS-DOS console. Example, for

```
        reverse video: >
            :highlight Visual ctermfg=bg ctermbg=fg
<       Note that the colors are used that are valid at the moment this
        command are given.  If the Normal group colors are changed later, the
        "fg" and "bg" colors will not be adjusted.


3. highlight arguments for the GUI

gui={attr-list}                                         *highlight-gui*
        These give the attributes to use in the GUI mode.
        See |attr-list| for a description.
        Note that "bold" can be used here and by using a bold font.  They
        have the same effect.
        Note that the attributes are ignored for the "Normal" group.

font={font-name}                                        *highlight-font*
        font-name is the name of a font, as it is used on the system Vim
        runs on.  For X11 this is a complicated name, for example: >
   font=-misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
<
        The font-name "NONE" can be used to revert to the default font.
        When setting the font for the "Normal" group, this becomes the default
        font (until the 'guifont' option is changed; the last one set is
        used).
        The following only works with Motif and Athena, not with other GUIs:
        When setting the font for the "Menu" group, the menus will be changed.
        When setting the font for the "Tooltip" group, the tooltips will be
        changed.
        All fonts used, except for Menu and Tooltip, should be of the same
        character size as the default font!  Otherwise redrawing problems will
        occur.
        To use a font name with an embedded space or other special character,
        put it in single quotes.  The single quote cannot be used then.
        Example: >
            :hi comment font='Monospace 10'

guifg={color-name}                                      *highlight-guifg*
guibg={color-name}                                      *highlight-guibg*
guisp={color-name}                                      *highlight-guisp*
        These give the foreground (guifg), background (guibg) and special
        (guisp) color to use in the GUI.  "guisp" is used for undercurl and
        strikethrough.
        There are a few special names:
                NONE            no color (transparent)
                bg              use normal background color
                background      use normal background color
                fg              use normal foreground color
                foreground      use normal foreground color
        To use a color name with an embedded space or other special character,
        put it in single quotes.  The single quote cannot be used then.
        Example: >
            :hi comment guifg='salmon pink'
<
                                                        *gui-colors*
        Suggested color names (these are available on most systems):
            Red         LightRed        DarkRed
            Green       LightGreen      DarkGreen       SeaGreen
            Blue        LightBlue       DarkBlue        SlateBlue
            Cyan        LightCyan       DarkCyan
            Magenta     LightMagenta    DarkMagenta
            Yellow      LightYellow     Brown           DarkYellow
```

```
            Gray        LightGray       DarkGray
            Black       White
            Orange      Purple          Violet

        In the Win32 GUI version, additional system colors are available.  See
        |win32-colors|.

        You can also specify a color by its Red, Green and Blue values.
        The format is "#rrggbb", where
                "rr"    is the Red value
                "gg"    is the Green value
                "bb"    is the Blue value
        All values are hexadecimal, range from "00" to "ff".  Examples: >
  :highlight Comment guifg=#11f0c3 guibg=#ff00ff
<
                                        *highlight-groups* *highlight-default*
These are the default highlighting groups.  These groups are used by the
'highlight' option default.  Note that the highlighting depends on the value
of 'background'.  You can see the current settings with the ":highlight"
command.
                                                        *hl-ColorColumn*
ColorColumn     used for the columns set with 'colorcolumn'
                                                        *hl-Conceal*
Conceal         placeholder characters substituted for concealed
                text (see 'conceallevel')
                                                        *hl-Cursor*
Cursor          the character under the cursor
                                                        *hl-CursorIM*
CursorIM        like Cursor, but used when in IME mode |CursorIM|
                                                        *hl-CursorColumn*
CursorColumn    the screen column that the cursor is in when 'cursorcolumn' is
                set
                                                        *hl-CursorLine*
CursorLine      the screen line that the cursor is in when 'cursorline' is
                set
                                                        *hl-Directory*
Directory       directory names (and other special names in listings)
                                                        *hl-DiffAdd*
DiffAdd         diff mode: Added line |diff.txt|
                                                        *hl-DiffChange*
DiffChange      diff mode: Changed line |diff.txt|
                                                        *hl-DiffDelete*
DiffDelete      diff mode: Deleted line |diff.txt|
                                                        *hl-DiffText*
DiffText        diff mode: Changed text within a changed line |diff.txt|
                                                        *hl-EndOfBuffer*
EndOfBuffer     filler lines (~) after the last line in the buffer.
                By default, this is highlighted like |hl-NonText|.
                                                        *hl-ErrorMsg*
ErrorMsg        error messages on the command line
                                                        *hl-VertSplit*
VertSplit       the column separating vertically split windows
                                                        *hl-Folded*
Folded          line used for closed folds
                                                        *hl-FoldColumn*
FoldColumn      'foldcolumn'
                                                        *hl-SignColumn*
SignColumn      column where |signs| are displayed
                                                        *hl-IncSearch*
IncSearch       'incsearch' highlighting; also used for the text replaced with
                ":s///c"
                                                        *hl-LineNr*
```

```
LineNr          Line number for ":number" and ":#" commands, and when 'number'
                or 'relativenumber' option is set.
                                                        *hl-CursorLineNr*
CursorLineNr    Like LineNr when 'cursorline' or 'relativenumber' is set for
                the cursor line.
                                                        *hl-MatchParen*
MatchParen      The character under the cursor or just before it, if it
                is a paired bracket, and its match. |pi_paren.txt|

                                                        *hl-ModeMsg*
ModeMsg         'showmode' message (e.g., "-- INSERT --")
                                                        *hl-MoreMsg*
MoreMsg         |more-prompt|
                                                        *hl-NonText*
NonText         '@' at the end of the window, characters from 'showbreak'
                and other characters that do not really exist in the text
                (e.g., ">" displayed when a double-wide character doesn't
                fit at the end of the line).
                                                        *hl-Normal*
Normal          normal text
                                                        *hl-Pmenu*
Pmenu           Popup menu: normal item.
                                                        *hl-PmenuSel*
PmenuSel        Popup menu: selected item.
                                                        *hl-PmenuSbar*
PmenuSbar       Popup menu: scrollbar.
                                                        *hl-PmenuThumb*
PmenuThumb      Popup menu: Thumb of the scrollbar.
                                                        *hl-Question*
Question        |hit-enter| prompt and yes/no questions
                                                        *hl-QuickFixLine*
QuickFixLine    Current |quickfix| item in the quickfix window.
                                                        *hl-Search*
Search          Last search pattern highlighting (see 'hlsearch').
                Also used for similar items that need to stand out.
                                                        *hl-SpecialKey*
SpecialKey      Meta and special keys listed with ":map", also for text used
                to show unprintable characters in the text, 'listchars'.
                Generally: text that is displayed differently from what it
                really is.
                                                        *hl-SpellBad*
SpellBad        Word that is not recognized by the spellchecker. |spell|
                This will be combined with the highlighting used otherwise.
                                                        *hl-SpellCap*
SpellCap        Word that should start with a capital. |spell|
                This will be combined with the highlighting used otherwise.
                                                        *hl-SpellLocal*
SpellLocal      Word that is recognized by the spellchecker as one that is
                used in another region. |spell|
                This will be combined with the highlighting used otherwise.
                                                        *hl-SpellRare*
SpellRare       Word that is recognized by the spellchecker as one that is
                hardly ever used. |spell|
                This will be combined with the highlighting used otherwise.
                                                        *hl-StatusLine*
StatusLine      status line of current window
                                                        *hl-StatusLineNC*
StatusLineNC    status lines of not-current windows
                Note: if this is equal to "StatusLine" Vim will use "^^^" in
                the status line of the current window.
                                                        *hl-TabLine*
TabLine         tab pages line, not active tab page label
```

                                                        hl-TabLineFill
TabLineFill   tab pages line, where there are no labels
                                                        hl-TabLineSel
TabLineSel   tab pages line, active tab page label
                                                        hl-Title
Title   titles for output from ":set all", ":autocmd" etc.
                                                        hl-Visual
Visual   Visual mode selection
                                                        hl-VisualNOS
VisualNOS   Visual mode selection when vim is "Not Owning the Selection".
                Only X11 Gui's |gui-x11| and |xterm-clipboard| supports this.
                                                        hl-WarningMsg
WarningMsg   warning messages
                                                        hl-WildMenu
WildMenu   current match in 'wildmenu' completion

                                        hl-User1 *hl-User1..9* *hl-User9*
The 'statusline' syntax allows the use of 9 different highlights in the
statusline and ruler (via 'rulerformat'). The names are User1 to User9.

For the GUI you can use the following groups to set the colors for the menu,
scrollbars and tooltips. They don't have defaults. This doesn't work for the
Win32 GUI. Only three highlight arguments have any effect here: font, guibg,
and guifg.

                                                        hl-Menu
Menu   Current font, background and foreground colors of the menus.
                Also used for the toolbar.
                Applicable highlight arguments: font, guibg, guifg.

                NOTE: For Motif and Athena the font argument actually
                specifies a fontset at all times, no matter if 'guifontset' is
                empty, and as such it is tied to the current |:language| when
                set.

                                                        hl-Scrollbar
Scrollbar   Current background and foreground of the main window's
                scrollbars.
                Applicable highlight arguments: guibg, guifg.

                                                        hl-Tooltip
Tooltip   Current font, background and foreground of the tooltips.
                Applicable highlight arguments: font, guibg, guifg.

                NOTE: For Motif and Athena the font argument actually
                specifies a fontset at all times, no matter if 'guifontset' is
                empty, and as such it is tied to the current |:language| when
                set.

==
13. Linking groups   *:hi-link* *:highlight-link* *E412* *E413*

When you want to use the same highlighting for several syntax groups, you
can do this more easily by linking the groups into one common highlight
group, and give the color attributes only for that group.

To set a link:

    :hi[ghlight][!] [default] link {from-group} {to-group}

To remove a link:

```
    :hi[ghlight][!] [default] link {from-group} NONE

Notes:                                                  *E414*
- If the {from-group} and/or {to-group} doesn't exist, it is created.  You
  don't get an error message for a non-existing group.
- As soon as you use a ":highlight" command for a linked group, the link is
  removed.
- If there are already highlight settings for the {from-group}, the link is
  not made, unless the '!' is given.  For a ":highlight link" command in a
  sourced file, you don't get an error message.  This can be used to skip
  links for groups that already have settings.

                                        *:hi-default* *:highlight-default*
The [default] argument is used for setting the default highlighting for a
group.  If highlighting has already been specified for the group the command
will be ignored.  Also when there is an existing link.

Using [default] is especially useful to overrule the highlighting of a
specific syntax file.  For example, the C syntax file contains: >
        :highlight default link cComment Comment
If you like Question highlighting for C comments, put this in your vimrc file: >
        :highlight link cComment Question
Without the "default" in the C syntax file, the highlighting would be
overruled when the syntax file is loaded.

==============================================================================
14. Cleaning up                                         *:syn-clear* *E391*

If you want to clear the syntax stuff for the current buffer, you can use this
command: >
  :syntax clear

This command should be used when you want to switch off syntax highlighting,
or when you want to switch to using another syntax.  It's normally not needed
in a syntax file itself, because syntax is cleared by the autocommands that
load the syntax file.
The command also deletes the "b:current_syntax" variable, since no syntax is
loaded after this command.

If you want to disable syntax highlighting for all buffers, you need to remove
the autocommands that load the syntax files: >
  :syntax off

What this command actually does, is executing the command >
  :source $VIMRUNTIME/syntax/nosyntax.vim
See the "nosyntax.vim" file for details.  Note that for this to work
$VIMRUNTIME must be valid.  See |$VIMRUNTIME|.

To clean up specific syntax groups for the current buffer: >
  :syntax clear {group-name} ..
This removes all patterns and keywords for {group-name}.

To clean up specific syntax group lists for the current buffer: >
  :syntax clear @{grouplist-name} ..
This sets {grouplist-name}'s contents to an empty list.

                                              *:syntax-reset* *:syn-reset*
If you have changed the colors and messed them up, use this command to get the
defaults back: >

  :syntax reset
```

It is a bit of a wrong name, since it does not reset any syntax items, it only affects the highlighting.

This doesn't change the colors for the 'highlight' option.

Note that the syntax colors that you set in your vimrc file will also be reset back to their Vim default.
Note that if you are using a color scheme, the colors defined by the color scheme for syntax highlighting will be lost.

What this actually does is: >

```
        let g:syntax_cmd = "reset"
        runtime! syntax/syncolor.vim
```

Note that this uses the 'runtimepath' option.

syncolor
If you want to use different colors for syntax highlighting, you can add a Vim script file to set these colors. Put this file in a directory in 'runtimepath' which comes after $VIMRUNTIME, so that your settings overrule the default colors. This way these colors will be used after the ":syntax reset" command.

For Unix you can use the file ~/.vim/after/syntax/syncolor.vim. Example: >

```
        if &background == "light"
          highlight comment ctermfg=darkgreen guifg=darkgreen
        else
          highlight comment ctermfg=green guifg=green
        endif
```

E679
Do make sure this syncolor.vim script does not use a "syntax on", set the 'background' option or uses a "colorscheme" command, because it results in an endless loop.

Note that when a color scheme is used, there might be some confusion whether your defined colors are to be used or the colors from the scheme. This depends on the color scheme file. See |:colorscheme|.

syntax_cmd
The "syntax_cmd" variable is set to one of these values when the syntax/syncolor.vim files are loaded:

"on"	":syntax on" command. Highlight colors are overruled but links are kept
"enable"	":syntax enable" command. Only define colors for groups that don't have highlighting yet. Use ":syntax default".
"reset"	":syntax reset" command or loading a color scheme. Define all the colors.
"skip"	Don't define colors. Used to skip the default settings when a syncolor.vim file earlier in 'runtimepath' has already set them.

==
15. Highlighting tags *tag-highlight*

If you want to highlight all the tags in your file, you can use the following mappings.

```
        <F11>   -- Generate tags.vim file, and highlight tags.
        <F12>   -- Just highlight tags based on existing tags.vim file.
```

```
>
  :map <F11>  :sp tags<CR>:%s/^\([^     :]*:\)\=\([^    ]*\).*/syntax keyword Tag \2/
<CR>:wq! tags.vim<CR>/^<CR><F12>
  :map <F12>  :so tags.vim<CR>

WARNING: The longer the tags file, the slower this will be, and the more
memory Vim will consume.

Only highlighting typedefs, unions and structs can be done too.  For this you
must use Exuberant ctags (found at http://ctags.sf.net).

Put these lines in your Makefile:

# Make a highlight file for types.  Requires Exuberant ctags and awk
types: types.vim
types.vim: *.[ch]
        ctags --c-kinds=gstu -o- *.[ch] |\
                awk 'BEGIN{printf("syntax keyword Type\t")}\
                        {printf("%s ", $$1)}END{print ""}' > $@

And put these lines in your .vimrc: >

   " load the types.vim highlighting file, if it exists
   autocmd BufRead,BufNewFile *.[ch] let fname = expand('<afile>:p:h') . '/types.vim'
   autocmd BufRead,BufNewFile *.[ch] if filereadable(fname)
   autocmd BufRead,BufNewFile *.[ch]   exe 'so ' . fname
   autocmd BufRead,BufNewFile *.[ch] endif

==============================================================================
16. Window-local syntax                         *:ownsyntax*

Normally all windows on a buffer share the same syntax settings. It is
possible, however, to set a particular window on a file to have its own
private syntax setting. A possible example would be to edit LaTeX source
with conventional highlighting in one window, while seeing the same source
highlighted differently (so as to hide control sequences and indicate bold,
italic etc regions) in another. The 'scrollbind' option is useful here.

To set the current window to have the syntax "foo", separately from all other
windows on the buffer: >
   :ownsyntax foo
<                                               *w:current_syntax*
This will set the "w:current_syntax" variable to "foo".  The value of
"b:current_syntax" does not change.  This is implemented by saving and
restoring "b:current_syntax", since the syntax files do set
"b:current_syntax".  The value set by the syntax file is assigned to
"w:current_syntax".
Note: This resets the 'spell', 'spellcapcheck' and 'spellfile' options.

Once a window has its own syntax, syntax commands executed from other windows
on the same buffer (including :syntax clear) have no effect. Conversely,
syntax commands executed from that window do not affect other windows on the
same buffer.

A window with its own syntax reverts to normal behavior when another buffer
is loaded into that window or the file is reloaded.
When splitting the window, the new window will use the original syntax.

==============================================================================
17. Color xterms                                *xterm-color* *color-xterm*

Most color xterms have only eight colors.  If you don't get colors with the
```

```
default setup, it should work with these lines in your .vimrc: >
   :if &term =~ "xterm"
   :  if has("terminfo")
   :    set t_Co=8
   :    set t_Sf=<Esc>[3%p1%dm
   :    set t_Sb=<Esc>[4%p1%dm
   :  else
   :    set t_Co=8
   :    set t_Sf=<Esc>[3%dm
   :    set t_Sb=<Esc>[4%dm
   :  endif
   :endif
<       [<Esc> is a real escape, type CTRL-V <Esc>]

You might want to change the first "if" to match the name of your terminal,
e.g. "dtterm" instead of "xterm".

Note: Do these settings BEFORE doing ":syntax on".  Otherwise the colors may
be wrong.
                                                        *xiterm* *rxvt*
The above settings have been mentioned to work for xiterm and rxvt too.
But for using 16 colors in an rxvt these should work with terminfo: >
        :set t_AB=<Esc>[%?%p1%{8}%<%t25;%p1%{40}%+%e5;%p1%{32}%+%;%dm
        :set t_AF=<Esc>[%?%p1%{8}%<%t22;%p1%{30}%+%e1;%p1%{22}%+%;%dm
<
                                                        *colortest.vim*
To test your color setup, a file has been included in the Vim distribution.
To use it, execute this command: >
   :runtime syntax/colortest.vim

Some versions of xterm (and other terminals, like the Linux console) can
output lighter foreground colors, even though the number of colors is defined
at 8.  Therefore Vim sets the "cterm=bold" attribute for light foreground
colors, when 't_Co' is 8.

                                                        *xfree-xterm*
To get 16 colors or more, get the newest xterm version (which should be
included with XFree86 3.3 and later).  You can also find the latest version
at: >
        http://invisible-island.net/xterm/xterm.html
Here is a good way to configure it.  This uses 88 colors and enables the
termcap-query feature, which allows Vim to ask the xterm how many colors it
supports. >
        ./configure --disable-bold-color --enable-88-color --enable-tcap-query
If you only get 8 colors, check the xterm compilation settings.
(Also see |UTF8-xterm| for using this xterm with UTF-8 character encoding).

This xterm should work with these lines in your .vimrc (for 16 colors): >
   :if has("terminfo")
   :  set t_Co=16
   :  set t_AB=<Esc>[%?%p1%{8}%<%t%p1%{40}%+%e%p1%{92}%+%;%dm
   :  set t_AF=<Esc>[%?%p1%{8}%<%t%p1%{30}%+%e%p1%{82}%+%;%dm
   :else
   :  set t_Co=16
   :  set t_Sf=<Esc>[3%dm
   :  set t_Sb=<Esc>[4%dm
   :endif
<       [<Esc> is a real escape, type CTRL-V <Esc>]

Without |+terminfo|, Vim will recognize these settings, and automatically
translate cterm colors of 8 and above to "<Esc>[9%dm" and "<Esc>[10%dm".
Colors above 16 are also translated automatically.
```

```
For 256 colors this has been reported to work: >

   :set t_AB=<Esc>[48;5;%dm
   :set t_AF=<Esc>[38;5;%dm

Or just set the TERM environment variable to "xterm-color" or "xterm-16color"
and try if that works.

You probably want to use these X resources (in your ~/.Xdefaults file):
        XTerm*color0:                   #000000
        XTerm*color1:                   #c00000
        XTerm*color2:                   #008000
        XTerm*color3:                   #808000
        XTerm*color4:                   #0000c0
        XTerm*color5:                   #c000c0
        XTerm*color6:                   #008080
        XTerm*color7:                   #c0c0c0
        XTerm*color8:                   #808080
        XTerm*color9:                   #ff6060
        XTerm*color10:                  #00ff00
        XTerm*color11:                  #ffff00
        XTerm*color12:                  #8080ff
        XTerm*color13:                  #ff40ff
        XTerm*color14:                  #00ffff
        XTerm*color15:                  #ffffff
        Xterm*cursorColor:              Black

[Note: The cursorColor is required to work around a bug, which changes the
cursor color to the color of the last drawn text.  This has been fixed by a
newer version of xterm, but not everybody is using it yet.]

To get these right away, reload the .Xdefaults file to the X Option database
Manager (you only need to do this when you just changed the .Xdefaults file): >
  xrdb -merge ~/.Xdefaults
<
                                        *xterm-blink* *xterm-blinking-cursor*
To make the cursor blink in an xterm, see tools/blink.c.  Or use Thomas
Dickey's xterm above patchlevel 107 (see above for where to get it), with
these resources:
        XTerm*cursorBlink:      on
        XTerm*cursorOnTime:     400
        XTerm*cursorOffTime:    250
        XTerm*cursorColor:      White

                                                            *hpterm-color*
These settings work (more or less) for an hpterm, which only supports 8
foreground colors: >
   :if has("terminfo")
   :  set t_Co=8
   :  set t_Sf=<Esc>[&v%p1%dS
   :  set t_Sb=<Esc>[&v7S
   :else
   :  set t_Co=8
   :  set t_Sf=<Esc>[&v%dS
   :  set t_Sb=<Esc>[&v7S
   :endif
<       [<Esc> is a real escape, type CTRL-V <Esc>]

                                                *Eterm* *enlightened-terminal*
These settings have been reported to work for the Enlightened terminal
emulator, or Eterm.  They might work for all xterm-like terminals that use the
```

```
bold attribute to get bright colors.  Add an ":if" like above when needed. >
       :set t_Co=16
       :set t_AF=^[[%?%p1%{8}%<%t3%p1%d%e%p1%{22}%+%d;1%;m
       :set t_AB=^[[%?%p1%{8}%<%t4%p1%d%e%p1%{32}%+%d;1%;m
<
                                                    *TTpro-telnet*
These settings should work for TTpro telnet.  Tera Term Pro is a freeware /
open-source program for MS-Windows. >
        set t_Co=16
        set t_AB=^[[%?%p1%{8}%<%t%p1%{40}%+%e%p1%{32}%+5;%;%dm
        set t_AF=^[[%?%p1%{8}%<%t%p1%{30}%+%e%p1%{22}%+1;%;%dm
Also make sure TTpro's Setup / Window / Full Color is enabled, and make sure
that Setup / Font / Enable Bold is NOT enabled.
(info provided by John Love-Jensen <eljay@Adobe.COM>)


==============================================================================
18. When syntax is slow                                          *:syntime*

This is aimed at authors of a syntax file.

If your syntax causes redrawing to be slow, here are a few hints on making it
faster.  To see slowness switch on some features that usually interfere, such
as 'relativenumber' and |folding|.

Note: this is only available when compiled with the |+profile| feature.
You many need to build Vim with "huge" features.

To find out what patterns are consuming most time, get an overview with this
sequence: >
        :syntime on
        [ redraw the text at least once with CTRL-L ]
        :syntime report

This will display a list of syntax patterns that were used, sorted by the time
it took to match them against the text.

:syntime on             Start measuring syntax times.  This will add some
                        overhead to compute the time spent on syntax pattern
                        matching.

:syntime off            Stop measuring syntax times.

:syntime clear          Set all the counters to zero, restart measuring.

:syntime report         Show the syntax items used since ":syntime on" in the
                        current window.  Use a wider display to see more of
                        the output.

                        The list is sorted by total time. The columns are:
                        TOTAL           Total time in seconds spent on
                                        matching this pattern.
                        COUNT           Number of times the pattern was used.
                        MATCH           Number of times the pattern actually
                                        matched
                        SLOWEST         The longest time for one try.
                        AVERAGE         The average time for one try.
                        NAME            Name of the syntax item.  Note that
                                        this is not unique.
                        PATTERN         The pattern being used.

Pattern matching gets slow when it has to try many alternatives.  Try to
```

include as much literal text as possible to reduce the number of ways a
pattern does NOT match.

When using the "\@<=" and "\@<!" items, add a maximum size to avoid trying at
all positions in the current and previous line. For example, if the item is
literal text specify the size of that text (in bytes):

"<\@<=span"   Matches "span" in "<span". This tries matching with "<" in
                many places.
"<\@1<=span"   Matches the same, but only tries one byte before "span".

 vim:tw=78:sw=4:ts=8:ft=help:norl:
spell.txt   For Vim version 8.0. Last change: 2016 Jan 08

                  VIM REFERENCE MANUAL   by Bram Moolenaar

Spell checking   *spell*

1. Quick start |spell-quickstart|
2. Remarks on spell checking |spell-remarks|
3. Generating a spell file |spell-mkspell|
4. Spell file format |spell-file-format|

{Vi does not have any of these commands}

Spell checking is not available when the |+syntax| feature has been disabled
at compile time.

Note: There also is a vimspell plugin. If you have it you can do ":help
vimspell" to find about it. But you will probably want to get rid of the
plugin and use the 'spell' option instead, it works better.

==
1. Quick start   *spell-quickstart* *E756*

This command switches on spell checking: >

        :setlocal spell spelllang=en_us

This switches on the 'spell' option and specifies to check for US English.

The words that are not recognized are highlighted with one of these:
        SpellBad   word not recognized |hl-SpellBad|
        SpellCap   word not capitalised |hl-SpellCap|
        SpellRare   rare word |hl-SpellRare|
        SpellLocal   wrong spelling for selected region |hl-SpellLocal|

Vim only checks words for spelling, there is no grammar check.

If the 'mousemodel' option is set to "popup" and the cursor is on a badly
spelled word or it is "popup_setpos" and the mouse pointer is on a badly
spelled word, then the popup menu will contain a submenu to replace the bad
word. Note: this slows down the appearance of the popup menu. Note for GTK:
don't release the right mouse button until the menu appears, otherwise it
won't work.

To search for the next misspelled word:

                                                            ]s

```
]s                      Move to next misspelled word after the cursor.
                        A count before the command can be used to repeat.
                        'wrapscan' applies.

                                                        *[s*
[s                      Like "]s" but search backwards, find the misspelled
                        word before the cursor.  Doesn't recognize words
                        split over two lines, thus may stop at words that are
                        not highlighted as bad.  Does not stop at word with
                        missing capital at the start of a line.

                                                        *]S*
]S                      Like "]s" but only stop at bad words, not at rare
                        words or words for another region.

                                                        *[S*
[S                      Like "]S" but search backwards.


To add words to your own word list:

                                                        *zg*
zg                      Add word under the cursor as a good word to the first
                        name in 'spellfile'.  A count may precede the command
                        to indicate the entry in 'spellfile' to be used.  A
                        count of two uses the second entry.

                        In Visual mode the selected characters are added as a
                        word (including white space!).
                        When the cursor is on text that is marked as badly
                        spelled then the marked text is used.
                        Otherwise the word under the cursor, separated by
                        non-word characters, is used.

                        If the word is explicitly marked as bad word in
                        another spell file the result is unpredictable.

                                                        *zG*
zG                      Like "zg" but add the word to the internal word list
                        |internal-wordlist|.

                                                        *zw*
zw                      Like "zg" but mark the word as a wrong (bad) word.
                        If the word already appears in 'spellfile' it is
                        turned into a comment line.  See |spellfile-cleanup|
                        for getting rid of those.

                                                        *zW*
zW                      Like "zw" but add the word to the internal word list
                        |internal-wordlist|.

zuw                                                     *zug* *zuw*
zug                     Undo |zw| and |zg|, remove the word from the entry in
                        'spellfile'.  Count used as with |zg|.

zuW                                                     *zuG* *zuW*
zuG                     Undo |zW| and |zG|, remove the word from the internal
                        word list.  Count used as with |zg|.

                                                        *:spe* *:spellgood*
:[count]spe[llgood] {word}
                        Add {word} as a good word to 'spellfile', like with
```

|zg|. Without count the first name is used, with a
count of two the second entry, etc.

:spe[llgood]! {word} Add {word} as a good word to the internal word list,
like with |zG|.

:spellw *:spellwrong*
:[count]spellw[rong] {word}
Add {word} as a wrong (bad) word to 'spellfile', as
with |zw|. Without count the first name is used, with
a count of two the second entry, etc.

:spellw[rong]! {word} Add {word} as a wrong (bad) word to the internal word
list, like with |zW|.

:spellu *:spellundo*
:[count]spellu[ndo] {word}
Like |zuw|. [count] used as with |:spellgood|.

:spellu[ndo]! {word} Like |zuW|. [count] used as with |:spellgood|.

After adding a word to 'spellfile' with the above commands its associated
".spl" file will automatically be updated and reloaded. If you change
'spellfile' manually you need to use the |:mkspell| command. This sequence of
commands mostly works well: >
:edit <file in 'spellfile'>
< (make changes to the spell file) >
:mkspell! %

More details about the 'spellfile' format below |spell-wordlist-format|.

internal-wordlist
The internal word list is used for all buffers where 'spell' is set. It is
not stored, it is lost when you exit Vim. It is also cleared when 'encoding'
is set.

Finding suggestions for bad words:

z=
z= For the word under/after the cursor suggest correctly
spelled words. This also works to find alternatives
for a word that is not highlighted as a bad word,
e.g., when the word after it is bad.
In Visual mode the highlighted text is taken as the
word to be replaced.
The results are sorted on similarity to the word being
replaced.
This may take a long time. Hit CTRL-C when you get
bored.

If the command is used without a count the
alternatives are listed and you can enter the number
of your choice or press <Enter> if you don't want to
replace. You can also use the mouse to click on your
choice (only works if the mouse can be used in Normal
mode and when there are no line wraps). Click on the
first line (the header) to cancel.

The suggestions listed normally replace a highlighted
bad word. Sometimes they include other text, in that
case the replaced text is also listed after a "<".

If a count is used that suggestion is used, without prompting. For example, "1z=" always takes the first suggestion.

If 'verbose' is non-zero a score will be displayed with the suggestions to indicate the likeliness to the badly spelled word (the higher the score the more different).
When a word was replaced the redo command "." will repeat the word replacement. This works like "ciw", the good word and <Esc>. This does NOT work for Thai and other languages without spaces between words.

:spellr *:spellrepall* *E752* *E753*

:spellr[epall] Repeat the replacement done by |z=| for all matches with the replaced word in the current window.

In Insert mode, when the cursor is after a badly spelled word, you can use CTRL-X s to find suggestions. This works like Insert mode completion. Use CTRL-N to use the next suggestion, CTRL-P to go back. |i_CTRL-X_s|

The 'spellsuggest' option influences how the list of suggestions is generated and sorted. See |'spellsuggest'|.

The 'spellcapcheck' option is used to check the first word of a sentence starts with a capital. This doesn't work for the first word in the file. When there is a line break right after a sentence the highlighting of the next line may be postponed. Use |CTRL-L| when needed. Also see |set-spc-auto| for how it can be set automatically when 'spelllang' is set.

Vim counts the number of times a good word is encountered. This is used to sort the suggestions: words that have been seen before get a small bonus, words that have been seen often get a bigger bonus. The COMMON item in the affix file can be used to define common words, so that this mechanism also works in a new or short file |spell-COMMON|.

==
2. Remarks on spell checking *spell-remarks*

PERFORMANCE

Vim does on-the-fly spell checking. To make this work fast the word list is loaded in memory. Thus this uses a lot of memory (1 Mbyte or more). There might also be a noticeable delay when the word list is loaded, which happens when 'spell' is set and when 'spelllang' is set while 'spell' was already set. To minimize the delay each word list is only loaded once, it is not deleted when 'spelllang' is made empty or 'spell' is reset. When 'encoding' is set all the word lists are reloaded, thus you may notice a delay then too.

REGIONS

A word may be spelled differently in various regions. For example, English comes in (at least) these variants:

en	all regions
en_au	Australia
en_ca	Canada
en_gb	Great Britain
en_nz	New Zealand
en_us	USA

```
Words that are not used in one region but are used in another region are
highlighted with SpellLocal |hl-SpellLocal|.

Always use lowercase letters for the language and region names.

When adding a word with |zg| or another command it's always added for all
regions.  You can change that by manually editing the 'spellfile'.  See
|spell-wordlist-format|.  Note that the regions as specified in the files in
'spellfile' are only used when all entries in 'spelllang' specify the same
region (not counting files specified by their .spl name).

                                                        *spell-german*
Specific exception: For German these special regions are used:
        de              all German words accepted
        de_de           old and new spelling
        de_19           old spelling
        de_20           new spelling
        de_at           Austria
        de_ch           Switzerland

                                                        *spell-russian*
Specific exception: For Russian these special regions are used:
        ru              all Russian words accepted
        ru_ru           "IE" letter spelling
        ru_yo           "YO" letter spelling

                                                        *spell-yiddish*
Yiddish requires using "utf-8" encoding, because of the special characters
used.  If you are using latin1 Vim will use transliterated (romanized) Yiddish
instead.  If you want to use transliterated Yiddish with utf-8 use "yi-tr".
In a table:
        'encoding'      'spelllang'
        utf-8           yi              Yiddish
        latin1          yi              transliterated Yiddish
        utf-8           yi-tr           transliterated Yiddish

                                                        *spell-cjk*
Chinese, Japanese and other East Asian characters are normally marked as
errors, because spell checking of these characters is not supported. If
'spelllang' includes "cjk", these characters are not marked as errors.  This
is useful when editing text with spell checking while some Asian words are
present.


SPELL FILES                                             *spell-load*

Vim searches for spell files in the "spell" subdirectory of the directories in
'runtimepath'.  The name is: LL.EEE.spl, where:
        LL      the language name
        EEE     the value of 'encoding'

The value for "LL" comes from 'spelllang', but excludes the region name.
Examples:
        'spelllang'     LL ~
        en_us           en
        en-rare         en-rare
        medical_ca      medical

Only the first file is loaded, the one that is first in 'runtimepath'.  If
this succeeds then additionally files with the name LL.EEE.add.spl are loaded.
All the ones that are found are used.
```

```
If no spell file is found the |SpellFileMissing| autocommand event is
triggered.  This may trigger the |spellfile.vim| plugin to offer you
downloading the spell file.

Additionally, the files related to the names in 'spellfile' are loaded.  These
are the files that |zg| and |zw| add good and wrong words to.

Exceptions:
- Vim uses "latin1" when 'encoding' is "iso-8859-15".  The euro sign doesn't
  matter for spelling.
- When no spell file for 'encoding' is found "ascii" is tried.  This only
  works for languages where nearly all words are ASCII, such as English.  It
  helps when 'encoding' is not "latin1", such as iso-8859-2, and English text
  is being edited.  For the ".add" files the same name as the found main
  spell file is used.

For example, with these values:
        'runtimepath' is "~/.vim,/usr/share/vim70,~/.vim/after"
        'encoding'    is "iso-8859-2"
        'spelllang'   is "pl"

Vim will look for:
1. ~/.vim/spell/pl.iso-8859-2.spl
2. /usr/share/vim70/spell/pl.iso-8859-2.spl
3. ~/.vim/spell/pl.iso-8859-2.add.spl
4. /usr/share/vim70/spell/pl.iso-8859-2.add.spl
5. ~/.vim/after/spell/pl.iso-8859-2.add.spl

This assumes 1. is not found and 2. is found.

If 'encoding' is "latin1" Vim will look for:
1. ~/.vim/spell/pl.latin1.spl
2. /usr/share/vim70/spell/pl.latin1.spl
3. ~/.vim/after/spell/pl.latin1.spl
4. ~/.vim/spell/pl.ascii.spl
5. /usr/share/vim70/spell/pl.ascii.spl
6. ~/.vim/after/spell/pl.ascii.spl

This assumes none of them are found (Polish doesn't make sense when leaving
out the non-ASCII characters).

Spelling for EBCDIC is currently not supported.

A spell file might not be available in the current 'encoding'.  See
|spell-mkspell| about how to create a spell file.  Converting a spell file
with "iconv" will NOT work!

Note: on VMS ".{enc}.spl" is changed to "_{enc}.spl" to avoid trouble with
filenames.

                                                    *spell-sug-file* *E781*
If there is a file with exactly the same name as the ".spl" file but ending in
".sug", that file will be used for giving better suggestions.  It isn't loaded
before suggestions are made to reduce memory use.

                                    *E758* *E759* *E778* *E779* *E780* *E782*
When loading a spell file Vim checks that it is properly formatted.  If you
get an error the file may be truncated, modified or intended for another Vim
version.


SPELLFILE CLEANUP                                       *spellfile-cleanup*
```

The |zw| command turns existing entries in 'spellfile' into comment lines.
This avoids having to write a new file every time, but results in the file
only getting longer, never shorter. To clean up the comment lines in all
".add" spell files do this: >
	:runtime spell/cleanadd.vim

This deletes all comment lines, except the ones that start with "##". Use
"##" lines to add comments that you want to keep.

You can invoke this script as often as you like. A variable is provided to
skip updating files that have been changed recently. Set it to the number of
seconds that has passed since a file was changed before it will be cleaned.
For example, to clean only files that were not changed in the last hour: >
	let g:spell_clean_limit = 60 * 60
The default is one second.

WORDS

Vim uses a fixed method to recognize a word. This is independent of
'iskeyword', so that it also works in help files and for languages that
include characters like '-' in 'iskeyword'. The word characters do depend on
'encoding'.

The table with word characters is stored in the main .spl file. Therefore it
matters what the current locale is when generating it! A .add.spl file does
not contain a word table though.

For a word that starts with a digit the digit is ignored, unless the word as a
whole is recognized. Thus if "3D" is a word and "D" is not then "3D" is
recognized as a word, but if "3D" is not a word then only the "D" is marked as
bad. Hex numbers in the form 0x12ab and 0X12AB are recognized.

WORD COMBINATIONS

It is possible to spell-check words that include a space. This is used to
recognize words that are invalid when used by themselves, e.g. for "et al.".
It can also be used to recognize "the the" and highlight it.

The number of spaces is irrelevant. In most cases a line break may also
appear. However, this makes it difficult to find out where to start checking
for spelling mistakes. When you make a change to one line and only that line
is redrawn Vim won't look in the previous line, thus when "et" is at the end
of the previous line "al." will be flagged as an error. And when you type
"the<CR>the" the highlighting doesn't appear until the first line is redrawn.
Use |CTRL-L| to redraw right away. "[s" will also stop at a word combination
with a line break.

When encountering a line break Vim skips characters such as '*', '>' and '"',
so that comments in C, shell and Vim code can be spell checked.

SYNTAX HIGHLIGHTING *spell-syntax*

Files that use syntax highlighting can specify where spell checking should be
done:

1. everywhere default
2. in specific items use "contains=@Spell"
3. everywhere but specific items use "contains=@NoSpell"

For the second method adding the @NoSpell cluster will disable spell checking
again. This can be used, for example, to add @Spell to the comments of a
program, and add @NoSpell for items that shouldn't be checked.
Also see |:syn-spell| for text that is not in a syntax item.

VIM SCRIPTS

If you want to write a Vim script that does something with spelling, you may
find these functions useful:

    spellbadword()   find badly spelled word at the cursor
    spellsuggest()   get list of spelling suggestions
    soundfold()   get the sound-a-like version of a word

SETTING 'spellcapcheck' AUTOMATICALLY *set-spc-auto*

After the 'spelllang' option has been set successfully, Vim will source the
files "spell/LANG.vim" in 'runtimepath'. "LANG" is the value of 'spelllang'
up to the first comma, dot or underscore. This can be used to set options
specifically for the language, especially 'spellcapcheck'.

The distribution includes a few of these files. Use this command to see what
they do: >
        :next $VIMRUNTIME/spell/*.vim

Note that the default scripts don't set 'spellcapcheck' if it was changed from
the default value. This assumes the user prefers another value then.

DOUBLE SCORING *spell-double-scoring*

The 'spellsuggest' option can be used to select "double" scoring. This
mechanism is based on the principle that there are two kinds of spelling
mistakes:

1. You know how to spell the word, but mistype something. This results in a
   small editing distance (character swapped/omitted/inserted) and possibly a
   word that sounds completely different.

2. You don't know how to spell the word and type something that sounds right.
   The edit distance can be big but the word is similar after sound-folding.

Since scores for these two mistakes will be very different we use a list
for each and mix them.

The sound-folding is slow and people that know the language won't make the
second kind of mistakes. Therefore 'spellsuggest' can be set to select the
preferred method for scoring the suggestions.

==
3. Generating a spell file *spell-mkspell*

Vim uses a binary file format for spelling. This greatly speeds up loading
the word list and keeps it small.
.aff *.dic* *Myspell*
You can create a Vim spell file from the .aff and .dic files that Myspell
uses. Myspell is used by OpenOffice.org and Mozilla. The OpenOffice .oxt
files are zip files which contain the .aff and .dic files. You should be able
to find them here:

```
        http://extensions.services.openoffice.org/dictionary
The older, OpenOffice 2 files may be used if this doesn't work:
        http://wiki.services.openoffice.org/wiki/Dictionaries
You can also use a plain word list.  The results are the same, the choice
depends on what word lists you can find.

If you install Aap (from www.a-a-p.org) you can use the recipes in the
runtime/spell/??/ directories.  Aap will take care of downloading the files,
apply patches needed for Vim and build the .spl file.

Make sure your current locale is set properly, otherwise Vim doesn't know what
characters are upper/lower case letters.  If the locale isn't available (e.g.,
when using an MS-Windows codepage on Unix) add tables to the .aff file
|spell-affix-chars|.  If the .aff file doesn't define a table then the word
table of the currently active spelling is used.  If spelling is not active
then Vim will try to guess.

                                                        *:mksp* *:mkspell*
:mksp[ell][!] [-ascii] {outname} {inname} ...
                        Generate a Vim spell file from word lists.  Example: >
                :mkspell /tmp/nl nl_NL.words
<                                                               *E751*
                        When {outname} ends in ".spl" it is used as the output
                        file name.  Otherwise it should be a language name,
                        such as "en", without the region name.  The file
                        written will be "{outname}.{encoding}.spl", where
                        {encoding} is the value of the 'encoding' option.

                        When the output file already exists [!] must be used
                        to overwrite it.

                        When the [-ascii] argument is present, words with
                        non-ascii characters are skipped.  The resulting file
                        ends in "ascii.spl".

                        The input can be the Myspell format files {inname}.aff
                        and {inname}.dic.  If {inname}.aff does not exist then
                        {inname} is used as the file name of a plain word
                        list.

                        Multiple {inname} arguments can be given to combine
                        regions into one Vim spell file.  Example: >
                :mkspell ~/.vim/spell/en /tmp/en_US /tmp/en_CA /tmp/en_AU
<                       This combines the English word lists for US, CA and AU
                        into one en.spl file.
                        Up to eight regions can be combined. *E754* *E755*
                        The REP and SAL items of the first .aff file where
                        they appear are used. |spell-REP| |spell-SAL|
                                                                *E845*
                        This command uses a lot of memory, required to find
                        the optimal word tree (Polish, Italian and Hungarian
                        require several hundred Mbyte).  The final result will
                        be much smaller, because compression is used.  To
                        avoid running out of memory compression will be done
                        now and then.  This can be tuned with the 'mkspellmem'
                        option.

                        After the spell file was written and it was being used
                        in a buffer it will be reloaded automatically.

:mksp[ell] [-ascii] {name}.{enc}.add
                        Like ":mkspell" above, using {name}.{enc}.add as the
```

input file and producing an output file in the same
directory that has ".spl" appended.

:mksp[ell] [-ascii] {name}
Like ":mkspell" above, using {name} as the input file
and producing an output file in the same directory
that has ".{enc}.spl" appended.

Vim will report the number of duplicate words. This might be a mistake in the list of words. But sometimes it is used to have different prefixes and suffixes for the same basic word to avoid them combining (e.g. Czech uses this). If you want Vim to report all duplicate words set the 'verbose' option.

Since you might want to change a Myspell word list for use with Vim the following procedure is recommended:

1. Obtain the xx_YY.aff and xx_YY.dic files from Myspell.
2. Make a copy of these files to xx_YY.orig.aff and xx_YY.orig.dic.
3. Change the xx_YY.aff and xx_YY.dic files to remove bad words, add missing words, define word characters with FOL/LOW/UPP, etc. The distributed "*.diff" files can be used.
4. Start Vim with the right locale and use |:mkspell| to generate the Vim spell file.
5. Try out the spell file with ":set spell spelllang=xx" if you wrote it in a spell directory in 'runtimepath', or ":set spelllang=xx.enc.spl" if you wrote it somewhere else.

When the Myspell files are updated you can merge the differences:
1. Obtain the new Myspell files as xx_YY.new.aff and xx_UU.new.dic.
2. Use Vimdiff to see what changed: >
vimdiff xx_YY.orig.dic xx_YY.new.dic
3. Take over the changes you like in xx_YY.dic.
You may also need to change xx_YY.aff.
4. Rename xx_YY.new.dic to xx_YY.orig.dic and xx_YY.new.aff to xx_YY.new.aff.

SPELL FILE VERSIONS *E770* *E771* *E772*

Spell checking is a relatively new feature in Vim, thus it's possible that the .spl file format will be changed to support more languages. Vim will check the validity of the spell file and report anything wrong.

E771: Old spell file, needs to be updated ~
This spell file is older than your Vim. You need to update the .spl file.

E772: Spell file is for newer version of Vim ~
This means the spell file was made for a later version of Vim. You need to update Vim.

E770: Unsupported section in spell file ~
This means the spell file was made for a later version of Vim and contains a section that is required for the spell file to work. In this case it's probably a good idea to upgrade your Vim.

SPELL FILE DUMP

If for some reason you want to check what words are supported by the currently used spelling files, use this command:

:spelldump *:spelld*

```
:spelld[ump]            Open a new window and fill it with all currently valid
                        words.  Compound words are not included.
                        Note: For some languages the result may be enormous,
                        causing Vim to run out of memory.

:spelld[ump]!           Like ":spelldump" and include the word count.  This is
                        the number of times the word was found while
                        updating the screen.  Words that are in COMMON items
                        get a starting count of 10.

The format of the word list is used |spell-wordlist-format|.  You should be
able to read it with ":mkspell" to generate one .spl file that includes all
the words.

When all entries to 'spelllang' use the same regions or no regions at all then
the region information is included in the dumped words.  Otherwise only words
for the current region are included and no "/regions" line is generated.

Comment lines with the name of the .spl file are used as a header above the
words that were generated from that .spl file.


SPELL FILE MISSING              *spell-SpellFileMissing* *spellfile.vim*

If the spell file for the language you are using is not available, you will
get an error message.  But if the "spellfile.vim" plugin is active it will
offer you to download the spell file.  Just follow the instructions, it will
ask you where to write the file (there must be a writable directory in
'runtimepath' for this).

The plugin has a default place where to look for spell files, on the Vim ftp
server.  If you want to use another location or another protocol, set the
g:spellfile_URL variable to the directory that holds the spell files.  The
|netrw| plugin is used for getting the file, look there for the specific
syntax of the URL.  Example: >
        let g:spellfile_URL = 'http://ftp.vim.org/vim/runtime/spell'
You may need to escape special characters.

The plugin will only ask about downloading a language once.  If you want to
try again anyway restart Vim, or set g:spellfile_URL to another value (e.g.,
prepend a space).

To avoid using the "spellfile.vim" plugin do this in your vimrc file: >

        let loaded_spellfile_plugin = 1

Instead of using the plugin you can define a |SpellFileMissing| autocommand to
handle the missing file yourself.  You can use it like this: >

        :au SpellFileMissing * call Download_spell_file(expand('<amatch>'))

Thus the <amatch> item contains the name of the language.  Another important
value is 'encoding', since every encoding has its own spell file.  With two
exceptions:
- For ISO-8859-15 (latin9) the name "latin1" is used (the encodings only
  differ in characters not used in dictionary words).
- The name "ascii" may also be used for some languages where the words use
  only ASCII letters for most of the words.

The default "spellfile.vim" plugin uses this autocommand, if you define your
autocommand afterwards you may want to use ":au! SpellFileMissing" to overrule
it.  If you define your autocommand before the plugin is loaded it will notice
```

this and not do anything.
E797
Note that the SpellFileMissing autocommand must not change or destroy the buffer the user was editing.

==
4. Spell file format *spell-file-format*

This is the format of the files that are used by the person who creates and maintains a word list.

Note that we avoid the word "dictionary" here. That is because the goal of spell checking differs from writing a dictionary (as in the book). For spelling we need a list of words that are OK, thus should not be highlighted. Person and company names will not appear in a dictionary, but do appear in a word list. And some old words are rarely used while they are common misspellings. These do appear in a dictionary but not in a word list.

There are two formats: A straight list of words and a list using affix compression. The files with affix compression are used by Myspell (Mozilla and OpenOffice.org). This requires two files, one with .aff and one with .dic extension.

FORMAT OF STRAIGHT WORD LIST *spell-wordlist-format*

The words must appear one per line. That is all that is required.

Additionally the following items are recognized:

- Empty and blank lines are ignored.

```
# comment ~
```
- Lines starting with a # are ignored (comment lines).

```
/encoding=utf-8 ~
```
- A line starting with "/encoding=", before any word, specifies the encoding of the file. After the second '=' comes an encoding name. This tells Vim to setup conversion from the specified encoding to 'encoding'. Thus you can use one word list for several target encodings.

```
/regions=usca ~
```
- A line starting with "/regions=" specifies the region names that are supported. Each region name must be two ASCII letters. The first one is region 1. Thus "/regions=usca" has region 1 "us" and region 2 "ca". In an addition word list the region names should be equal to the main word list!

- Other lines starting with '/' are reserved for future use. The ones that are not recognized are ignored. You do get a warning message, so that you know something won't work.

- A "/" may follow the word with the following items:
  - `=` Case must match exactly.
  - `?` Rare word.
  - `!` Bad (wrong) word.
  - `digit` A region in which the word is valid. If no regions are specified the word is valid in all regions.

Example:

```
# This is an example word list          comment
```

```
        /encoding=latin1                        encoding of the file
        /regions=uscagb                         regions "us", "ca" and "gb"
        example                                 word for all regions
        blah/12                                 word for regions "us" and "ca"
        vim/!                                   bad word
        Campbell/?3                             rare word in region 3 "gb"
        's mornings/=                           keep-case word
```

Note that when "/=" is used the same word with all upper-case letters is not accepted. This is different from a word with mixed case that is automatically marked as keep-case, those words may appear in all upper-case letters.

FORMAT WITH .AFF AND .DIC FILES *aff-dic-format*

There are two files: the basic word list and an affix file. The affix file specifies settings for the language and can contain affixes. The affixes are used to modify the basic words to get the full word list. This significantly reduces the number of words, especially for a language like Polish. This is called affix compression.

The basic word list and the affix file are combined with the ":mkspell" command and results in a binary spell file. All the preprocessing has been done, thus this file loads fast. The binary spell file format is described in the source code (src/spell.c). But only developers need to know about it.

The preprocessing also allows us to take the Myspell language files and modify them before the Vim word list is made. The tools for this can be found in the "src/spell" directory.

The format for the affix and word list files is based on what Myspell uses (the spell checker of Mozilla and OpenOffice.org). A description can be found here:

```
        http://lingucomponent.openoffice.org/affix.readme ~
```

Note that affixes are case sensitive, this isn't obvious from the description.

Vim supports quite a few extras. They are described below |spell-affix-vim|. Attempts have been made to keep this compatible with other spell checkers, so that the same files can often be used. One other project that offers more than Myspell is Hunspell (http://hunspell.sf.net).

WORD LIST FORMAT *spell-dic-format*

A short example, with line numbers:

```
        1       1234 ~
        2       aan ~
        3       Als ~
        4       Etten-Leur ~
        5       et al. ~
        6       's-Gravenhage ~
        7       's-Gravenhaags ~
        8       # word that differs between regions ~
        9       kado/1 ~
        10      cadeau/2 ~
        11      TCP,IP ~
        12      /the S affix may add a 's' ~
        13      bedel/S ~
```

The first line contains the number of words. Vim ignores it, but you do get an error message if it's not there. *E760*

What follows is one word per line. White space at the end of the line is ignored, all other white space matters. The encoding is specified in the affix file |spell-SET|.

Comment lines start with '#' or '/'. See the example lines 8 and 12. Note that putting a comment after a word is NOT allowed:

```
someword   # comment that causes an error! ~
```

After the word there is an optional slash and flags. Most of these flags are letters that indicate the affixes that can be used with this word. These are specified with SFX and PFX lines in the .aff file, see |spell-SFX| and |spell-PFX|. Vim allows using other flag types with the FLAG item in the affix file |spell-FLAG|.

When the word only has lower-case letters it will also match with the word starting with an upper-case letter.

When the word includes an upper-case letter, this means the upper-case letter is required at this position. The same word with a lower-case letter at this position will not match. When some of the other letters are upper-case it will not match either.

The word with all upper-case characters will always be OK,

word list	matches	does not match ~
als	als Als ALS	ALs AlS aLs aLS
Als	Als ALS	als ALs AlS aLs aLS
ALS	ALS	als Als ALs AlS aLs aLS
AlS	AlS ALS	als Als ALs aLs aLS

The KEEPCASE affix ID can be used to specifically match a word with identical case only, see below |spell-KEEPCASE|.

Note: in line 5 to 7 non-word characters are used. You can include any character in a word. When checking the text a word still only matches when it appears with a non-word character before and after it. For Myspell a word starting with a non-word character probably won't work.

In line 12 the word "TCP/IP" is defined. Since the slash has a special meaning the comma is used instead. This is defined with the SLASH item in the affix file, see |spell-SLASH|. Note that without this SLASH item the word will be "TCP,IP".

AFFIX FILE FORMAT *spell-aff-format* *spell-affix-vim*

spell-affix-comment

Comment lines in the .aff file start with a '#':

```
# comment line ~
```

Items with a fixed number of arguments can be followed by a comment. But only if none of the arguments can contain white space. The comment must start with a "#" character. Example:

```
KEEPCASE =  # fix case for words with this flag ~
```

ENCODING *spell-SET*

The affix file can be in any encoding that is supported by "iconv". However, in some cases the current locale should also be set properly at the time |:mkspell| is invoked. Adding FOL/LOW/UPP lines removes this requirement |spell-FOL|.

The encoding should be specified before anything where the encoding matters. The encoding applies both to the affix file and the dictionary file. It is done with a SET line:

```
SET utf-8 ~
```

The encoding can be different from the value of the 'encoding' option at the time ":mkspell" is used. Vim will then convert everything to 'encoding' and generate a spell file for 'encoding'. If some of the used characters to not fit in 'encoding' you will get an error message.

spell-affix-mbyte

When using a multi-byte encoding it's possible to use more different affix flags. But Myspell doesn't support that, thus you may not want to use it anyway. For compatibility use an 8-bit encoding.

INFORMATION

These entries in the affix file can be used to add information to the spell file. There are no restrictions on the format, but they should be in the right encoding.

spell-NAME *spell-VERSION* *spell-HOME*
spell-AUTHOR *spell-EMAIL* *spell-COPYRIGHT*

```
NAME            Name of the language
VERSION         1.0.1  with fixes
HOME            http://www.myhome.eu
AUTHOR          John Doe
EMAIL           john AT Doe DOT net
COPYRIGHT       LGPL
```

These fields are put in the .spl file as-is. The |:spellinfo| command can be used to view the info.

:spellinfo *:spelli*

```
:spelli[nfo]            Display the information for the spell file(s) used for
                        the current buffer.
```

CHARACTER TABLES

spell-affix-chars

When using an 8-bit encoding the affix file should define what characters are word characters. This is because the system where ":mkspell" is used may not support a locale with this encoding and isalpha() won't work. For example when using "cp1250" on Unix.

E761 *E762* *spell-FOL*
spell-LOW *spell-UPP*

Three lines in the affix file are needed. Simplistic example:

```
FOL  \E1\EB\F1 ~
LOW  \E1\EB\F1 ~
UPP  \C1\CB\D1 ~
```

All three lines must have exactly the same number of characters.

The "FOL" line specifies the case-folded characters. These are used to compare words while ignoring case. For most encodings this is identical to

the lower case line.

The "LOW" line specifies the characters in lower-case. Mostly it's equal to the "FOL" line.

The "UPP" line specifies the characters with upper-case. That is, a character is upper-case where it's different from the character at the same position in "FOL".

An exception is made for the German sharp s **\DF**. The upper-case version is "SS". In the FOL/LOW/UPP lines it should be included, so that it's recognized as a word character, but use the **\DF** character in all three.

ASCII characters should be omitted, Vim always handles these in the same way. When the encoding is UTF-8 no word characters need to be specified.

E763

Vim allows you to use spell checking for several languages in the same file. You can list them in the 'spelllang' option. As a consequence all spell files for the same encoding must use the same word characters, otherwise they can't be combined without errors.

If you get an E763 warning that the word tables differ you need to update your ".spl" spell files. If you downloaded the files, get the latest version of all spell files you use. If you are only using one, e.g., German, then also download the recent English spell files. Otherwise generate the .spl file again with |:mkspell|. If you still get errors check the FOL, LOW and UPP lines in the used .aff files.

The XX.ascii.spl spell file generated with the "-ascii" argument will not contain the table with characters, so that it can be combine with spell files for any encoding. The .add.spl files also do not contain the table.

MID-WORD CHARACTERS

spell-midword

Some characters are only to be considered word characters if they are used in between two ordinary word characters. An example is the single quote: It is often used to put text in quotes, thus it can't be recognized as a word character, but when it appears in between word characters it must be part of the word. This is needed to detect a spelling error such as they'are. That should be they're, but since "they" and "are" are words themselves that would go unnoticed.

These characters are defined with MIDWORD in the .aff file. Example:

```
MIDWORD '- ~
```

FLAG TYPES

spell-FLAG

Flags are used to specify the affixes that can be used with a word and for other properties of the word. Normally single-character flags are used. This limits the number of possible flags, especially for 8-bit encodings. The FLAG item can be used if more affixes are to be used. Possible values:

```
FLAG long       use two-character flags
FLAG num        use numbers, from 1 up to 65000
FLAG caplong    use one-character flags without A-Z and two-character
                flags that start with A-Z
```

With "FLAG num" the numbers in a list of affixes need to be separated with a

comma: "234,2143,1435". This method is inefficient, but useful if the file is generated with a program.

When using "caplong" the two-character flags all start with a capital: "Aa", "B1", "BB", etc. This is useful to use one-character flags for the most common items and two-character flags for uncommon items.

Note: When using utf-8 only characters up to 65000 may be used for flags.

Note: even when using "num" or "long" the number of flags available to compounding and prefixes is limited to about 250.

AFFIXES

spell-PFX *spell-SFX*

The usual PFX (prefix) and SFX (suffix) lines are supported (see the Myspell documentation or the Aspell manual: http://aspell.net/man-html/Affix-Compression.html).

Summary:

```
SFX L Y 2 ~
SFX L 0 re [^x] ~
SFX L 0 ro x ~
```

The first line is a header and has four fields:

```
SFX {flag} {combine} {count}
```

{flag} The name used for the suffix. Mostly it's a single letter, but other characters can be used, see |spell-FLAG|.

{combine} Can be 'Y' or 'N'. When 'Y' then the word plus suffix can also have a prefix. When 'N' then a prefix is not allowed.

{count} The number of lines following. If this is wrong you will get an error message.

For PFX the fields are exactly the same.

The basic format for the following lines is:

```
SFX {flag} {strip} {add} {condition} {extra}
```

{flag} Must be the same as the {flag} used in the first line.

{strip} Characters removed from the basic word. There is no check if the characters are actually there, only the length is used (in bytes). This better match the {condition}, otherwise strange things may happen. If the {strip} length is equal to or longer than the basic word the suffix won't be used. When {strip} is 0 (zero) then nothing is stripped.

{add} Characters added to the basic word, after removing {strip}. Optionally there is a '/' followed by flags. The flags apply to the word plus affix. See |spell-affix-flags|

{condition} A simplistic pattern. Only when this matches with a basic word will the suffix be used for that word. This is normally for using one suffix letter with different {add} and {strip} fields for words with different endings. When {condition} is a . (dot) there is no condition. The pattern may contain:
- Literal characters.
- A set of characters in []. [abc] matches a, b and c.

```
                  A dash is allowed for a range [a-c], but this is
                  Vim-specific.
                - A set of characters that starts with a ^, meaning the
                  complement of the specified characters. [^abc] matches any
                  character but a, b and c.

{extra}         Optional extra text:
                    # comment           Comment is ignored
                    -                   Hunspell uses this, ignored
```

For PFX the fields are the same, but the {strip}, {add} and {condition} apply to the start of the word.

Note: Myspell ignores any extra text after the relevant info. Vim requires this text to start with a "#" so that mistakes don't go unnoticed. Example:

```
        SFX F 0 in   [^i]n      # Spion > Spionin  ~
        SFX F 0 nen  in         # Bauerin > Bauerinnen ~
```

However, to avoid lots of errors in affix files written for Myspell, you can add the IGNOREEXTRA flag.

Apparently Myspell allows an affix name to appear more than once. Since this might also be a mistake, Vim checks for an extra "S". The affix files for Myspell that use this feature apparently have this flag. Example:

```
        SFX a Y 1 S ~
        SFX a 0 an . ~

        SFX a Y 2 S ~
        SFX a 0 en . ~
        SFX a 0 on . ~
```

AFFIX FLAGS *spell-affix-flags*

This is a feature that comes from Hunspell: The affix may specify flags. This works similar to flags specified on a basic word. The flags apply to the basic word plus the affix (but there are restrictions). Example:

```
        SFX S Y 1 ~
        SFX S 0 s . ~

        SFX A Y 1 ~
        SFX A 0 able/S . ~
```

When the dictionary file contains "drink/AS" then these words are possible:

```
        drink
        drinks          uses S suffix
        drinkable       uses A suffix
        drinkables      uses A suffix and then S suffix
```

Generally the flags of the suffix are added to the flags of the basic word, both are used for the word plus suffix. But the flags of the basic word are only used once for affixes, except that both one prefix and one suffix can be used when both support combining.

Specifically, the affix flags can be used for:
- Suffixes on suffixes, as in the example above. This works once, thus you can have two suffixes on a word (plus one prefix).
- Making the word with the affix rare, by using the |spell-RARE| flag.

- Exclude the word with the affix from compounding, by using the
  |spell-COMPOUNDFORBIDFLAG| flag.
- Allow the word with the affix to be part of a compound word on the side of
  the affix with the |spell-COMPOUNDPERMITFLAG|.
- Use the NEEDCOMPOUND flag: word plus affix can only be used as part of a
  compound word. |spell-NEEDCOMPOUND|
- Compound flags: word plus affix can be part of a compound word at the end,
  middle, start, etc. The flags are combined with the flags of the basic
  word. |spell-compound|
- NEEDAFFIX: another affix is needed to make a valid word.
- CIRCUMFIX, as explained just below.

IGNOREEXTRA *spell-IGNOREEXTRA*

Normally Vim gives an error for an extra field that does not start with '#'.
This avoids errors going unnoticed. However, some files created for Myspell
or Hunspell may contain many entries with an extra field. Use the IGNOREEXTRA
flag to avoid lots of errors.

CIRCUMFIX *spell-CIRCUMFIX*

The CIRCUMFIX flag means a prefix and suffix must be added at the same time.
If a prefix has the CIRCUMFIX flag than only suffixes with the CIRCUMFIX flag
can be added, and the other way around.
An alternative is to only specify the suffix, and give the that suffix two
flags: The required prefix and the NEEDAFFIX flag. |spell-NEEDAFFIX|

PFXPOSTPONE *spell-PFXPOSTPONE*

When an affix file has very many prefixes that apply to many words it's not
possible to build the whole word list in memory. This applies to Hebrew (a
list with all words is over a Gbyte). In that case applying prefixes must be
postponed. This makes spell checking slower. It is indicated by this keyword
in the .aff file:

```
PFXPOSTPONE ~
```

Only prefixes without a chop string and without flags can be postponed.
Prefixes with a chop string or with flags will still be included in the word
list. An exception if the chop string is one character and equal to the last
character of the added string, but in lower case. Thus when the chop string
is used to allow the following word to start with an upper case letter.

WORDS WITH A SLASH *spell-SLASH*

The slash is used in the .dic file to separate the basic word from the affix
letters and other flags. Unfortunately, this means you cannot use a slash in
a word. Thus "TCP/IP" is not a word but "TCP" with the flags "IP". To include
a slash in the word put a backslash before it: "TCP\/IP". In the rare case
you want to use a backslash inside a word you need to use two backslashes.
Any other use of the backslash is reserved for future expansion.

KEEP-CASE WORDS *spell-KEEPCASE*

In the affix file a KEEPCASE line can be used to define the affix name used
for keep-case words. Example:

```
        KEEPCASE = ~
```

This flag is not supported by Myspell. It has the meaning that case matters.
This can be used if the word does not have the first letter in upper case at
the start of a sentence. Example:

```
    word list       matches                 does not match ~
    's morgens/=    's morgens              'S morgens 's Morgens 'S MORGENS
    's Morgens      's Morgens 'S MORGENS   'S morgens 's morgens
```

The flag can also be used to avoid that the word matches when it is in all
upper-case letters.

RARE WORDS *spell-RARE*

In the affix file a RARE line can be used to define the affix name used for
rare words. Example:

```
        RARE ? ~
```

Rare words are highlighted differently from bad words. This is to be used for
words that are correct for the language, but are hardly ever used and could be
a typing mistake anyway. When the same word is found as good it won't be
highlighted as rare.

This flag can also be used on an affix, so that a basic word is not rare but
the basic word plus affix is rare |spell-affix-flags|. However, if the word
also appears as a good word in another way (e.g., in another region) it won't
be marked as rare.

BAD WORDS *spell-BAD*

In the affix file a BAD line can be used to define the affix name used for
bad words. Example:

```
        BAD ! ~
```

This can be used to exclude words that would otherwise be good. For example
"the the" in the .dic file:

```
        the the/! ~
```

Once a word has been marked as bad it won't be undone by encountering the same
word as good.

The flag also applies to the word with affixes, thus this can be used to mark
a whole bunch of related words as bad.

spell-FORBIDDENWORD
FORBIDDENWORD can be used just like BAD. For compatibility with Hunspell.

spell-NEEDAFFIX
The NEEDAFFIX flag is used to require that a word is used with an affix. The
word itself is not a good word (unless there is an empty affix). Example:

```
        NEEDAFFIX + ~
```

COMPOUND WORDS *spell-compound*

A compound word is a longer word made by concatenating words that appear in
the .dic file. To specify which words may be concatenated a character is
used. This character is put in the list of affixes after the word. We will
call this character a flag here. Obviously these flags must be different from
any affix IDs used.

spell-COMPOUNDFLAG
The Myspell compatible method uses one flag, specified with COMPOUNDFLAG. All
words with this flag combine in any order. This means there is no control
over which word comes first. Example:
```
COMPOUNDFLAG c ~
```

spell-COMPOUNDRULE
A more advanced method to specify how compound words can be formed uses
multiple items with multiple flags. This is not compatible with Myspell 3.0.
Let's start with an example:
```
COMPOUNDRULE c+ ~
COMPOUNDRULE se ~
```

The first line defines that words with the "c" flag can be concatenated in any
order. The second line defines compound words that are made of one word with
the "s" flag and one word with the "e" flag. With this dictionary:
```
bork/c ~
onion/s ~
soup/e ~
```

You can make these words:
```
bork
borkbork
borkborkbork
(etc.)
onion
soup
onionsoup
```

The COMPOUNDRULE item may appear multiple times. The argument is made out of
one or more groups, where each group can be:
```
one flag                        e.g., c
alternate flags inside []       e.g., [abc]
```
Optionally this may be followed by:
```
*       the group appears zero or more times, e.g., sm*e
+       the group appears one or more times, e.g., c+
?       the group appears zero times or once, e.g., x?
```

This is similar to the regexp pattern syntax (but not the same!). A few
examples with the sequence of word flags they require:
```
    COMPOUNDRULE x+         x xx xxx etc.
    COMPOUNDRULE yz         yz
    COMPOUNDRULE x+z        xz xxz xxxz etc.
    COMPOUNDRULE yx+        yx yxx yxxx etc.
    COMPOUNDRULE xy?z       xz xyz

    COMPOUNDRULE [abc]z    az bz cz
    COMPOUNDRULE [abc]+z   az aaz abaz bz baz bcbz cz caz cbaz etc.
    COMPOUNDRULE a[xyz]+   ax axx axyz ay ayx ayzz az azy azxy etc.
    COMPOUNDRULE sm*e       se sme smme smmme etc.
    COMPOUNDRULE s[xyz]*e  se sxe sxye sxyxe sye syze sze szye szyxe  etc.
```

A specific example: Allow a compound to be made of two words and a dash:
```
        In the .aff file:
            COMPOUNDRULE sde ~
            NEEDAFFIX x ~
```

```
            COMPOUNDWORDMAX 3 ~
            COMPOUNDMIN 1 ~
        In the .dic file:
            start/s ~
            end/e ~
            -/xd ~
```

This allows for the word "start-end", but not "startend".

An additional implied rule is that, without further flags, a word with a
prefix cannot be compounded after another word, and a word with a suffix
cannot be compounded with a following word. Thus the affix cannot appear
on the inside of a compound word. This can be changed with the
|spell-COMPOUNDPERMITFLAG|.

spell-NEEDCOMPOUND
The NEEDCOMPOUND flag is used to require that a word is used as part of a
compound word. The word itself is not a good word. Example:

```
        NEEDCOMPOUND & ~
```

spell-ONLYINCOMPOUND
The ONLYINCOMPOUND does exactly the same as NEEDCOMPOUND. Supported for
compatibility with Hunspell.

spell-COMPOUNDMIN
The minimal character length of a word used for compounding is specified with
COMPOUNDMIN. Example:

```
        COMPOUNDMIN 5 ~
```

When omitted there is no minimal length. Obviously you could just leave out
the compound flag from short words instead, this feature is present for
compatibility with Myspell.

spell-COMPOUNDWORDMAX
The maximum number of words that can be concatenated into a compound word is
specified with COMPOUNDWORDMAX. Example:

```
        COMPOUNDWORDMAX 3 ~
```

When omitted there is no maximum. It applies to all compound words.

To set a limit for words with specific flags make sure the items in
COMPOUNDRULE where they appear don't allow too many words.

spell-COMPOUNDSYLMAX
The maximum number of syllables that a compound word may contain is specified
with COMPOUNDSYLMAX. Example:

```
        COMPOUNDSYLMAX 6 ~
```

This has no effect if there is no SYLLABLE item. Without COMPOUNDSYLMAX there
is no limit on the number of syllables.

If both COMPOUNDWORDMAX and COMPOUNDSYLMAX are defined, a compound word is
accepted if it fits one of the criteria, thus is either made from up to
COMPOUNDWORDMAX words or contains up to COMPOUNDSYLMAX syllables.

spell-COMPOUNDFORBIDFLAG
The COMPOUNDFORBIDFLAG specifies a flag that can be used on an affix. It
means that the word plus affix cannot be used in a compound word. Example:

```
        affix file:
                COMPOUNDFLAG c ~
                COMPOUNDFORBIDFLAG x ~
```

```
                SFX a Y 2 ~
                SFX a 0 s   . ~
                SFX a 0 ize/x . ~
        dictionary:
                word/c ~
                util/ac ~
```

This allows for "wordutil" and "wordutils" but not "wordutilize".
Note: this doesn't work for postponed prefixes yet.

spell-COMPOUNDPERMITFLAG
The COMPOUNDPERMITFLAG specifies a flag that can be used on an affix. It means that the word plus affix can also be used in a compound word in a way where the affix ends up halfway the word. Without this flag that is not allowed.
Note: this doesn't work for postponed prefixes yet.

spell-COMPOUNDROOT
The COMPOUNDROOT flag is used for words in the dictionary that are already a compound. This means it counts for two words when checking the compounding rules. Can also be used for an affix to count the affix as a compounding word.

spell-CHECKCOMPOUNDPATTERN
CHECKCOMPOUNDPATTERN is used to define patterns that, when matching at the position where two words are compounded together forbids the compound.
For example:

```
        CHECKCOMPOUNDPATTERN o e ~
```

This forbids compounding if the first word ends in "o" and the second word starts with "e".

The arguments must be plain text, no patterns are actually supported, despite the item name. Case is always ignored.

The Hunspell feature to use three arguments and flags is not supported.

spell-NOCOMPOUNDSUGS
This item indicates that using compounding to make suggestions is not a good idea. Use this when compounding is used with very short or one-character words. E.g. to make numbers out of digits. Without this flag creating suggestions would spend most time trying all kind of weird compound words.

```
        NOCOMPOUNDSUGS ~
```

spell-SYLLABLE
The SYLLABLE item defines characters or character sequences that are used to count the number of syllables in a word. Example:

```
        SYLLABLE a\E1e\E9i\EDo\F3\F6\F5u\FA\FC\FBy/aa/au/ea/ee/ei/ie/oa/oe/oo/ou/uu/
ui ~
```

Before the first slash is the set of characters that are counted for one syllable, also when repeated and mixed, until the next character that is not in this set. After the slash come sequences of characters that are counted for one syllable. These are preferred over using characters from the set. With the example "ideeen" has three syllables, counted by "i", "ee" and "e".

Only case-folded letters need to be included.

Another way to restrict compounding was mentioned above: Adding the |spell-COMPOUNDFORBIDFLAG| flag to an affix causes all words that are made with that affix to not be used for compounding.

UNLIMITED COMPOUNDING *spell-NOBREAK*

For some languages, such as Thai, there is no space in between words. This looks like all words are compounded. To specify this use the NOBREAK item in the affix file, without arguments:

```
NOBREAK ~
```

Vim will try to figure out where one word ends and a next starts. When there are spelling mistakes this may not be quite right.

spell-COMMON

Common words can be specified with the COMMON item. This will give better suggestions when editing a short file. Example:

```
COMMON the of to and a in is it you that he was for on are ~
```

The words must be separated by white space, up to 25 per line.
When multiple regions are specified in a ":mkspell" command the common words for all regions are combined and used for all regions.

spell-NOSPLITSUGS

This item indicates that splitting a word to make suggestions is not a good idea. Split-word suggestions will appear only when there are few similar words.

```
NOSPLITSUGS ~
```

spell-NOSUGGEST

The flag specified with NOSUGGEST can be used for words that will not be suggested. Can be used for obscene words.

```
NOSUGGEST % ~
```

REPLACEMENTS *spell-REP*

In the affix file REP items can be used to define common mistakes. This is used to make spelling suggestions. The items define the "from" text and the "to" replacement. Example:

```
REP 4 ~
REP f ph ~
REP ph f ~
REP k ch ~
REP ch k ~
```

The first line specifies the number of REP lines following. Vim ignores the number, but it must be there (for compatibility with Myspell).

Don't include simple one-character replacements or swaps. Vim will try these anyway. You can include whole words if you want to, but you might want to use the "file:" item in 'spellsuggest' instead.

You can include a space by using an underscore:

```
REP the_the the ~
```

SIMILAR CHARACTERS *spell-MAP* *E783*

In the affix file MAP items can be used to define letters that are very much
alike. This is mostly used for a letter with different accents. This is used
to prefer suggestions with these letters substituted. Example:

```
MAP 2 ~
MAP e\E9\EB\EA\E8 ~
MAP u\FC\F9\FA\FB ~
```

The first line specifies the number of MAP lines following. Vim ignores the
number, but the line must be there.

Each letter must appear in only one of the MAP items. It's a bit more
efficient if the first letter is ASCII or at least one without accents.

.SUG FILE *spell-NOSUGFILE*

When soundfolding is specified in the affix file then ":mkspell" will normally
produce a .sug file next to the .spl file. This file is used to find
suggestions by their sound-a-like form quickly. At the cost of a lot of
memory (the amount depends on the number of words, |:mkspell| will display an
estimate when it's done).

To avoid producing a .sug file use this item in the affix file:

```
NOSUGFILE ~
```

Users can simply omit the .sug file if they don't want to use it.

SOUND-A-LIKE *spell-SAL*

In the affix file SAL items can be used to define the sounds-a-like mechanism
to be used. The main items define the "from" text and the "to" replacement.
Simplistic example:

```
SAL CIA                 X ~
SAL CH                  X ~
SAL C                   K ~
SAL K                   K ~
```

There are a few rules and this can become quite complicated. An explanation
how it works can be found in the Aspell manual:
http://aspell.net/man-html/Phonetic-Code.html.

There are a few special items:

```
SAL followup            true ~
SAL collapse_result     true ~
SAL remove_accents      true ~
```

"1" has the same meaning as "true". Any other value means "false".

SIMPLE SOUNDFOLDING *spell-SOFOFROM* *spell-SOFOTO*

The SAL mechanism is complex and slow. A simpler mechanism is mapping all
characters to another character, mapping similar sounding characters to the
same character. At the same time this does case folding. You can not have
both SAL items and simple soundfolding.

```
There are two items required: one to specify the characters that are mapped
and one that specifies the characters they are mapped to.  They must have
exactly the same number of characters.  Example:

    SOFOFROM abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ~
    SOFOTO   ebctefghejklnnepkrstevvkesebctefghejklnnepkrstevvkes ~

In the example all vowels are mapped to the same character 'e'.  Another
method would be to leave out all vowels.  Some characters that sound nearly
the same and are often mixed up, such as 'm' and 'n', are mapped to the same
character.  Don't do this too much, all words will start looking alike.

Characters that do not appear in SOFOFROM will be left out, except that all
white space is replaced by one space.  Sequences of the same character in
SOFOFROM are replaced by one.

You can use the |soundfold()| function to try out the results.  Or set the
'verbose' option to see the score in the output of the |z=| command.


UNSUPPORTED ITEMS                                   *spell-affix-not-supported*

These items appear in the affix file of other spell checkers.  In Vim they are
ignored, not supported or defined in another way.

ACCENT          (Hunspell)                                  *spell-ACCENT*
                Use MAP instead. |spell-MAP|

BREAK           (Hunspell)                                  *spell-BREAK*
                Define break points.  Unclear how it works exactly.
                Not supported.

CHECKCOMPOUNDCASE  (Hunspell)                   *spell-CHECKCOMPOUNDCASE*
                Disallow uppercase letters at compound word boundaries.
                Not supported.

CHECKCOMPOUNDDUP  (Hunspell)                    *spell-CHECKCOMPOUNDDUP*
                Disallow using the same word twice in a compound.  Not
                supported.

CHECKCOMPOUNDREP  (Hunspell)                    *spell-CHECKCOMPOUNDREP*
                Something about using REP items and compound words.  Not
                supported.

CHECKCOMPOUNDTRIPLE  (Hunspell)                 *spell-CHECKCOMPOUNDTRIPLE*
                Forbid three identical characters when compounding.  Not
                supported.

COMPLEXPREFIXES  (Hunspell)                             *spell-COMPLEXPREFIXES*
                Enables using two prefixes.  Not supported.

COMPOUND        (Hunspell)                                  *spell-COMPOUND*
                This is one line with the count of COMPOUND items, followed by
                that many COMPOUND lines with a pattern.
                Remove the first line with the count and rename the other
                items to COMPOUNDRULE |spell-COMPOUNDRULE|

COMPOUNDFIRST   (Hunspell)                              *spell-COMPOUNDFIRST*
                Use COMPOUNDRULE instead. |spell-COMPOUNDRULE|

COMPOUNDBEGIN   (Hunspell)                              *spell-COMPOUNDBEGIN*
                Use COMPOUNDRULE instead. |spell-COMPOUNDRULE|
```

```
COMPOUNDEND     (Hunspell)                              *spell-COMPOUNDEND*
                Use COMPOUNDRULE instead. |spell-COMPOUNDRULE|

COMPOUNDMIDDLE  (Hunspell)                              *spell-COMPOUNDMIDDLE*
                Use COMPOUNDRULE instead. |spell-COMPOUNDRULE|

COMPOUNDRULES   (Hunspell)                              *spell-COMPOUNDRULES*
                Number of COMPOUNDRULE lines following.  Ignored, but the
                argument must be a number.

COMPOUNDSYLLABLE  (Hunspell)                    *spell-COMPOUNDSYLLABLE*
                Use SYLLABLE and COMPOUNDSYLMAX instead. |spell-SYLLABLE|
                |spell-COMPOUNDSYLMAX|

KEY             (Hunspell)                              *spell-KEY*
                Define characters that are close together on the keyboard.
                Used to give better suggestions.  Not supported.

LANG            (Hunspell)                              *spell-LANG*
                This specifies language-specific behavior.  This actually
                moves part of the language knowledge into the program,
                therefore Vim does not support it.  Each language property
                must be specified separately.

LEMMA_PRESENT   (Hunspell)                              *spell-LEMMA_PRESENT*
                Only needed for morphological analysis.

MAXNGRAMSUGS    (Hunspell)                              *spell-MAXNGRAMSUGS*
                Set number of n-gram suggestions.  Not supported.

PSEUDOROOT      (Hunspell)                              *spell-PSEUDOROOT*
                Use NEEDAFFIX instead. |spell-NEEDAFFIX|

SUGSWITHDOTS    (Hunspell)                              *spell-SUGSWITHDOTS*
                Adds dots to suggestions.  Vim doesn't need this.

SYLLABLENUM     (Hunspell)                              *spell-SYLLABLENUM*
                Not supported.

TRY             (Myspell, Hunspell, others)             *spell-TRY*
                Vim does not use the TRY item, it is ignored.  For making
                suggestions the actual characters in the words are used, that
                is much more efficient.

WORDCHARS       (Hunspell)                              *spell-WORDCHARS*
                Used to recognize words.  Vim doesn't need it, because there
                is no need to separate words before checking them (using a
                trie instead of a hashtable).

 vim:tw=78:sw=4:ts=8:ft=help:norl:
*diff.txt*      For Vim version 8.0.  Last change: 2017 Sep 26


                  VIM REFERENCE MANUAL    by Bram Moolenaar


                                *diff* *vimdiff* *gvimdiff* *diff-mode*
This file describes the |+diff| feature: Showing differences between two to
eight versions of the same file.

The basics are explained in section |08.7| of the user manual.
```

{not in Vi}

==
1. Starting diff mode *start-vimdiff*

The easiest way to start editing in diff mode is with the "vimdiff" command.
This starts Vim as usual, and additionally sets up for viewing the differences
between the arguments. >

```
vimdiff file1 file2 [file3 [file4]]
```

This is equivalent to: >

```
vim -d file1 file2 [file3 [file4]]
```

You may also use "gvimdiff" or "vim -d -g". The GUI is started then.
You may also use "viewdiff" or "gviewdiff". Vim starts in readonly mode then.
"r" may be prepended for restricted mode (see |-Z|).

The second and following arguments may also be a directory name. Vim will
then append the file name of the first argument to the directory name to find
the file.

This only works when a standard "diff" command is available. See 'diffexpr'.

Diffs are local to the current tab page |tab-page|. You can't see diffs with
a window in another tab page. This does make it possible to have several
diffs at the same time, each in their own tab page.

What happens is that Vim opens a window for each of the files. This is like
using the |-O| argument. This uses vertical splits. If you prefer horizontal
splits add the |-o| argument: >

```
vimdiff -o file1 file2 [file3 [file4]]
```

If you always prefer horizontal splits include "horizontal" in 'diffopt'.

In each of the edited files these options are set:

```
'diff'          on
'scrollbind'    on
'cursorbind'    on
'scrollopt'     includes "hor"
'wrap'          off
'foldmethod'    "diff"
'foldcolumn'    value from 'diffopt', default is 2
```

These options are set local to the window. When editing another file they are
reset to the global value.
The options can still be overruled from a modeline when re-editing the file.
However, 'foldmethod' and 'wrap' won't be set from a modeline when 'diff' is
set.

The differences shown are actually the differences in the buffer. Thus if you
make changes after loading a file, these will be included in the displayed

diffs. You might have to do ":diffupdate" now and then, not all changes are
immediately taken into account.

In your .vimrc file you could do something special when Vim was started in
diff mode. You could use a construct like this: >

```
        if &diff
           setup for diff mode
        else
           setup for non-diff mode
        endif
```

While already in Vim you can start diff mode in three ways.

E98
:diffs[plit] {filename} *:diffs* *:diffsplit*
Open a new window on the file {filename}. The options are set as for "vimdiff" for the current and the newly opened window. Also see 'diffexpr'.

:difft *:diffthis*
:difft[his] Make the current window part of the diff windows. This sets the options like for "vimdiff".

:diffp[atch] {patchfile} *E816* *:diffp* *:diffpatch*
Use the current buffer, patch it with the diff found in {patchfile} and open a buffer on the result. The options are set as for "vimdiff".
{patchfile} can be in any format that the "patch" program understands or 'patchexpr' can handle.
Note that {patchfile} should only contain a diff for one file, the current file. If {patchfile} contains diffs for other files as well, the results are unpredictable. Vim changes directory to /tmp to avoid files in the current directory accidentally being patched. But it may still result in various ".rej" files to be created. And when absolute path names are present these files may get patched anyway.

To make these commands use a vertical split, prepend |:vertical|. Examples: >

```
        :vert diffsplit main.c~
        :vert diffpatch /tmp/diff
```

If you always prefer a vertical split include "vertical" in 'diffopt'.

E96
There can be up to eight buffers with 'diff' set.

Since the option values are remembered with the buffer, you can edit another file for a moment and come back to the same file and be in diff mode again.

:diffo *:diffoff*
:diffo[ff] Switch off diff mode for the current window. Resets related options also when 'diff' was not set.

:diffo[ff]! Switch off diff mode for the current window and in all windows in the current tab page where 'diff' is set. Resetting related options only happens in a window that has 'diff' set, if the current window does not have 'diff' set then no options in it are changed.
Hidden buffers are also removed from the list of diff'ed buffers.

The `:diffoff` command resets the relevant options to the values they had when using `:diffsplit`, `:diffpatch` , `:diffthis`. or starting Vim in diff mode. When using `:diffoff` twice the last saved values are restored. Otherwise they are set to their default value:

```
'diff'          off
'scrollbind'    off
'cursorbind'    off
'scrollopt'     without "hor"
'wrap'          on
'foldmethod'    "manual"
'foldcolumn'    0
```

==========

2. Viewing diffs *view-diffs*

The effect is that the diff windows show the same text, with the differences highlighted. When scrolling the text, the 'scrollbind' option will make the text in other windows to be scrolled as well. With vertical splits the text should be aligned properly.

The alignment of text will go wrong when:
- 'wrap' is on, some lines will be wrapped and occupy two or more screen lines
- folds are open in one window but not another
- 'scrollbind' is off
- changes have been made to the text
- "filler" is not present in 'diffopt', deleted/inserted lines makes the alignment go wrong

All the buffers edited in a window where the 'diff' option is set will join in the diff. This is also possible for hidden buffers. They must have been edited in a window first for this to be possible. To get rid of the hidden buffers use `:diffoff!`.

:DiffOrig *diff-original-file*

Since 'diff' is a window-local option, it's possible to view the same buffer in diff mode in one window and "normal" in another window. It is also possible to view the changes you have made to a buffer since the file was loaded. Since Vim doesn't allow having two buffers for the same file, you need another buffer. This command is useful: >

```
command DiffOrig vert new | set bt=nofile | r ++edit # | 0d_
       \ | diffthis | wincmd p | diffthis
```

(this is in |vimrc_example.vim|). Use ":DiffOrig" to see the differences between the current buffer and the file it was loaded from.

A buffer that is unloaded cannot be used for the diff. But it does work for hidden buffers. You can use ":hide" to close a window without unloading the buffer. If you don't want a buffer to remain used for the diff do ":set nodiff" before hiding it.

:dif *:diffupdate*

:dif[fupdate][!] Update the diff highlighting and folds.

Vim attempts to keep the differences updated when you make changes to the text. This mostly takes care of inserted and deleted lines. Changes within a line and more complicated changes do not cause the differences to be updated. To force the differences to be updated use: >

```
:diffupdate
```

If the ! is included Vim will check if the file was changed externally and
needs to be reloaded. It will prompt for each changed file, like `:checktime`
was used.

Vim will show filler lines for lines that are missing in one window but are
present in another. These lines were inserted in another file or deleted in
this file. Removing "filler" from the 'diffopt' option will make Vim not
display these filler lines.

Folds are used to hide the text that wasn't changed. See |folding| for all
the commands that can be used with folds.

The context of lines above a difference that are not included in the fold can
be set with the 'diffopt' option. For example, to set the context to three
lines: >

```
:set diffopt=filler,context:3
```

The diffs are highlighted with these groups:

|hl-DiffAdd|	DiffAdd	Added (inserted) lines. These lines exist in this buffer but not in another.
|hl-DiffChange|	DiffChange	Changed lines.
|hl-DiffText|	DiffText	Changed text inside a Changed line. Vim finds the first character that is different, and the last character that is different (searching from the end of the line). The text in between is highlighted. This means that parts in the middle that are still the same are highlighted anyway. The 'diffopt' flags "iwhite" and "icase" are used here.
|hl-DiffDelete|	DiffDelete	Deleted lines. Also called filler lines, because they don't really exist in this buffer.

==
3. Jumping to diffs *jumpto-diffs*

Two commands can be used to jump to diffs:

[c
[c Jump backwards to the previous start of a change. When a count is used, do it that many times.

]c
]c Jump forwards to the next start of a change. When a count is used, do it that many times.

It is an error if there is no change for the cursor to move to.

==
4. Diff copying *copy-diffs* *E99* *E100* *E101* *E102* *E103* *merge*

There are two commands to copy text from one buffer to another. The result is
that the buffers will be equal within the specified range.

:diffg *:diffget*
:[range]diffg[et] [bufspec]
Modify the current buffer to undo difference with another buffer. If [bufspec] is given, that buffer is used. If [bufspec] refers to the current buffer then nothing happens. Otherwise this only works if there is one other buffer in diff

mode.
See below for [range].

:diffpu *:diffput* *E793*
:[range]diffpu[t] [bufspec]
Modify another buffer to undo difference with the current buffer. Just like ":diffget" but the other buffer is modified instead of the current one.
When [bufspec] is omitted and there is more than one other buffer in diff mode where 'modifiable' is set this fails.
See below for [range].

do
[count]do Same as ":diffget" without range. The "o" stands for "obtain" ("dg" can't be used, it could be the start of "dgg"!). Note: this doesn't work in Visual mode.
If you give a [count], it is used as the [bufspec] argument for ":diffget".

dp
[count]dp Same as ":diffput" without range. Note: this doesn't work in Visual mode.
If you give a [count], it is used as the [bufspec] argument for ":diffput".

When no [range] is given, the diff at the cursor position or just above it is affected. When [range] is used, Vim tries to only put or get the specified lines. When there are deleted lines, this may not always be possible.

There can be deleted lines below the last line of the buffer. When the cursor is on the last line in the buffer and there is no diff above this line, the ":diffget" and "do" commands will obtain lines from the other buffer.

To be able to get those lines from another buffer in a [range] it's allowed to use the last line number plus one. This command gets all diffs from the other buffer: >

```
:1,$+1diffget
```

Note that deleted lines are displayed, but not counted as text lines. You can't move the cursor into them. To fill the deleted lines with the lines from another buffer use ":diffget" on the line below them.

E787
When the buffer that is about to be modified is read-only and the autocommand that is triggered by |FileChangedRO| changes buffers the command will fail. The autocommand must not change buffers.

The [bufspec] argument above can be a buffer number, a pattern for a buffer name or a part of a buffer name. Examples:

```
:diffget               Use the other buffer which is in diff mode
:diffget 3             Use buffer 3
:diffget v2            Use the buffer which matches "v2" and is in
                       diff mode (e.g., "file.c.v2")
```

==
5. Diff options *diff-options*

Also see |'diffopt'| and the "diff" item of |'fillchars'|.

diff-slow *diff_translations*

```
For very long lines, the diff syntax highlighting might be slow, especially
since it tries to match all different kind of localisations. To disable
localisations and speed up the syntax highlighting, set the global variable
g:diff_translations to zero: >

    let g:diff_translations = 0
<
After setting this variable, Reload the syntax script: >

    set syntax=diff
<


FINDING THE DIFFERENCES                                    *diff-diffexpr*

The 'diffexpr' option can be set to use something else than the standard
"diff" program to compare two files and find the differences.

When 'diffexpr' is empty, Vim uses this command to find the differences
between file1 and file2: >

        diff file1 file2 > outfile

The ">" is replaced with the value of 'shellredir'.

The output of "diff" must be a normal "ed" style diff.  Do NOT use a context
diff.  This example explains the format that Vim expects: >

        1a2
        > bbb
        4d4
        < 111
        7c7
        < GGG
        ---
        > ggg

The "1a2" item appends the line "bbb".
The "4d4" item deletes the line "111".
The "7c7" item replaces the line "GGG" with "ggg".

When 'diffexpr' is not empty, Vim evaluates it to obtain a diff file in the
format mentioned.  These variables are set to the file names used:

        v:fname_in              original file
        v:fname_new             new version of the same file
        v:fname_out             resulting diff file

Additionally, 'diffexpr' should take care of "icase" and "iwhite" in the
'diffopt' option.  'diffexpr' cannot change the value of 'lines' and
'columns'.

Example (this does almost the same as 'diffexpr' being empty): >

        set diffexpr=MyDiff()
        function MyDiff()
           let opt = ""
           if &diffopt =~ "icase"
             let opt = opt . "-i "
           endif
           if &diffopt =~ "iwhite"
             let opt = opt . "-b "
```

```
           endif
           silent execute "!diff -a --binary " . opt . v:fname_in . " " .
v:fname_new .
                \  " > " . v:fname_out
        endfunction
```

The "-a" argument is used to force comparing the files as text, comparing as binaries isn't useful. The "--binary" argument makes the files read in binary mode, so that a CTRL-Z doesn't end the text on DOS.

E810 *E97*

Vim will do a test if the diff output looks alright. If it doesn't, you will get an error message. Possible causes:
- The "diff" program cannot be executed.
- The "diff" program doesn't produce normal "ed" style diffs (see above).
- The 'shell' and associated options are not set correctly. Try if filtering works with a command like ":!sort".
- You are using 'diffexpr' and it doesn't work.

If it's not clear what the problem is set the 'verbose' option to one or more to see more messages.

The self-installing Vim for MS-Windows includes a diff program. If you don't have it you might want to download a diff.exe. For example from http://gnuwin32.sourceforge.net/packages/diffutils.htm.

USING PATCHES *diff-patchexpr*

The 'patchexpr' option can be set to use something else than the standard "patch" program.

When 'patchexpr' is empty, Vim will call the "patch" program like this: >

```
        patch -o outfile origfile < patchfile
```

This should work fine with most versions of the "patch" program. Note that a CR in the middle of a line may cause problems, it is seen as a line break.

If the default doesn't work for you, set the 'patchexpr' to an expression that will have the same effect. These variables are set to the file names used:

```
        v:fname_in              original file
        v:fname_diff            patch file
        v:fname_out             resulting patched file
```

Example (this does the same as 'patchexpr' being empty): >

```
        set patchexpr=MyPatch()
        function MyPatch()
           :call system("patch -o " . v:fname_out . " " . v:fname_in .
           \  " < " . v:fname_diff)
        endfunction
```

Make sure that using the "patch" program doesn't have unwanted side effects. For example, watch out for additionally generated files, which should be deleted. It should just patch the file and nothing else.

Vim will change directory to "/tmp" or another temp directory before evaluating 'patchexpr'. This hopefully avoids that files in the current directory are accidentally patched. Vim will also delete files starting with v:fname_in and ending in ".rej" and ".orig".

diff.txt   For Vim version 8.0. Last change: 2017 Sep 26

VIM REFERENCE MANUAL   by Bram Moolenaar

diff *vimdiff* *gvimdiff* *diff-mode*

This file describes the |+diff| feature: Showing differences between two to eight versions of the same file.

The basics are explained in section |08.7| of the user manual.

{not in Vi}

==

1. Starting diff mode *start-vimdiff*

The easiest way to start editing in diff mode is with the "vimdiff" command. This starts Vim as usual, and additionally sets up for viewing the differences between the arguments. >

```
vimdiff file1 file2 [file3 [file4]]
```

This is equivalent to: >

```
vim -d file1 file2 [file3 [file4]]
```

You may also use "gvimdiff" or "vim -d -g". The GUI is started then.
You may also use "viewdiff" or "gviewdiff". Vim starts in readonly mode then.
"r" may be prepended for restricted mode (see |-Z|).

The second and following arguments may also be a directory name. Vim will then append the file name of the first argument to the directory name to find the file.

This only works when a standard "diff" command is available. See 'diffexpr'.

Diffs are local to the current tab page |tab-page|. You can't see diffs with a window in another tab page. This does make it possible to have several diffs at the same time, each in their own tab page.

What happens is that Vim opens a window for each of the files. This is like using the |-O| argument. This uses vertical splits. If you prefer horizontal splits add the |-o| argument: >

```
vimdiff -o file1 file2 [file3 [file4]]
```

If you always prefer horizontal splits include "horizontal" in 'diffopt'.

In each of the edited files these options are set:

```
'diff'          on
'scrollbind'    on
'cursorbind'    on
'scrollopt'     includes "hor"
'wrap'          off
```

```
        'foldmethod'    "diff"
        'foldcolumn'    value from 'diffopt', default is 2

These options are set local to the window.  When editing another file they are
reset to the global value.
The options can still be overruled from a modeline when re-editing the file.
However, 'foldmethod' and 'wrap' won't be set from a modeline when 'diff' is
set.

The differences shown are actually the differences in the buffer.  Thus if you
make changes after loading a file, these will be included in the displayed
diffs.  You might have to do ":diffupdate" now and then, not all changes are
immediately taken into account.

In your .vimrc file you could do something special when Vim was started in
diff mode.  You could use a construct like this: >

        if &diff
           setup for diff mode
        else
           setup for non-diff mode
        endif

While already in Vim you can start diff mode in three ways.

                                                        *E98*
:diffs[plit] {filename}                                 *:diffs* *:diffsplit*
                Open a new window on the file {filename}.  The options are set
                as for "vimdiff" for the current and the newly opened window.
                Also see 'diffexpr'.

                                                        *:difft* *:diffthis*
:difft[his]     Make the current window part of the diff windows.  This sets
                the options like for "vimdiff".

:diffp[atch] {patchfile}                          *E816* *:diffp* *:diffpatch*
                Use the current buffer, patch it with the diff found in
                {patchfile} and open a buffer on the result.  The options are
                set as for "vimdiff".
                {patchfile} can be in any format that the "patch" program
                understands or 'patchexpr' can handle.
                Note that {patchfile} should only contain a diff for one file,
                the current file.  If {patchfile} contains diffs for other
                files as well, the results are unpredictable.  Vim changes
                directory to /tmp to avoid files in the current directory
                accidentally being patched.  But it may still result in
                various ".rej" files to be created.  And when absolute path
                names are present these files may get patched anyway.

To make these commands use a vertical split, prepend |:vertical|.  Examples: >

        :vert diffsplit main.c~
        :vert diffpatch /tmp/diff

If you always prefer a vertical split include "vertical" in 'diffopt'.

                                                        *E96*
There can be up to eight buffers with 'diff' set.

Since the option values are remembered with the buffer, you can edit another
file for a moment and come back to the same file and be in diff mode again.
```

:diffo *:diffoff*

:diffo[ff] Switch off diff mode for the current window. Resets related options also when 'diff' was not set.

:diffo[ff]! Switch off diff mode for the current window and in all windows in the current tab page where 'diff' is set. Resetting related options only happens in a window that has 'diff' set, if the current window does not have 'diff' set then no options in it are changed.
Hidden buffers are also removed from the list of diff'ed buffers.

The `:diffoff` command resets the relevant options to the values they had when using `:diffsplit`, `:diffpatch` , `:diffthis`. or starting Vim in diff mode.
When using `:diffoff` twice the last saved values are restored.
Otherwise they are set to their default value:

```
'diff'          off
'scrollbind'    off
'cursorbind'    off
'scrollopt'     without "hor"
'wrap'          on
'foldmethod'    "manual"
'foldcolumn'    0
```

==
2. Viewing diffs *view-diffs*

The effect is that the diff windows show the same text, with the differences highlighted. When scrolling the text, the 'scrollbind' option will make the text in other windows to be scrolled as well. With vertical splits the text should be aligned properly.

The alignment of text will go wrong when:
- 'wrap' is on, some lines will be wrapped and occupy two or more screen lines
- folds are open in one window but not another
- 'scrollbind' is off
- changes have been made to the text
- "filler" is not present in 'diffopt', deleted/inserted lines makes the alignment go wrong

All the buffers edited in a window where the 'diff' option is set will join in the diff. This is also possible for hidden buffers. They must have been edited in a window first for this to be possible. To get rid of the hidden buffers use `:diffoff!`.

:DiffOrig *diff-original-file*

Since 'diff' is a window-local option, it's possible to view the same buffer in diff mode in one window and "normal" in another window. It is also possible to view the changes you have made to a buffer since the file was loaded. Since Vim doesn't allow having two buffers for the same file, you need another buffer. This command is useful: >

```
command DiffOrig vert new | set bt=nofile | r ++edit # | 0d_
       \ | diffthis | wincmd p | diffthis
```

(this is in |vimrc_example.vim|). Use ":DiffOrig" to see the differences between the current buffer and the file it was loaded from.

A buffer that is unloaded cannot be used for the diff. But it does work for hidden buffers. You can use ":hide" to close a window without unloading the buffer. If you don't want a buffer to remain used for the diff do ":set nodiff" before hiding it.

:dif *:diffupdate*
:dif[fupdate][!] Update the diff highlighting and folds.

Vim attempts to keep the differences updated when you make changes to the text. This mostly takes care of inserted and deleted lines. Changes within a line and more complicated changes do not cause the differences to be updated. To force the differences to be updated use: >

```
:diffupdate
```

If the ! is included Vim will check if the file was changed externally and needs to be reloaded. It will prompt for each changed file, like `:checktime` was used.

Vim will show filler lines for lines that are missing in one window but are present in another. These lines were inserted in another file or deleted in this file. Removing "filler" from the 'diffopt' option will make Vim not display these filler lines.

Folds are used to hide the text that wasn't changed. See |folding| for all the commands that can be used with folds.

The context of lines above a difference that are not included in the fold can be set with the 'diffopt' option. For example, to set the context to three lines: >

```
:set diffopt=filler,context:3
```

The diffs are highlighted with these groups:

|hl-DiffAdd|	DiffAdd	Added (inserted) lines. These lines exist in this buffer but not in another.
|hl-DiffChange|	DiffChange	Changed lines.
|hl-DiffText|	DiffText	Changed text inside a Changed line. Vim finds the first character that is different, and the last character that is different (searching from the end of the line). The text in between is highlighted. This means that parts in the middle that are still the same are highlighted anyway. The 'diffopt' flags "iwhite" and "icase" are used here.
|hl-DiffDelete|	DiffDelete	Deleted lines. Also called filler lines, because they don't really exist in this buffer.

==========

3. Jumping to diffs *jumpto-diffs*

Two commands can be used to jump to diffs:

[c
[c Jump backwards to the previous start of a change. When a count is used, do it that many times.

]c
]c Jump forwards to the next start of a change. When a count is used, do it that many times.

It is an error if there is no change for the cursor to move to.

==========

4. Diff copying *copy-diffs* *E99* *E100* *E101* *E102* *E103*
merge

There are two commands to copy text from one buffer to another. The result is that the buffers will be equal within the specified range.

:diffg *:diffget*

:[range]diffg[et] [bufspec]
Modify the current buffer to undo difference with another buffer. If [bufspec] is given, that buffer is used. If [bufspec] refers to the current buffer then nothing happens. Otherwise this only works if there is one other buffer in diff mode.
See below for [range].

:diffpu *:diffput* *E793*

:[range]diffpu[t] [bufspec]
Modify another buffer to undo difference with the current buffer. Just like ":diffget" but the other buffer is modified instead of the current one.
When [bufspec] is omitted and there is more than one other buffer in diff mode where 'modifiable' is set this fails.
See below for [range].

do

[count]do Same as ":diffget" without range. The "o" stands for "obtain" ("dg" can't be used, it could be the start of "dgg"!). Note: this doesn't work in Visual mode.
If you give a [count], it is used as the [bufspec] argument for ":diffget".

dp

[count]dp Same as ":diffput" without range. Note: this doesn't work in Visual mode.
If you give a [count], it is used as the [bufspec] argument for ":diffput".

When no [range] is given, the diff at the cursor position or just above it is affected. When [range] is used, Vim tries to only put or get the specified lines. When there are deleted lines, this may not always be possible.

There can be deleted lines below the last line of the buffer. When the cursor is on the last line in the buffer and there is no diff above this line, the ":diffget" and "do" commands will obtain lines from the other buffer.

To be able to get those lines from another buffer in a [range] it's allowed to use the last line number plus one. This command gets all diffs from the other buffer: >

```
:1,$+1diffget
```

Note that deleted lines are displayed, but not counted as text lines. You can't move the cursor into them. To fill the deleted lines with the lines from another buffer use ":diffget" on the line below them.

E787

When the buffer that is about to be modified is read-only and the autocommand that is triggered by |FileChangedRO| changes buffers the command will fail. The autocommand must not change buffers.

The [bufspec] argument above can be a buffer number, a pattern for a buffer name or a part of a buffer name. Examples:

```
        :diffget               Use the other buffer which is in diff mode
        :diffget 3             Use buffer 3
        :diffget v2            Use the buffer which matches "v2" and is in
                               diff mode (e.g., "file.c.v2")

==============================================================================
5. Diff options                                          *diff-options*

Also see |'diffopt'| and the "diff" item of |'fillchars'|.

                                              *diff-slow* *diff_translations*
For very long lines, the diff syntax highlighting might be slow, especially
since it tries to match all different kind of localisations. To disable
localisations and speed up the syntax highlighting, set the global variable
g:diff_translations to zero: >

    let g:diff_translations = 0
<
After setting this variable, Reload the syntax script: >

    set syntax=diff
<


FINDING THE DIFFERENCES                                  *diff-diffexpr*

The 'diffexpr' option can be set to use something else than the standard
"diff" program to compare two files and find the differences.

When 'diffexpr' is empty, Vim uses this command to find the differences
between file1 and file2: >

        diff file1 file2 > outfile

The ">" is replaced with the value of 'shellredir'.

The output of "diff" must be a normal "ed" style diff.  Do NOT use a context
diff.  This example explains the format that Vim expects: >

        1a2
        > bbb
        4d4
        < 111
        7c7
        < GGG
        ---
        > ggg

The "1a2" item appends the line "bbb".
The "4d4" item deletes the line "111".
The "7c7" item replaces the line "GGG" with "ggg".

When 'diffexpr' is not empty, Vim evaluates it to obtain a diff file in the
format mentioned.  These variables are set to the file names used:

        v:fname_in             original file
        v:fname_new            new version of the same file
        v:fname_out            resulting diff file

Additionally, 'diffexpr' should take care of "icase" and "iwhite" in the
'diffopt' option.  'diffexpr' cannot change the value of 'lines' and
'columns'.
```

Example (this does almost the same as 'diffexpr' being empty): >

```
        set diffexpr=MyDiff()
        function MyDiff()
           let opt = ""
           if &diffopt =~ "icase"
             let opt = opt . "-i "
           endif
           if &diffopt =~ "iwhite"
             let opt = opt . "-b "
           endif
           silent execute "!diff -a --binary " . opt . v:fname_in . " " .
v:fname_new .
                \  " > " . v:fname_out
        endfunction
```

The "-a" argument is used to force comparing the files as text, comparing as binaries isn't useful. The "--binary" argument makes the files read in binary mode, so that a CTRL-Z doesn't end the text on DOS.

E810 *E97*

Vim will do a test if the diff output looks alright. If it doesn't, you will get an error message. Possible causes:
- The "diff" program cannot be executed.
- The "diff" program doesn't produce normal "ed" style diffs (see above).
- The 'shell' and associated options are not set correctly. Try if filtering works with a command like ":!sort".
- You are using 'diffexpr' and it doesn't work.

If it's not clear what the problem is set the 'verbose' option to one or more to see more messages.

The self-installing Vim for MS-Windows includes a diff program. If you don't have it you might want to download a diff.exe. For example from http://gnuwin32.sourceforge.net/packages/diffutils.htm.

USING PATCHES *diff-patchexpr*

The 'patchexpr' option can be set to use something else than the standard "patch" program.

When 'patchexpr' is empty, Vim will call the "patch" program like this: >

```
        patch -o outfile origfile < patchfile
```

This should work fine with most versions of the "patch" program. Note that a CR in the middle of a line may cause problems, it is seen as a line break.

If the default doesn't work for you, set the 'patchexpr' to an expression that will have the same effect. These variables are set to the file names used:

```
        v:fname_in              original file
        v:fname_diff            patch file
        v:fname_out             resulting patched file
```

Example (this does the same as 'patchexpr' being empty): >

```
        set patchexpr=MyPatch()
        function MyPatch()
           :call system("patch -o " . v:fname_out . " " . v:fname_in .
           \  " < " . v:fname_diff)
```

```
        endfunction

Make sure that using the "patch" program doesn't have unwanted side effects.
For example, watch out for additionally generated files, which should be
deleted.  It should just patch the file and nothing else.
   Vim will change directory to "/tmp" or another temp directory before
evaluating 'patchexpr'.  This hopefully avoids that files in the current
directory are accidentally patched.  Vim will also delete files starting with
v:fname_in and ending in ".rej" and ".orig".

 vim:tw=78:ts=8:ft=help:norl:
*autocmd.txt*   For Vim version 8.0.  Last change: 2017 Jul 14


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Automatic commands                                          *autocommand*

For a basic explanation, see section |40.3| in the user manual.

1.  Introduction                |autocmd-intro|
2.  Defining autocommands       |autocmd-define|
3.  Removing autocommands       |autocmd-remove|
4.  Listing autocommands        |autocmd-list|
5.  Events                      |autocmd-events|
6.  Patterns                    |autocmd-patterns|
7.  Buffer-local autocommands   |autocmd-buflocal|
8.  Groups                      |autocmd-groups|
9.  Executing autocommands      |autocmd-execute|
10. Using autocommands          |autocmd-use|
11. Disabling autocommands      |autocmd-disable|

{Vi does not have any of these commands}
{only when the |+autocmd| feature has not been disabled at compile time}

==============================================================================
1. Introduction                                             *autocmd-intro*

You can specify commands to be executed automatically when reading or writing
a file, when entering or leaving a buffer or window, and when exiting Vim.
For example, you can create an autocommand to set the 'cindent' option for
files matching *.c.  You can also use autocommands to implement advanced
features, such as editing compressed files (see |gzip-example|).  The usual
place to put autocommands is in your .vimrc or .exrc file.

                                        *E203* *E204* *E143* *E855* *E937*
WARNING: Using autocommands is very powerful, and may lead to unexpected side
effects.  Be careful not to destroy your text.
- It's a good idea to do some testing on an expendable copy of a file first.
  For example: If you use autocommands to decompress a file when starting to
  edit it, make sure that the autocommands for compressing when writing work
  correctly.
- Be prepared for an error halfway through (e.g., disk full).  Vim will mostly
  be able to undo the changes to the buffer, but you may have to clean up the
  changes to other files by hand (e.g., compress a file that has been
  decompressed).
- If the BufRead* events allow you to edit a compressed file, the FileRead*
  events should do the same (this makes recovery possible in some rare cases).
  It's a good idea to use the same autocommands for the File* and Buf* events
  when possible.
```

```
==============================================================================
2. Defining autocommands                                *autocmd-define*

                                                        *:au* *:autocmd*
:au[tocmd] [group] {event} {pat} [nested] {cmd}
                        Add {cmd} to the list of commands that Vim will
                        execute automatically on {event} for a file matching
                        {pat} |autocmd-patterns|.
                        Vim always adds the {cmd} after existing autocommands,
                        so that the autocommands execute in the order in which
                        they were given.  See |autocmd-nested| for [nested].

The special pattern <buffer> or <buffer=N> defines a buffer-local autocommand.
See |autocmd-buflocal|.

Note: The ":autocmd" command can only be followed by another command when the
'|' appears before {cmd}.  This works: >
        :augroup mine | au! BufRead | augroup END
But this sees "augroup" as part of the defined command: >
        :augroup mine | au BufRead * set tw=70 | augroup END

Note that special characters (e.g., "%", "<cword>") in the ":autocmd"
arguments are not expanded when the autocommand is defined.  These will be
expanded when the Event is recognized, and the {cmd} is executed.  The only
exception is that "<sfile>" is expanded when the autocmd is defined.  Example:
>
        :au BufNewFile,BufRead *.html so <sfile>:h/html.vim

Here Vim expands <sfile> to the name of the file containing this line.

`:autocmd` adds to the list of autocommands regardless of whether they are
already present.  When your .vimrc file is sourced twice, the autocommands
will appear twice.  To avoid this, define your autocommands in a group, so
that you can easily clear them: >

        augroup vimrc
          autocmd!      " Remove all vimrc autocommands
          au BufNewFile,BufRead *.html so <sfile>:h/html.vim
        augroup END

If you don't want to remove all autocommands, you can instead use a variable
to ensure that Vim includes the autocommands only once: >

        :if !exists("autocommands_loaded")
        :  let autocommands_loaded = 1
        :  au ...
        :endif

When the [group] argument is not given, Vim uses the current group (as defined
with ":augroup"); otherwise, Vim uses the group defined with [group].  Note
that [group] must have been defined before.  You cannot define a new group
with ":au group ..."; use ":augroup" for that.

While testing autocommands, you might find the 'verbose' option to be useful: >
        :set verbose=9
This setting makes Vim echo the autocommands as it executes them.

When defining an autocommand in a script, it will be able to call functions
local to the script and use mappings local to the script.  When the event is
triggered and the command executed, it will run in the context of the script
it was defined in.  This matters if |<SID>| is used in a command.
```

When executing the commands, the message from one command overwrites a
previous message. This is different from when executing the commands
manually. Mostly the screen will not scroll up, thus there is no hit-enter
prompt. When one command outputs two messages this can happen anyway.

==
3. Removing autocommands *autocmd-remove*

:au[tocmd]! [group] {event} {pat} [nested] {cmd}
Remove all autocommands associated with {event} and
{pat}, and add the command {cmd}. See
|autocmd-nested| for [nested].

:au[tocmd]! [group] {event} {pat}
Remove all autocommands associated with {event} and
{pat}.

:au[tocmd]! [group] * {pat}
Remove all autocommands associated with {pat} for all
events.

:au[tocmd]! [group] {event}
Remove ALL autocommands for {event}.
Warning: You should not do this without a group for
|BufRead| and other common events, it can break
plugins, syntax highlighting, etc.

:au[tocmd]! [group] Remove ALL autocommands.
Warning: You should normally not do this without a
group, it breaks plugins, syntax highlighting, etc.

When the [group] argument is not given, Vim uses the current group (as defined
with ":augroup"); otherwise, Vim uses the group defined with [group].

==
4. Listing autocommands *autocmd-list*

:au[tocmd] [group] {event} {pat}
Show the autocommands associated with {event} and
{pat}.

:au[tocmd] [group] * {pat}
Show the autocommands associated with {pat} for all
events.

:au[tocmd] [group] {event}
Show all autocommands for {event}.

:au[tocmd] [group] Show all autocommands.

If you provide the [group] argument, Vim lists only the autocommands for
[group]; otherwise, Vim lists the autocommands for ALL groups. Note that this
argument behavior differs from that for defining and removing autocommands.

In order to list buffer-local autocommands, use a pattern in the form <buffer>
or <buffer=N>. See |autocmd-buflocal|.

:autocmd-verbose
When 'verbose' is non-zero, listing an autocommand will also display where it
was last defined. Example: >

```
:verbose autocmd BufEnter
```

```
    FileExplorer  BufEnter
        *         call s:LocalBrowse(expand("<amatch>"))
            Last set from /usr/share/vim/vim-7.0/plugin/NetrwPlugin.vim
<
See |:verbose-cmd| for more information.

================================================================================
5. Events                                       *autocmd-events* *E215* *E216*

You can specify a comma-separated list of event names.  No white space can be
used in this list.  The command applies to all the events in the list.

For READING FILES there are four kinds of events possible:
        BufNewFile                      starting to edit a non-existent file
        BufReadPre      BufReadPost     starting to edit an existing file
        FilterReadPre   FilterReadPost  read the temp file with filter output
        FileReadPre     FileReadPost    any other file read
Vim uses only one of these four kinds when reading a file.  The "Pre" and
"Post" events are both triggered, before and after reading the file.

Note that the autocommands for the *ReadPre events and all the Filter events
are not allowed to change the current buffer (you will get an error message if
this happens).  This is to prevent the file to be read into the wrong buffer.

Note that the 'modified' flag is reset AFTER executing the BufReadPost
and BufNewFile autocommands.  But when the 'modified' option was set by the
autocommands, this doesn't happen.

You can use the 'eventignore' option to ignore a number of events or all
events.
                                        *autocommand-events* *{event}*
Vim recognizes the following events.  Vim ignores the case of event names
(e.g., you can use "BUFread" or "bufread" instead of "BufRead").

First an overview by function with a short explanation.  Then the list
alphabetically with full explanations |autocmd-events-abc|.

Name                    triggered by ~

        Reading
|BufNewFile|            starting to edit a file that doesn't exist
|BufReadPre|            starting to edit a new buffer, before reading the file
|BufRead|               starting to edit a new buffer, after reading the file
|BufReadPost|           starting to edit a new buffer, after reading the file
|BufReadCmd|            before starting to edit a new buffer |Cmd-event|

|FileReadPre|           before reading a file with a ":read" command
|FileReadPost|          after reading a file with a ":read" command
|FileReadCmd|           before reading a file with a ":read" command |Cmd-event|

|FilterReadPre|         before reading a file from a filter command
|FilterReadPost|        after reading a file from a filter command

|StdinReadPre|          before reading from stdin into the buffer
|StdinReadPost|         After reading from the stdin into the buffer

        Writing
|BufWrite|              starting to write the whole buffer to a file
|BufWritePre|           starting to write the whole buffer to a file
|BufWritePost|          after writing the whole buffer to a file
|BufWriteCmd|           before writing the whole buffer to a file |Cmd-event|
```

```
|FileWritePre|          starting to write part of a buffer to a file
|FileWritePost|         after writing part of a buffer to a file
|FileWriteCmd|          before writing part of a buffer to a file |Cmd-event|

|FileAppendPre|         starting to append to a file
|FileAppendPost|        after appending to a file
|FileAppendCmd|         before appending to a file |Cmd-event|

|FilterWritePre|        starting to write a file for a filter command or diff
|FilterWritePost|       after writing a file for a filter command or diff

        Buffers
|BufAdd|                just after adding a buffer to the buffer list
|BufCreate|             just after adding a buffer to the buffer list
|BufDelete|             before deleting a buffer from the buffer list
|BufWipeout|            before completely deleting a buffer

|BufFilePre|            before changing the name of the current buffer
|BufFilePost|           after changing the name of the current buffer

|BufEnter|              after entering a buffer
|BufLeave|              before leaving to another buffer
|BufWinEnter|           after a buffer is displayed in a window
|BufWinLeave|           before a buffer is removed from a window

|BufUnload|             before unloading a buffer
|BufHidden|             just after a buffer has become hidden
|BufNew|                just after creating a new buffer

|SwapExists|            detected an existing swap file

        Options
|FileType|              when the 'filetype' option has been set
|Syntax|                when the 'syntax' option has been set
|EncodingChanged|       after the 'encoding' option has been changed
|TermChanged|           after the value of 'term' has changed
|OptionSet|             after setting any option

        Startup and exit
|VimEnter|              after doing all the startup stuff
|GUIEnter|              after starting the GUI successfully
|GUIFailed|             after starting the GUI failed
|TermResponse|          after the terminal response to |t_RV| is received

|QuitPre|               when using `:quit`, before deciding whether to quit
|VimLeavePre|           before exiting Vim, before writing the viminfo file
|VimLeave|              before exiting Vim, after writing the viminfo file

        Various
|FileChangedShell|      Vim notices that a file changed since editing started
|FileChangedShellPost|  After handling a file changed since editing started
|FileChangedRO|         before making the first change to a read-only file

|ShellCmdPost|          after executing a shell command
|ShellFilterPost|       after filtering with a shell command

|CmdUndefined|          a user command is used but it isn't defined
|FuncUndefined|         a user function is used but it isn't defined
|SpellFileMissing|      a spell file is used but it can't be found
|SourcePre|             before sourcing a Vim script
|SourceCmd|             before sourcing a Vim script |Cmd-event|
```

```
|VimResized|            after the Vim window size changed
|FocusGained|           Vim got input focus
|FocusLost|             Vim lost input focus
|CursorHold|            the user doesn't press a key for a while
|CursorHoldI|           the user doesn't press a key for a while in Insert mode
|CursorMoved|           the cursor was moved in Normal mode
|CursorMovedI|          the cursor was moved in Insert mode

|WinNew|                after creating a new window
|TabNew|                after creating a new tab page
|TabClosed|             after closing a tab page
|WinEnter|              after entering another window
|WinLeave|              before leaving a window
|TabEnter|              after entering another tab page
|TabLeave|              before leaving a tab page
|CmdwinEnter|           after entering the command-line window
|CmdwinLeave|           before leaving the command-line window

|InsertEnter|           starting Insert mode
|InsertChange|          when typing <Insert> while in Insert or Replace mode
|InsertLeave|           when leaving Insert mode
|InsertCharPre|         when a character was typed in Insert mode, before
                        inserting it

|TextChanged|           after a change was made to the text in Normal mode
|TextChangedI|          after a change was made to the text in Insert mode

|ColorScheme|           after loading a color scheme

|RemoteReply|           a reply from a server Vim was received

|QuickFixCmdPre|        before a quickfix command is run
|QuickFixCmdPost|       after a quickfix command is run

|SessionLoadPost|       after loading a session file

|MenuPopup|             just before showing the popup menu
|CompleteDone|          after Insert mode completion is done

|User|                  to be used in combination with ":doautocmd"


The alphabetical list of autocommand events:            *autocmd-events-abc*

                                                        *BufCreate* *BufAdd*
BufAdd or BufCreate             Just after creating a new buffer which is
                                added to the buffer list, or adding a buffer
                                to the buffer list.
                                Also used just after a buffer in the buffer
                                list has been renamed.
                                The BufCreate event is for historic reasons.
                                NOTE: When this autocommand is executed, the
                                current buffer "%" may be different from the
                                buffer being created "<afile>".
                                                        *BufDelete*
BufDelete                       Before deleting a buffer from the buffer list.
                                The BufUnload may be called first (if the
                                buffer was loaded).
                                Also used just before a buffer in the buffer
                                list is renamed.
                                NOTE: When this autocommand is executed, the
                                current buffer "%" may be different from the
```

buffer being deleted "<afile>" and "<abuf>".
Don't change to another buffer, it will cause
problems.
BufEnter

BufEnter After entering a buffer. Useful for setting
options for a file type. Also executed when
starting to edit a buffer, after the
BufReadPost autocommands.
BufFilePost

BufFilePost After changing the name of the current buffer
with the ":file" or ":saveas" command.
BufFilePre

BufFilePre Before changing the name of the current buffer
with the ":file" or ":saveas" command.
BufHidden

BufHidden Just after a buffer has become hidden. That
is, when there are no longer windows that show
the buffer, but the buffer is not unloaded or
deleted. Not used for ":qa" or ":q" when
exiting Vim.
NOTE: When this autocommand is executed, the
current buffer "%" may be different from the
buffer being unloaded "<afile>".
BufLeave

BufLeave Before leaving to another buffer. Also when
leaving or closing the current window and the
new current window is not for the same buffer.
Not used for ":qa" or ":q" when exiting Vim.
BufNew

BufNew Just after creating a new buffer. Also used
just after a buffer has been renamed. When
the buffer is added to the buffer list BufAdd
will be triggered too.
NOTE: When this autocommand is executed, the
current buffer "%" may be different from the
buffer being created "<afile>".
BufNewFile

BufNewFile When starting to edit a file that doesn't
exist. Can be used to read in a skeleton
file.
BufRead *BufReadPost*

BufRead or BufReadPost When starting to edit a new buffer, after
reading the file into the buffer, before
executing the modelines. See |BufWinEnter|
for when you need to do something after
processing the modelines.
This does NOT work for ":r file". Not used
when the file doesn't exist. Also used after
successfully recovering a file.
Also triggered for the filetypedetect group
when executing ":filetype detect" and when
writing an unnamed buffer in a way that the
buffer gets a name.
BufReadCmd

BufReadCmd Before starting to edit a new buffer. Should
read the file into the buffer. |Cmd-event|
BufReadPre *E200* *E201*

BufReadPre When starting to edit a new buffer, before
reading the file into the buffer. Not used
if the file doesn't exist.
BufUnload

BufUnload Before unloading a buffer. This is when the

text in the buffer is going to be freed. This may be after a BufWritePost and before a BufDelete. Also used for all buffers that are loaded when Vim is going to exit.
NOTE: When this autocommand is executed, the current buffer "%" may be different from the buffer being unloaded "<afile>".
Don't change to another buffer or window, it will cause problems!
When exiting and v:dying is 2 or more this event is not triggered.

BufWinEnter

BufWinEnter — After a buffer is displayed in a window. This can be when the buffer is loaded (after processing the modelines) or when a hidden buffer is displayed in a window (and is no longer hidden).
Does not happen for |:split| without arguments, since you keep editing the same buffer, or ":split" with a file that's already open in a window, because it re-uses an existing buffer. But it does happen for a ":split" with the name of the current buffer, since it reloads that buffer.

BufWinLeave

BufWinLeave — Before a buffer is removed from a window. Not when it's still visible in another window. Also triggered when exiting. It's triggered before BufUnload or BufHidden.
NOTE: When this autocommand is executed, the current buffer "%" may be different from the buffer being unloaded "<afile>".
When exiting and v:dying is 2 or more this event is not triggered.

BufWipeout

BufWipeout — Before completely deleting a buffer. The BufUnload and BufDelete events may be called first (if the buffer was loaded and was in the buffer list). Also used just before a buffer is renamed (also when it's not in the buffer list).
NOTE: When this autocommand is executed, the current buffer "%" may be different from the buffer being deleted "<afile>".
Don't change to another buffer, it will cause problems.

BufWrite *BufWritePre*

BufWrite or BufWritePre — Before writing the whole buffer to a file.

BufWriteCmd

BufWriteCmd — Before writing the whole buffer to a file. Should do the writing of the file and reset 'modified' if successful, unless '+' is in 'cpo' and writing to another file |cpo-+|.
The buffer contents should not be changed.
When the command resets 'modified' the undo information is adjusted to mark older undo states as 'modified', like |:write| does.
|Cmd-event|

BufWritePost

BufWritePost — After writing the whole buffer to a file (should undo the commands for BufWritePre).

CmdUndefined

CmdUndefined When a user command is used but it isn't defined. Useful for defining a command only when it's used. The pattern is matched against the command name. Both <amatch> and <afile> are set to the name of the command. NOTE: Autocompletion won't work until the command is defined. An alternative is to always define the user command and have it invoke an autoloaded function. See |autoload|.

CmdwinEnter

CmdwinEnter After entering the command-line window. Useful for setting options specifically for this special type of window. This is triggered _instead_ of BufEnter and WinEnter. <afile> is set to a single character, indicating the type of command-line. |cmdwin-char|

CmdwinLeave

CmdwinLeave Before leaving the command-line window. Useful to clean up any global setting done with CmdwinEnter. This is triggered _instead_ of BufLeave and WinLeave. <afile> is set to a single character, indicating the type of command-line. |cmdwin-char|

ColorScheme

ColorScheme After loading a color scheme. |:colorscheme| The pattern is matched against the colorscheme name. <afile> can be used for the name of the actual file where this option was set, and <amatch> for the new colorscheme name.

CompleteDone

CompleteDone After Insert mode completion is done. Either when something was completed or abandoning completion. |ins-completion| The |v:completed_item| variable contains information about the completed item.

CursorHold

CursorHold When the user doesn't press a key for the time specified with 'updatetime'. Not re-triggered until the user has pressed a key (i.e. doesn't fire every 'updatetime' ms if you leave Vim to make some coffee. :) See |CursorHold-example| for previewing tags. This event is only triggered in Normal mode. It is not triggered when waiting for a command argument to be typed, or a movement after an operator. While recording the CursorHold event is not triggered. |q|

<CursorHold>

Internally the autocommand is triggered by the <CursorHold> key. In an expression mapping |getchar()| may see this character.

Note: Interactive commands cannot be used for this event. There is no hit-enter prompt, the screen is updated directly (when needed).

Note: In the future there will probably be another option to set the time.
Hint: to force an update of the status lines use: >
:let &ro = &ro
< {only on Amiga, Unix, Win32, MSDOS and all GUI versions}

CursorHoldI

CursorHoldI Just like CursorHold, but in Insert mode. Not triggered when waiting for another key, e.g. after CTRL-V, and not when in CTRL-X mode |insert_expand|.

CursorMoved

CursorMoved After the cursor was moved in Normal or Visual mode. Also when the text of the cursor line has been changed, e.g., with "x", "rx" or "p". Not triggered when there is typeahead or when an operator is pending.
For an example see |match-parens|.
Careful: This is triggered very often, don't do anything that the user does not expect or that is slow.

CursorMovedI

CursorMovedI After the cursor was moved in Insert mode. Not triggered when the popup menu is visible. Otherwise the same as CursorMoved.

EncodingChanged

EncodingChanged Fires off after the 'encoding' option has been changed. Useful to set up fonts, for example.

FileAppendCmd

FileAppendCmd Before appending to a file. Should do the appending to the file. Use the '[and '] marks for the range of lines.|Cmd-event|

FileAppendPost

FileAppendPost After appending to a file.

FileAppendPre

FileAppendPre Before appending to a file. Use the '[and '] marks for the range of lines.

FileChangedRO

FileChangedRO Before making the first change to a read-only file. Can be used to check-out the file from a source control system. Not triggered when the change was caused by an autocommand. This event is triggered when making the first change in a buffer or the first change after 'readonly' was set, just before the change is applied to the text.
WARNING: If the autocommand moves the cursor the effect of the change is undefined.

E788

It is not allowed to change to another buffer here. You can reload the buffer but not edit another one.

E881

If the number of lines changes saving for undo may fail and the change will be aborted.

FileChangedShell

FileChangedShell When Vim notices that the modification time of a file has changed since editing started. Also when the file attributes of the file change or when the size of the file changes.

```
                        |timestamp|
                        Mostly triggered after executing a shell
                        command, but also with a |:checktime| command
                        or when gvim regains input focus.
                        This autocommand is triggered for each changed
                        file.  It is not used when 'autoread' is set
                        and the buffer was not changed.  If a
                        FileChangedShell autocommand is present the
                        warning message and prompt is not given.
                        The |v:fcs_reason| variable is set to indicate
                        what happened and |v:fcs_choice| can be used
                        to tell Vim what to do next.
                        NOTE: When this autocommand is executed, the
                        current buffer "%" may be different from the
                        buffer that was changed, which is in "<afile>".
                        NOTE: The commands must not change the current
                        buffer, jump to another buffer or delete a
                        buffer.  *E246* *E811*
                        NOTE: This event never nests, to avoid an
                        endless loop.  This means that while executing
                        commands for the FileChangedShell event no
                        other FileChangedShell event will be
                        triggered.
                                                *FileChangedShellPost*
FileChangedShellPost    After handling a file that was changed outside
                        of Vim.  Can be used to update the statusline.
                                                *FileEncoding*
FileEncoding            Obsolete.  It still works and is equivalent
                        to |EncodingChanged|.
                                                *FileReadCmd*
FileReadCmd             Before reading a file with a ":read" command.
                        Should do the reading of the file. |Cmd-event|
                                                *FileReadPost*
FileReadPost            After reading a file with a ":read" command.
                        Note that Vim sets the '[ and '] marks to the
                        first and last line of the read.  This can be
                        used to operate on the lines just read.
                                                *FileReadPre*
FileReadPre             Before reading a file with a ":read" command.
                                                *FileType*
FileType                When the 'filetype' option has been set.  The
                        pattern is matched against the filetype.
                        <afile> can be used for the name of the file
                        where this option was set, and <amatch> for
                        the new value of 'filetype'.  Navigating to
                        another window or buffer is not allowed.
                        See |filetypes|.
                                                *FileWriteCmd*
FileWriteCmd            Before writing to a file, when not writing the
                        whole buffer.  Should do the writing to the
                        file.  Should not change the buffer.  Use the
                        '[ and '] marks for the range of lines.
                        |Cmd-event|
                                                *FileWritePost*
FileWritePost           After writing to a file, when not writing the
                        whole buffer.
                                                *FileWritePre*
FileWritePre            Before writing to a file, when not writing the
                        whole buffer.  Use the '[ and '] marks for the
                        range of lines.
                                                *FilterReadPost*
FilterReadPost          After reading a file from a filter command.
```

```
                                Vim checks the pattern against the name of
                                the current buffer as with FilterReadPre.
                                Not triggered when 'shelltemp' is off.
                                                        *FilterReadPre* *E135*
FilterReadPre                   Before reading a file from a filter command.
                                Vim checks the pattern against the name of
                                the current buffer, not the name of the
                                temporary file that is the output of the
                                filter command.
                                Not triggered when 'shelltemp' is off.
                                                        *FilterWritePost*
FilterWritePost                 After writing a file for a filter command or
                                making a diff.
                                Vim checks the pattern against the name of
                                the current buffer as with FilterWritePre.
                                Not triggered when 'shelltemp' is off.
                                                        *FilterWritePre*
FilterWritePre                  Before writing a file for a filter command or
                                making a diff.
                                Vim checks the pattern against the name of
                                the current buffer, not the name of the
                                temporary file that is the output of the
                                filter command.
                                Not triggered when 'shelltemp' is off.
                                                        *FocusGained*
FocusGained                     When Vim got input focus.  Only for the GUI
                                version and a few console versions where this
                                can be detected.
                                                        *FocusLost*
FocusLost                       When Vim lost input focus.  Only for the GUI
                                version and a few console versions where this
                                can be detected.  May also happen when a
                                dialog pops up.
                                                        *FuncUndefined*
FuncUndefined                   When a user function is used but it isn't
                                defined.  Useful for defining a function only
                                when it's used.  The pattern is matched
                                against the function name.  Both <amatch> and
                                <afile> are set to the name of the function.
                                NOTE: When writing Vim scripts a better
                                alternative is to use an autoloaded function.
                                See |autoload-functions|.
                                                        *GUIEnter*
GUIEnter                        After starting the GUI successfully, and after
                                opening the window.  It is triggered before
                                VimEnter when using gvim.  Can be used to
                                position the window from a .gvimrc file: >
        :autocmd GUIEnter * winpos 100 50
<                                                       *GUIFailed*
GUIFailed                       After starting the GUI failed.  Vim may
                                continue to run in the terminal, if possible
                                (only on Unix and alikes, when connecting the
                                X server fails).  You may want to quit Vim: >
        :autocmd GUIFailed * qall
<                                                       *InsertChange*
InsertChange                    When typing <Insert> while in Insert or
                                Replace mode.  The |v:insertmode| variable
                                indicates the new mode.
                                Be careful not to move the cursor or do
                                anything else that the user does not expect.
                                                        *InsertCharPre*
InsertCharPre                   When a character is typed in Insert mode,
```

```
                        before inserting the char.
                        The |v:char| variable indicates the char typed
                        and can be changed during the event to insert
                        a different character.  When |v:char| is set
                        to more than one character this text is
                        inserted literally.
                        It is not allowed to change the text |textlock|.
                        The event is not triggered when 'paste' is
                        set.
                                                *InsertEnter*
InsertEnter             Just before starting Insert mode.  Also for
                        Replace mode and Virtual Replace mode.  The
                        |v:insertmode| variable indicates the mode.
                        Be careful not to do anything else that the
                        user does not expect.
                        The cursor is restored afterwards.  If you do
                        not want that set |v:char| to a non-empty
                        string.
                                                *InsertLeave*
InsertLeave             When leaving Insert mode.  Also when using
                        CTRL-O |i_CTRL-O|.  But not for |i_CTRL-C|.
                                                *MenuPopup*
MenuPopup               Just before showing the popup menu (under the
                        right mouse button).  Useful for adjusting the
                        menu for what is under the cursor or mouse
                        pointer.
                        The pattern is matched against a single
                        character representing the mode:
                                n       Normal
                                v       Visual
                                o       Operator-pending
                                i       Insert
                                c       Command line
                                                *OptionSet*
OptionSet               After setting an option.  The pattern is
                        matched against the long option name.
                        The |v:option_old| variable indicates the
                        old option value, |v:option_new| variable
                        indicates the newly set value, the
                        |v:option_type| variable indicates whether
                        it's global or local scoped and |<amatch>|
                        indicates what option has been set.

                        Is not triggered on startup and for the 'key'
                        option for obvious reasons.

                        Usage example: Check for the existence of the
                        directory in the 'backupdir' and 'undodir'
                        options, create the directory if it doesn't
                        exist yet.

                        Note: It's a bad idea to reset an option
                        during this autocommand, this may break a
                        plugin. You can always use `:noa` to prevent
                        triggering this autocommand.

                                                *QuickFixCmdPre*
QuickFixCmdPre          Before a quickfix command is run (|:make|,
                        |:lmake|, |:grep|, |:lgrep|, |:grepadd|,
                        |:lgrepadd|, |:vimgrep|, |:lvimgrep|,
                        |:vimgrepadd|, |:lvimgrepadd|, |:cscope|,
                        |:cfile|, |:cgetfile|, |:caddfile|, |:lfile|,
```

|:lgetfile|, |:laddfile|, |:helpgrep|,
|:lhelpgrep|, |:cexpr|, |:cgetexpr|,
|:caddexpr|, |:cbuffer|, |:cgetbuffer|,
|:caddbuffer|).
The pattern is matched against the command
being run. When |:grep| is used but 'grepprg'
is set to "internal" it still matches "grep".
This command cannot be used to set the
'makeprg' and 'grepprg' variables.
If this command causes an error, the quickfix
command is not executed.

QuickFixCmdPost

QuickFixCmdPost
Like QuickFixCmdPre, but after a quickfix
command is run, before jumping to the first
location. For |:cfile| and |:lfile| commands
it is run after error file is read and before
moving to the first error.
See |QuickFixCmdPost-example|.

QuitPre

QuitPre
When using `:quit`, `:wq` or `:qall`, before
deciding whether it closes the current window
or quits Vim. Can be used to close any
non-essential window if the current window is
the last ordinary window.

RemoteReply

RemoteReply
When a reply from a Vim that functions as
server was received |server2client()|. The
pattern is matched against the {serverid}.
<amatch> is equal to the {serverid} from which
the reply was sent, and <afile> is the actual
reply string.
Note that even if an autocommand is defined,
the reply should be read with |remote_read()|
to consume it.

SessionLoadPost

SessionLoadPost
After loading the session file created using
the |:mksession| command.

ShellCmdPost

ShellCmdPost
After executing a shell command with |:!cmd|,
|:shell|, |:make| and |:grep|. Can be used to
check for any changed files.

ShellFilterPost

ShellFilterPost
After executing a shell command with
":{range}!cmd", ":w !cmd" or ":r !cmd".
Can be used to check for any changed files.

SourcePre

SourcePre
Before sourcing a Vim script. |:source|
<afile> is the name of the file being sourced.

SourceCmd

SourceCmd
When sourcing a Vim script. |:source|
<afile> is the name of the file being sourced.
The autocommand must source this file.
|Cmd-event|

SpellFileMissing

SpellFileMissing
When trying to load a spell checking file and
it can't be found. The pattern is matched
against the language. <amatch> is the
language, 'encoding' also matters. See
|spell-SpellFileMissing|.

StdinReadPost

StdinReadPost
After reading from the stdin into the buffer,
before executing the modelines. Only used

when the "-" argument was used when Vim was started |--|.
StdinReadPre

StdinReadPre Before reading from stdin into the buffer. Only used when the "-" argument was used when Vim was started |--|.
SwapExists

SwapExists Detected an existing swap file when starting to edit a file. Only when it is possible to select a way to handle the situation, when Vim would ask the user what to do.
The |v:swapname| variable holds the name of the swap file found, <afile> the file being edited. |v:swapcommand| may contain a command to be executed in the opened file.
The commands should set the |v:swapchoice| variable to a string with one character to tell Vim what should be done next:

```
'o'     open read-only
'e'     edit the file anyway
'r'     recover
'd'     delete the swap file
'q'     quit, don't edit the file
'a'     abort, like hitting CTRL-C
```

When set to an empty string the user will be asked, as if there was no SwapExists autocmd.
E812
It is not allowed to change to another buffer, change a buffer name or change directory here.
Syntax

Syntax When the 'syntax' option has been set. The pattern is matched against the syntax name. <afile> can be used for the name of the file where this option was set, and <amatch> for the new value of 'syntax'.
See |:syn-on|.
TabClosed

TabClosed After closing a tab page.
TabEnter

TabEnter Just after entering a tab page. |tab-page| After triggering the WinEnter and before triggering the BufEnter event.
TabLeave

TabLeave Just before leaving a tab page. |tab-page| A WinLeave event will have been triggered first.
TabNew

TabNew When a tab page was created. |tab-page| A WinEnter event will have been triggered first, TabEnter follows.
TermChanged

TermChanged After the value of 'term' has changed. Useful for re-loading the syntax file to update the colors, fonts and other terminal-dependent settings. Executed for all loaded buffers.
TermResponse

TermResponse After the response to |t_RV| is received from the terminal. The value of |v:termresponse| can be used to do things depending on the terminal version. Note that this event may be triggered halfway executing another event,

```
                                    especially if file I/O, a shell command or
                                    anything else that takes time is involved.
                                                                *TextChanged*
TextChanged                         After a change was made to the text in the
                                    current buffer in Normal mode.  That is when
                                    |b:changedtick| has changed.
                                    Not triggered when there is typeahead or when
                                    an operator is pending.
                                    Careful: This is triggered very often, don't
                                    do anything that the user does not expect or
                                    that is slow.
                                                                *TextChangedI*
TextChangedI                        After a change was made to the text in the
                                    current buffer in Insert mode.
                                    Not triggered when the popup menu is visible.
                                    Otherwise the same as TextChanged.
                                                                *User*
User                                Never executed automatically.  To be used for
                                    autocommands that are only executed with
                                    ":doautocmd".
                                                                *UserGettingBored*
UserGettingBored                    When the user presses the same key 42 times.
                                    Just kidding! :-)
                                                                *VimEnter*
VimEnter                            After doing all the startup stuff, including
                                    loading .vimrc files, executing the "-c cmd"
                                    arguments, creating all windows and loading
                                    the buffers in them.
                                    Just before this event is triggered the
                                    |v:vim_did_enter| variable is set, so that you
                                    can do: >
                                       if v:vim_did_enter
                                         call s:init()
                                       else
                                         au VimEnter * call s:init()
                                       endif
<                                                               *VimLeave*
VimLeave                            Before exiting Vim, just after writing the
                                    .viminfo file.  Executed only once, like
                                    VimLeavePre.
                                    To detect an abnormal exit use |v:dying|.
                                    When v:dying is 2 or more this event is not
                                    triggered.
                                                                *VimLeavePre*
VimLeavePre                         Before exiting Vim, just before writing the
                                    .viminfo file.  This is executed only once,
                                    if there is a match with the name of what
                                    happens to be the current buffer when exiting.
                                    Mostly useful with a "*" pattern. >
        :autocmd VimLeavePre * call CleanupStuff()
<                                   To detect an abnormal exit use |v:dying|.
                                    When v:dying is 2 or more this event is not
                                    triggered.
                                                                *VimResized*
VimResized                          After the Vim window was resized, thus 'lines'
                                    and/or 'columns' changed.  Not when starting
                                    up though.
                                                                *WinEnter*
WinEnter                            After entering another window.  Not done for
                                    the first window, when Vim has just started.
                                    Useful for setting the window height.
                                    If the window is for another buffer, Vim
```

executes the BufEnter autocommands after the
WinEnter autocommands.
Note: When using ":split fname" the WinEnter
event is triggered after the split but before
the file "fname" is loaded.

WinLeave

WinLeave Before leaving a window. If the window to be
entered next is for a different buffer, Vim
executes the BufLeave autocommands before the
WinLeave autocommands (but not for ":new").
Not used for ":qa" or ":q" when exiting Vim.

WinNew

WinNew When a new window was created. Not done for
the first window, when Vim has just started.
Before a WinEnter event.

==
6. Patterns *autocmd-patterns* *{pat}*

The {pat} argument can be a comma separated list. This works as if the
command was given with each pattern separately. Thus this command: >
:autocmd BufRead *.txt,*.info set et
Is equivalent to: >
:autocmd BufRead *.txt set et
:autocmd BufRead *.info set et

The file pattern {pat} is tested for a match against the file name in one of
two ways:
1. When there is no '/' in the pattern, Vim checks for a match against only
the tail part of the file name (without its leading directory path).
2. When there is a '/' in the pattern, Vim checks for a match against both the
short file name (as you typed it) and the full file name (after expanding
it to a full path and resolving symbolic links).

The special pattern <buffer> or <buffer=N> is used for buffer-local
autocommands |autocmd-buflocal|. This pattern is not matched against the name
of a buffer.

Examples: >
:autocmd BufRead *.txt set et
Set the 'et' option for all text files. >

:autocmd BufRead /vim/src/*.c set cindent
Set the 'cindent' option for C files in the /vim/src directory. >

:autocmd BufRead /tmp/*.c set ts=5
If you have a link from "/tmp/test.c" to "/home/nobody/vim/src/test.c", and
you start editing "/tmp/test.c", this autocommand will match.

Note: To match part of a path, but not from the root directory, use a '*' as
the first character. Example: >
:autocmd BufRead */doc/*.txt set tw=78
This autocommand will for example be executed for "/tmp/doc/xx.txt" and
"/usr/home/piet/doc/yy.txt". The number of directories does not matter here.

The file name that the pattern is matched against is after expanding
wildcards. Thus if you issue this command: >
:e $ROOTDIR/main.$EXT
The argument is first expanded to: >
/usr/root/main.py

```
Before it's matched with the pattern of the autocommand.  Careful with this
when using events like FileReadCmd, the value of <amatch> may not be what you
expect.


Environment variables can be used in a pattern: >
        :autocmd BufRead $VIMRUNTIME/doc/*.txt  set expandtab
And ~ can be used for the home directory (if $HOME is defined): >
        :autocmd BufWritePost ~/.vimrc   so ~/.vimrc
        :autocmd BufRead ~archive/*      set readonly
The environment variable is expanded when the autocommand is defined, not when
the autocommand is executed.  This is different from the command!

                                                            *file-pattern*
The pattern is interpreted like mostly used in file names:
        *       matches any sequence of characters; Unusual: includes path
                separators
        ?       matches any single character
        \?      matches a '?'
        .       matches a '.'
        ~       matches a '~'
        ,       separates patterns
        \,      matches a ','
        { }     like \( \) in a |pattern|
        ,       inside { }: like \| in a |pattern|
        \}      literal }
        \{      literal {
        \\\{n,m\}  like \{n,m} in a |pattern|
        \       special meaning like in a |pattern|
        [ch]    matches 'c' or 'h'
        [^ch]   match any character but 'c' and 'h'

Note that for all systems the '/' character is used for path separator (even
MS-DOS and OS/2).  This was done because the backslash is difficult to use
in a pattern and to make the autocommands portable across different systems.

It is possible to use |pattern| items, but they may not work as expected,
because of the translation done for the above.

                                                            *autocmd-changes*
Matching with the pattern is done when an event is triggered.  Changing the
buffer name in one of the autocommands, or even deleting the buffer, does not
change which autocommands will be executed.  Example: >

        au BufEnter *.foo  bdel
        au BufEnter *.foo  set modified

This will delete the current buffer and then set 'modified' in what has become
the current buffer instead.  Vim doesn't take into account that "*.foo"
doesn't match with that buffer name.  It matches "*.foo" with the name of the
buffer at the moment the event was triggered.

However, buffer-local autocommands will not be executed for a buffer that has
been wiped out with |:bwipe|.  After deleting the buffer with |:bdel| the
buffer actually still exists (it becomes unlisted), thus the autocommands are
still executed.

==============================================================================
7. Buffer-local autocommands    *autocmd-buflocal* *autocmd-buffer-local*
                                        *<buffer=N>* *<buffer=abuf>* *E680*

Buffer-local autocommands are attached to a specific buffer.  They are useful
```

```
if the buffer does not have a name and when the name does not match a specific
pattern.  But it also means they must be explicitly added to each buffer.

Instead of a pattern buffer-local autocommands use one of these forms:
        <buffer>        current buffer
        <buffer=99>     buffer number 99
        <buffer=abuf>   using <abuf> (only when executing autocommands)
                        |<abuf>|

Examples: >
    :au CursorHold <buffer>  echo 'hold'
    :au CursorHold <buffer=33>  echo 'hold'
    :au BufNewFile * au CursorHold <buffer=abuf>  echo 'hold'

All the commands for autocommands also work with buffer-local autocommands,
simply use the special string instead of the pattern.  Examples: >
    :au! * <buffer>                 " remove buffer-local autocommands for
                                    " current buffer
    :au! * <buffer=33>              " remove buffer-local autocommands for
                                    " buffer #33
    :bufdo :au! CursorHold <buffer>  " remove autocmd for given event for all
                                    " buffers
    :au * <buffer>                  " list buffer-local autocommands for
                                    " current buffer

Note that when an autocommand is defined for the current buffer, it is stored
with the buffer number.  Thus it uses the form "<buffer=12>", where 12 is the
number of the current buffer.  You will see this when listing autocommands,
for example.

To test for presence of buffer-local autocommands use the |exists()| function
as follows: >
    :if exists("#CursorHold#<buffer=12>") | ... | endif
    :if exists("#CursorHold#<buffer>") | ... | endif    " for current buffer

When a buffer is wiped out its buffer-local autocommands are also gone, of
course.  Note that when deleting a buffer, e.g., with ":bdel", it is only
unlisted, the autocommands are still present.  In order to see the removal of
buffer-local autocommands: >
    :set verbose=6

It is not possible to define buffer-local autocommands for a non-existent
buffer.

==============================================================================
8. Groups                                               *autocmd-groups*

Autocommands can be put together in a group.  This is useful for removing or
executing a group of autocommands.  For example, all the autocommands for
syntax highlighting are put in the "highlight" group, to be able to execute
":doautoall highlight BufRead" when the GUI starts.

When no specific group is selected, Vim uses the default group.  The default
group does not have a name.  You cannot execute the autocommands from the
default group separately; you can execute them only by executing autocommands
for all groups.

Normally, when executing autocommands automatically, Vim uses the autocommands
for all groups.  The group only matters when executing autocommands with
":doautocmd" or ":doautoall", or when defining or deleting autocommands.

The group name can contain any characters except white space.  The group name
```

"end" is reserved (also in uppercase).

The group name is case sensitive. Note that this is different from the event name!

:aug *:augroup*
:aug[roup] {name} Define the autocmd group name for the following ":autocmd" commands. The name "end" or "END" selects the default group.
To avoid confusion, the name should be different from existing {event} names, as this most likely will not do what you intended.

:augroup-delete *E367* *W19* *E936*
:aug[roup]! {name} Delete the autocmd group {name}. Don't use this if there is still an autocommand using this group! You will get a warning if doing it anyway. when the group is the current group you will get error E936.

To enter autocommands for a specific group, use this method:
1. Select the group with ":augroup {name}".
2. Delete any old autocommands with ":au!".
3. Define the autocommands.
4. Go back to the default group with "augroup END".

Example: >
```
:augroup uncompress
:  au!
:  au BufEnter *.gz     %!gunzip
:augroup END
```

This prevents having the autocommands defined twice (e.g., after sourcing the .vimrc file again).

==========
9. Executing autocommands *autocmd-execute*

Vim can also execute Autocommands non-automatically. This is useful if you have changed autocommands, or when Vim has executed the wrong autocommands (e.g., the file pattern match was wrong).

Note that the 'eventignore' option applies here too. Events listed in this option will not cause any commands to be executed.

:do *:doau* *:doautocmd* *E217*
:do[autocmd] [<nomodeline>] [group] {event} [fname]
Apply the autocommands matching [fname] (default: current file name) for {event} to the current buffer.
You can use this when the current file name does not match the right pattern, after changing settings, or to execute autocommands for a certain event.
It's possible to use this inside an autocommand too, so you can base the autocommands for one extension on another extension. Example: >
```
:au BufEnter *.cpp so ~/.vimrc_cpp
:au BufEnter *.cpp doau BufEnter x.c
```
<
Be careful to avoid endless loops. See |autocmd-nested|.

When the [group] argument is not given, Vim executes the autocommands for all groups. When the [group]

argument is included, Vim executes only the matching
autocommands for that group. Note: if you use an
undefined group name, Vim gives you an error message.
<nomodeline>
After applying the autocommands the modelines are
processed, so that their settings overrule the
settings from autocommands, like what happens when
editing a file. This is skipped when the <nomodeline>
argument is present. You probably want to use
<nomodeline> for events that are not used when loading
a buffer, such as |User|.
Processing modelines is also skipped when no
matching autocommands were executed.

:doautoa *:doautoall*
:doautoa[ll] [<nomodeline>] [group] {event} [fname]
Like ":doautocmd", but apply the autocommands to each
loaded buffer. Note that [fname] is used to select
the autocommands, not the buffers to which they are
applied.
Careful: Don't use this for autocommands that delete a
buffer, change to another buffer or change the
contents of a buffer; the result is unpredictable.
This command is intended for autocommands that set
options, change highlighting, and things like that.

==
10. Using autocommands *autocmd-use*

For WRITING FILES there are four possible sets of events. Vim uses only one
of these sets for a write command:

BufWriteCmd	BufWritePre	BufWritePost	writing the whole buffer
	FilterWritePre	FilterWritePost	writing to filter temp file
FileAppendCmd	FileAppendPre	FileAppendPost	appending to a file
FileWriteCmd	FileWritePre	FileWritePost	any other file write

When there is a matching "*Cmd" autocommand, it is assumed it will do the
writing. No further writing is done and the other events are not triggered.
|Cmd-event|

Note that the *WritePost commands should undo any changes to the buffer that
were caused by the *WritePre commands; otherwise, writing the file will have
the side effect of changing the buffer.

Before executing the autocommands, the buffer from which the lines are to be
written temporarily becomes the current buffer. Unless the autocommands
change the current buffer or delete the previously current buffer, the
previously current buffer is made the current buffer again.

The *WritePre and *AppendPre autocommands must not delete the buffer from
which the lines are to be written.

The '[and '] marks have a special position:
- Before the *ReadPre event the '[mark is set to the line just above where
  the new lines will be inserted.
- Before the *ReadPost event the '[mark is set to the first line that was
  just read, the '] mark to the last line.
- Before executing the *WriteCmd, *WritePre and *AppendPre autocommands the '[
  mark is set to the first line that will be written, the '] mark to the last
  line.
Careful: '[and '] change when using commands that change the buffer.

```
In commands which expect a file name, you can use "<afile>" for the file name
that is being read |:<afile>| (you can also use "%" for the current file
name).  "<abuf>" can be used for the buffer number of the currently effective
buffer.  This also works for buffers that doesn't have a name.  But it doesn't
work for files without a buffer (e.g., with ":r file").

                                                        *gzip-example*
Examples for reading and writing compressed files: >
  :augroup gzip
  :  autocmd!
  :  autocmd BufReadPre,FileReadPre     *.gz set bin
  :  autocmd BufReadPost,FileReadPost   *.gz '[,']!gunzip
  :  autocmd BufReadPost,FileReadPost   *.gz set nobin
  :  autocmd BufReadPost,FileReadPost   *.gz execute ":doautocmd BufReadPost " .
expand("%:r")
  :  autocmd BufWritePost,FileWritePost *.gz !mv <afile> <afile>:r
  :  autocmd BufWritePost,FileWritePost *.gz !gzip <afile>:r

  :  autocmd FileAppendPre              *.gz !gunzip <afile>
  :  autocmd FileAppendPre              *.gz !mv <afile>:r <afile>
  :  autocmd FileAppendPost             *.gz !mv <afile> <afile>:r
  :  autocmd FileAppendPost             *.gz !gzip <afile>:r
  :augroup END

The "gzip" group is used to be able to delete any existing autocommands with
":autocmd!", for when the file is sourced twice.

("<afile>:r" is the file name without the extension, see |:_%:|)

The commands executed for the BufNewFile, BufRead/BufReadPost, BufWritePost,
FileAppendPost and VimLeave events do not set or reset the changed flag of the
buffer.  When you decompress the buffer with the BufReadPost autocommands, you
can still exit with ":q".  When you use ":undo" in BufWritePost to undo the
changes made by BufWritePre commands, you can still do ":q" (this also makes
"ZZ" work).  If you do want the buffer to be marked as modified, set the
'modified' option.

To execute Normal mode commands from an autocommand, use the ":normal"
command.  Use with care!  If the Normal mode command is not finished, the user
needs to type characters (e.g., after ":normal m" you need to type a mark
name).

If you want the buffer to be unmodified after changing it, reset the
'modified' option.  This makes it possible to exit the buffer with ":q"
instead of ":q!".

                                                    *autocmd-nested* *E218*
By default, autocommands do not nest.  If you use ":e" or ":w" in an
autocommand, Vim does not execute the BufRead and BufWrite autocommands for
those commands.  If you do want this, use the "nested" flag for those commands
in which you want nesting.  For example: >
  :autocmd FileChangedShell *.c nested e!
The nesting is limited to 10 levels to get out of recursive loops.

It's possible to use the ":au" command in an autocommand.  This can be a
self-modifying command!  This can be useful for an autocommand that should
execute only once.

If you want to skip autocommands for one command, use the |:noautocmd| command
modifier or the 'eventignore' option.
```

```
Note: When reading a file (with ":read file" or with a filter command) and the
last line in the file does not have an <EOL>, Vim remembers this.  At the next
write (with ":write file" or with a filter command), if the same line is
written again as the last line in a file AND 'binary' is set, Vim does not
supply an <EOL>.  This makes a filter command on the just read lines write the
same file as was read, and makes a write command on just filtered lines write
the same file as was read from the filter.  For example, another way to write
a compressed file: >

  :autocmd FileWritePre *.gz   set bin|'[,']!gzip
  :autocmd FileWritePost *.gz  undo|set nobin
<
                                                        *autocommand-pattern*
You can specify multiple patterns, separated by commas.  Here are some
examples: >

  :autocmd BufRead   *          set tw=79 nocin ic infercase fo=2croq
  :autocmd BufRead   .letter    set tw=72 fo=2tcrq
  :autocmd BufEnter  .letter    set dict=/usr/lib/dict/words
  :autocmd BufLeave  .letter    set dict=
  :autocmd BufRead,BufNewFile   *.c,*.h set tw=0 cin noic
  :autocmd BufEnter  *.c,*.h    abbr FOR for (i = 0; i < 3; ++i)<CR>{<CR>}<Esc>O
  :autocmd BufLeave  *.c,*.h    unabbr FOR

For makefiles (makefile, Makefile, imakefile, makefile.unix, etc.): >

  :autocmd BufEnter  ?akefile*  set include=^s\=include
  :autocmd BufLeave  ?akefile*  set include&

To always start editing C files at the first function: >

  :autocmd BufRead   *.c,*.h    1;/^{

Without the "1;" above, the search would start from wherever the file was
entered, rather than from the start of the file.

                                                *skeleton* *template*
To read a skeleton (template) file when opening a new file: >

  :autocmd BufNewFile  *.c      0r ~/vim/skeleton.c
  :autocmd BufNewFile  *.h      0r ~/vim/skeleton.h
  :autocmd BufNewFile  *.java   0r ~/vim/skeleton.java

To insert the current date and time in a *.html file when writing it: >

  :autocmd BufWritePre,FileWritePre *.html   ks|call LastMod()|'s
  :fun LastMod()
  :  if line("$") > 20
  :    let l = 20
  :  else
  :    let l = line("$")
  :  endif
  :  exe "1," . l . "g/Last modified: /s/Last modified: .*/Last modified: " .
  :  \ strftime("%Y %b %d")
  :endfun

You need to have a line "Last modified: <date time>" in the first 20 lines
of the file for this to work.  Vim replaces <date time> (and anything in the
same line after it) with the current date and time.  Explanation:
        ks              mark current position with mark 's'
        call LastMod()  call the LastMod() function to do the work
        's              return the cursor to the old position
```

The LastMod() function checks if the file is shorter than 20 lines, and then
uses the ":g" command to find lines that contain "Last modified: ". For those
lines the ":s" command is executed to replace the existing date with the
current one. The ":execute" command is used to be able to use an expression
for the ":g" and ":s" commands. The date is obtained with the strftime()
function. You can change its argument to get another date string.

When entering :autocmd on the command-line, completion of events and command
names may be done (with <Tab>, CTRL-D, etc.) where appropriate.

Vim executes all matching autocommands in the order that you specify them.
It is recommended that your first autocommand be used for all files by using
"*" as the file pattern. This means that you can define defaults you like
here for any settings, and if there is another matching autocommand it will
override these. But if there is no other matching autocommand, then at least
your default settings are recovered (if entering this file from another for
which autocommands did match). Note that "*" will also match files starting
with ".", unlike Unix shells.

autocmd-searchpat

Autocommands do not change the current search patterns. Vim saves the current
search patterns before executing autocommands then restores them after the
autocommands finish. This means that autocommands do not affect the strings
highlighted with the 'hlsearch' option. Within autocommands, you can still
use search patterns normally, e.g., with the "n" command.
If you want an autocommand to set the search pattern, such that it is used
after the autocommand finishes, use the ":let @/ =" command.
The search-highlighting cannot be switched off with ":nohlsearch" in an
autocommand. Use the 'h' flag in the 'viminfo' option to disable search-
highlighting when starting Vim.

Cmd-event

When using one of the "*Cmd" events, the matching autocommands are expected to
do the file reading, writing or sourcing. This can be used when working with
a special kind of file, for example on a remote system.
CAREFUL: If you use these events in a wrong way, it may have the effect of
making it impossible to read or write the matching files! Make sure you test
your autocommands properly. Best is to use a pattern that will never match a
normal file name, for example "ftp://*".

When defining a BufReadCmd it will be difficult for Vim to recover a crashed
editing session. When recovering from the original file, Vim reads only those
parts of a file that are not found in the swap file. Since that is not
possible with a BufReadCmd, use the |:preserve| command to make sure the
original file isn't needed for recovery. You might want to do this only when
you expect the file to be modified.

For file read and write commands the |v:cmdarg| variable holds the "++enc="
and "++ff=" argument that are effective. These should be used for the command
that reads/writes the file. The |v:cmdbang| variable is one when "!" was
used, zero otherwise.

See the $VIMRUNTIME/plugin/netrwPlugin.vim for examples.

==
11. Disabling autocommands *autocmd-disable*

To disable autocommands for some time use the 'eventignore' option. Note that
this may cause unexpected behavior, make sure you restore 'eventignore'
afterwards, using a |:try| block with |:finally|.

:noautocmd *:noa*

```
To disable autocommands for just one command use the ":noautocmd" command
modifier.  This will set 'eventignore' to "all" for the duration of the
following command.  Example: >

        :noautocmd w fname.gz

This will write the file without triggering the autocommands defined by the
gzip plugin.


 vim:tw=78:ts=8:ft=help:norl:
*filetype.txt*  For Vim version 8.0.  Last change: 2017 Mar 28


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Filetypes                                                *filetype* *file-type*

1. Filetypes                                        |filetypes|
2. Filetype plugin                                  |filetype-plugins|
3. Docs for the default filetype plugins.           |ftplugin-docs|

Also see |autocmd.txt|.

{Vi does not have any of these commands}

==============================================================================
1. Filetypes                                       *filetypes* *file-types*

Vim can detect the type of file that is edited.  This is done by checking the
file name and sometimes by inspecting the contents of the file for specific
text.

                                                           *:filetype* *:filet*
To enable file type detection, use this command in your vimrc: >
        :filetype on
Each time a new or existing file is edited, Vim will try to recognize the type
of the file and set the 'filetype' option.  This will trigger the FileType
event, which can be used to set the syntax highlighting, set options, etc.

NOTE: Filetypes and 'compatible' don't work together well, since being Vi
compatible means options are global.  Resetting 'compatible' is recommended,
if you didn't do that already.

Detail: The ":filetype on" command will load one of these files:
                Amiga       $VIMRUNTIME/filetype.vim
                Mac         $VIMRUNTIME:filetype.vim
                MS-DOS      $VIMRUNTIME\filetype.vim
                RiscOS      Vim:Filetype
                Unix        $VIMRUNTIME/filetype.vim
                VMS         $VIMRUNTIME/filetype.vim
        This file is a Vim script that defines autocommands for the
        BufNewFile and BufRead events.  If the file type is not found by the
        name, the file $VIMRUNTIME/scripts.vim is used to detect it from the
        contents of the file.
        When the GUI is running or will start soon, the menu.vim script is
        also sourced.  See |'go-M'| about avoiding that.

To add your own file types, see |new-filetype| below.  To search for help on a
filetype prepend "ft-" and optionally append "-syntax", "-indent" or
"-plugin".  For example: >
```

```
        :help ft-vim-indent
        :help ft-vim-syntax
        :help ft-man-plugin

If the file type is not detected automatically, or it finds the wrong type,
you can either set the 'filetype' option manually, or add a modeline to your
file.  Example, for an IDL file use the command: >
        :set filetype=idl

or add this |modeline| to the file:
        /* vim: set filetype=idl : */ ~

                                                *:filetype-plugin-on*
You can enable loading the plugin files for specific file types with: >
        :filetype plugin on
If filetype detection was not switched on yet, it will be as well.
This actually loads the file "ftplugin.vim" in 'runtimepath'.
The result is that when a file is edited its plugin file is loaded (if there
is one for the detected filetype). |filetype-plugin|

                                                *:filetype-plugin-off*
You can disable it again with: >
        :filetype plugin off
The filetype detection is not switched off then.  But if you do switch off
filetype detection, the plugins will not be loaded either.
This actually loads the file "ftplugof.vim" in 'runtimepath'.

                                                *:filetype-indent-on*
You can enable loading the indent file for specific file types with: >
        :filetype indent on
If filetype detection was not switched on yet, it will be as well.
This actually loads the file "indent.vim" in 'runtimepath'.
The result is that when a file is edited its indent file is loaded (if there
is one for the detected filetype). |indent-expression|

                                                *:filetype-indent-off*
You can disable it again with: >
        :filetype indent off
The filetype detection is not switched off then.  But if you do switch off
filetype detection, the indent files will not be loaded either.
This actually loads the file "indoff.vim" in 'runtimepath'.
This disables auto-indenting for files you will open.  It will keep working in
already opened files.  Reset 'autoindent', 'cindent', 'smartindent' and/or
'indentexpr' to disable indenting in an opened file.

                                                *:filetype-off*
To disable file type detection, use this command: >
        :filetype off
This will keep the flags for "plugin" and "indent", but since no file types
are being detected, they won't work until the next ":filetype on".


Overview:                                       *:filetype-overview*

command                         detection       plugin          indent ~
:filetype on                    on              unchanged       unchanged
:filetype off                   off             unchanged       unchanged
:filetype plugin on             on              on              unchanged
:filetype plugin off            unchanged       off             unchanged
:filetype indent on             on              unchanged       on
:filetype indent off            unchanged       unchanged       off
:filetype plugin indent on      on              on              on
```

```
:filetype plugin indent off     unchanged       off             off

To see the current status, type: >
        :filetype
The output looks something like this: >
        filetype detection:ON  plugin:ON  indent:OFF

The file types are also used for syntax highlighting.  If the ":syntax on"
command is used, the file type detection is installed too.  There is no need
to do ":filetype on" after ":syntax on".

To disable one of the file types, add a line in your filetype file, see
|remove-filetype|.

                                                        *filetype-detect*
To detect the file type again: >
        :filetype detect
Use this if you started with an empty file and typed text that makes it
possible to detect the file type.  For example, when you entered this in a
shell script: "#!/bin/csh".
   When filetype detection was off, it will be enabled first, like the "on"
argument was used.

                                                        *filetype-overrule*
When the same extension is used for two filetypes, Vim tries to guess what
kind of file it is.  This doesn't always work.  A number of global variables
can be used to overrule the filetype used for certain extensions:

        file name       variable ~
        *.asa           g:filetype_asa  |ft-aspvbs-syntax| |ft-aspperl-syntax|
        *.asp           g:filetype_asp  |ft-aspvbs-syntax| |ft-aspperl-syntax|
        *.asm           g:asmsyntax     |ft-asm-syntax|
        *.prg           g:filetype_prg
        *.pl            g:filetype_pl
        *.inc           g:filetype_inc
        *.w             g:filetype_w    |ft-cweb-syntax|
        *.i             g:filetype_i    |ft-progress-syntax|
        *.p             g:filetype_p    |ft-pascal-syntax|
        *.sh            g:bash_is_sh    |ft-sh-syntax|
        *.tex           g:tex_flavor    |ft-tex-plugin|

                                                        *filetype-ignore*
To avoid that certain files are being inspected, the g:ft_ignore_pat variable
is used.  The default value is set like this: >
        :let g:ft_ignore_pat = '\.\(Z\|gz\|bz2\|zip\|tgz\)$'
This means that the contents of compressed files are not inspected.

                                                        *new-filetype*
If a file type that you want to use is not detected yet, there are four ways
to add it.  In any way, it's better not to modify the $VIMRUNTIME/filetype.vim
file.  It will be overwritten when installing a new version of Vim.

A. If you want to overrule all default file type checks.
   This works by writing one file for each filetype.  The disadvantage is that
   means there can be many files.  The advantage is that you can simply drop
   this file in the right directory to make it work.
                                                        *ftdetect*
   1. Create your user runtime directory.  You would normally use the first
      item of the 'runtimepath' option.  Then create the directory "ftdetect"
      inside it.  Example for Unix: >
        :!mkdir ~/.vim
        :!mkdir ~/.vim/ftdetect
```

```
<
   2. Create a file that contains an autocommand to detect the file type.
      Example: >
        au BufRead,BufNewFile *.mine            set filetype=mine
<     Note that there is no "augroup" command, this has already been done
      when sourcing your file.  You could also use the pattern "*" and then
      check the contents of the file to recognize it.
      Write this file as "mine.vim" in the "ftdetect" directory in your user
      runtime directory.  For example, for Unix: >
        :w ~/.vim/ftdetect/mine.vim

<  3. To use the new filetype detection you must restart Vim.

   The files in the "ftdetect" directory are used after all the default
   checks, thus they can overrule a previously detected file type.  But you
   can also use |:setfiletype| to keep a previously detected filetype.

B. If you want to detect your file after the default file type checks.

   This works like A above, but instead of setting 'filetype' unconditionally
   use ":setfiletype".  This will only set 'filetype' if no file type was
   detected yet.  Example: >
        au BufRead,BufNewFile *.txt             setfiletype text
<
   You can also use the already detected file type in your command.  For
   example, to use the file type "mypascal" when "pascal" has been detected: >
        au BufRead,BufNewFile *         if &ft == 'pascal' | set ft=mypascal
                                                                   | endif

C. If your file type can be detected by the file name.
   1. Create your user runtime directory.  You would normally use the first
      item of the 'runtimepath' option.  Example for Unix: >
        :!mkdir ~/.vim
<
   2. Create a file that contains autocommands to detect the file type.
      Example: >
        " my filetype file
        if exists("did_load_filetypes")
          finish
        endif
        augroup filetypedetect
          au! BufRead,BufNewFile *.mine         setfiletype mine
          au! BufRead,BufNewFile *.xyz          setfiletype drawing
        augroup END
<     Write this file as "filetype.vim" in your user runtime directory.  For
      example, for Unix: >
        :w ~/.vim/filetype.vim

<  3. To use the new filetype detection you must restart Vim.

   Your filetype.vim will be sourced before the default FileType autocommands
   have been installed.  Your autocommands will match first, and the
   ":setfiletype" command will make sure that no other autocommands will set
   'filetype' after this.
                                                        *new-filetype-scripts*
D. If your filetype can only be detected by inspecting the contents of the
   file.

   1. Create your user runtime directory.  You would normally use the first
      item of the 'runtimepath' option.  Example for Unix: >
        :!mkdir ~/.vim
<
```

```
   2. Create a vim script file for doing this.  Example: >
        if did_filetype()       " filetype already set..
          finish                " ..don't do these checks
        endif
        if getline(1) =~ '^#!.*\<mine\>'
          setfiletype mine
        elseif getline(1) =~? '\<drawing\>'
          setfiletype drawing
        endif
<     See $VIMRUNTIME/scripts.vim for more examples.
      Write this file as "scripts.vim" in your user runtime directory.  For
      example, for Unix: >
        :w ~/.vim/scripts.vim
<
   3. The detection will work right away, no need to restart Vim.

   Your scripts.vim is loaded before the default checks for file types, which
   means that your rules override the default rules in
   $VIMRUNTIME/scripts.vim.

                                                    *remove-filetype*
If a file type is detected that is wrong for you, install a filetype.vim or
scripts.vim to catch it (see above).  You can set 'filetype' to a non-existing
name to avoid that it will be set later anyway: >
        :set filetype=ignored

If you are setting up a system with many users, and you don't want each user
to add/remove the same filetypes, consider writing the filetype.vim and
scripts.vim files in a runtime directory that is used for everybody.  Check
the 'runtimepath' for a directory to use.  If there isn't one, set
'runtimepath' in the |system-vimrc|.  Be careful to keep the default
directories!


                                                    *autocmd-osfiletypes*
NOTE: this code is currently disabled, as the RISC OS implementation was
removed.  In the future this will use the 'filetype' option.

On operating systems which support storing a file type with the file, you can
specify that an autocommand should only be executed if the file is of a
certain type.

The actual type checking depends on which platform you are running Vim
on; see your system's documentation for details.

To use osfiletype checking in an autocommand you should put a list of types to
match in angle brackets in place of a pattern, like this: >

        :au BufRead *.html,<&faf;HTML>  runtime! syntax/html.vim

This will match:

- Any file whose name ends in ".html"
- Any file whose type is "&faf" or "HTML", where the meaning of these types
  depends on which version of Vim you are using.
  Unknown types are considered NOT to match.

You can also specify a type and a pattern at the same time (in which case they
must both match): >

        :au BufRead <&fff>diff*
```

```
This will match files of type "&fff" whose names start with "diff".


                                                        *plugin-details*
The "plugin" directory can be in any of the directories in the 'runtimepath'
option.  All of these directories will be searched for plugins and they are
all loaded.  For example, if this command: >

        set runtimepath

produces this output:

        runtimepath=/etc/vim,~/.vim,/usr/local/share/vim/vim60 ~

then Vim will load all plugins in these directories and below:

        /etc/vim/plugin/  ~
        ~/.vim/plugin/  ~
        /usr/local/share/vim/vim60/plugin/  ~

Note that the last one is the value of $VIMRUNTIME which has been expanded.

What if it looks like your plugin is not being loaded?  You can find out what
happens when Vim starts up by using the |-V| argument: >

        vim -V2

You will see a lot of messages, in between them is a remark about loading the
plugins.  It starts with:

        Searching for "plugin/**/*.vim" in ~

There you can see where Vim looks for your plugin scripts.

==============================================================================
2. Filetype plugin                                      *filetype-plugins*

When loading filetype plugins has been enabled |:filetype-plugin-on|, options
will be set and mappings defined.  These are all local to the buffer, they
will not be used for other files.

Defining mappings for a filetype may get in the way of the mappings you
define yourself.  There are a few ways to avoid this:
1. Set the "maplocalleader" variable to the key sequence you want the mappings
   to start with.  Example: >
        :let maplocalleader = ","
<  All mappings will then start with a comma instead of the default, which
   is a backslash.  Also see |<LocalLeader>|.

2. Define your own mapping.  Example: >
        :map ,p <Plug>MailQuote
<  You need to check the description of the plugin file below for the
   functionality it offers and the string to map to.
   You need to define your own mapping before the plugin is loaded (before
   editing a file of that type).  The plugin will then skip installing the
   default mapping.
                                                *no_mail_maps*
3. Disable defining mappings for a specific filetype by setting a variable,
   which contains the name of the filetype.  For the "mail" filetype this
   would be: >
        :let no_mail_maps = 1
<                                               *no_plugin_maps*
```

```
4. Disable defining mappings for all filetypes by setting a variable: >
        :let no_plugin_maps = 1
<

                                                        *ftplugin-overrule*
If a global filetype plugin does not do exactly what you want, there are three
ways to change this:

1. Add a few settings.
   You must create a new filetype plugin in a directory early in
   'runtimepath'.  For Unix, for example you could use this file: >
        vim ~/.vim/ftplugin/fortran.vim
<  You can set those settings and mappings that you would like to add.  Note
   that the global plugin will be loaded after this, it may overrule the
   settings that you do here.  If this is the case, you need to use one of the
   following two methods.

2. Make a copy of the plugin and change it.
   You must put the copy in a directory early in 'runtimepath'.  For Unix, for
   example, you could do this: >
        cp $VIMRUNTIME/ftplugin/fortran.vim ~/.vim/ftplugin/fortran.vim
<  Then you can edit the copied file to your liking.  Since the b:did_ftplugin
   variable will be set, the global plugin will not be loaded.
   A disadvantage of this method is that when the distributed plugin gets
   improved, you will have to copy and modify it again.

3. Overrule the settings after loading the global plugin.
   You must create a new filetype plugin in a directory from the end of
   'runtimepath'.  For Unix, for example, you could use this file: >
        vim ~/.vim/after/ftplugin/fortran.vim
<  In this file you can change just those settings that you want to change.

==============================================================================
3.  Docs for the default filetype plugins.              *ftplugin-docs*


CHANGELOG                                               *ft-changelog-plugin*

Allows for easy entrance of Changelog entries in Changelog files.  There are
some commands, mappings, and variables worth exploring:

Options:
'comments'              is made empty to not mess up formatting.
'textwidth'             is set to 78, which is standard.
'formatoptions'         the 't' flag is added to wrap when inserting text.

Commands:
NewChangelogEntry       Adds a new Changelog entry in an intelligent fashion
                        (see below).

Local mappings:
<Leader>o               Starts a new Changelog entry in an equally intelligent
                        fashion (see below).

Global mappings:
                        NOTE: The global mappings are accessed by sourcing the
                        ftplugin/changelog.vim file first, e.g. with >
                                runtime ftplugin/changelog.vim
<                       in your |.vimrc|.
<Leader>o               Switches to the ChangeLog buffer opened for the
                        current directory, or opens it in a new buffer if it
                        exists in the current directory.  Then it does the
```

```
                        same as the local <Leader>o described above.

Variables:
g:changelog_timeformat  Deprecated; use g:changelog_dateformat instead.
g:changelog_dateformat  The date (and time) format used in ChangeLog entries.
                        The format accepted is the same as for the
                        |strftime()| function.
                        The default is "%Y-%m-%d" which is the standard format
                        for many ChangeLog layouts.
g:changelog_username    The name and email address of the user.
                        The default is deduced from environment variables and
                        system files.  It searches /etc/passwd for the comment
                        part of the current user, which informally contains
                        the real name of the user up to the first separating
                        comma.  then it checks the $NAME environment variable
                        and finally runs `whoami` and `hostname` to build an
                        email address.  The final form is >
                                Full Name  <user@host>
<
g:changelog_new_date_format
                        The format to use when creating a new date-entry.
                        The following table describes special tokens in the
                        string:
                                %%      insert a single '%' character
                                %d      insert the date from above
                                %u      insert the user from above
                                %p      insert result of b:changelog_entry_prefix
                                %c      where to position cursor when done
                        The default is "%d  %u\n\n\t* %p%c\n\n", which produces
                        something like (| is where cursor will be, unless at
                        the start of the line where it denotes the beginning
                        of the line) >
                                |2003-01-14  Full Name  <user@host>
                                |
                                |        * prefix|
<
g:changelog_new_entry_format
                        The format used when creating a new entry.
                        The following table describes special tokens in the
                        string:
                                %p      insert result of b:changelog_entry_prefix
                                %c      where to position cursor when done
                        The default is "\t*%c", which produces something
                        similar to >
                                |        * prefix|
<
g:changelog_date_entry_search
                        The search pattern to use when searching for a
                        date-entry.
                        The same tokens that can be used for
                        g:changelog_new_date_format can be used here as well.
                        The default is '^\s*%d\_s*%u' which finds lines
                        matching the form >
                                |2003-01-14  Full Name  <user@host>
<                       and some similar formats.

g:changelog_date_end_entry_search
                        The search pattern to use when searching for the end
                        of a date-entry.
                        The same tokens that can be used for
                        g:changelog_new_date_format can be used here as well.
                        The default is '^\s*$' which finds lines that contain
```

only whitespace or are completely empty.

b:changelog_name *b:changelog_name*
Name of the ChangeLog file to look for.
The default is 'ChangeLog'.

b:changelog_path
Path of the ChangeLog to use for the current buffer.
The default is empty, thus looking for a file named
|b:changelog_name| in the same directory as the
current buffer. If not found, the parent directory of
the current buffer is searched. This continues
recursively until a file is found or there are no more
parent directories to search.

b:changelog_entry_prefix
Name of a function to call to generate a prefix to a
new entry. This function takes no arguments and
should return a string containing the prefix.
Returning an empty prefix is fine.
The default generates the shortest path between the
ChangeLog's pathname and the current buffers pathname.
In the future, it will also be possible to use other
variable contexts for this variable, for example, g:.

The Changelog entries are inserted where they add the least amount of text.
After figuring out the current date and user, the file is searched for an
entry beginning with the current date and user and if found adds another item
under it. If not found, a new entry and item is prepended to the beginning of
the Changelog.

FORTRAN *ft-fortran-plugin*

Options:
'expandtab' is switched on to avoid tabs as required by the Fortran
standards unless the user has set fortran_have_tabs in .vimrc.
'textwidth' is set to 72 for fixed source format as required by the
Fortran standards and to 80 for free source format.
'formatoptions' is set to break code and comment lines and to preserve long
lines. You can format comments with |gq|.
For further discussion of fortran_have_tabs and the method used for the
detection of source format see |ft-fortran-syntax|.

GIT COMMIT *ft-gitcommit-plugin*

One command, :DiffGitCached, is provided to show a diff of the current commit
in the preview window. It is equivalent to calling "git diff --cached" plus
any arguments given to the command.

MAIL *ft-mail-plugin*

Options:
'modeline' is switched off to avoid the danger of trojan horses, and to
avoid that a Subject line with "Vim:" in it will cause an
error message.
'textwidth' is set to 72. This is often recommended for e-mail.
'formatoptions' is set to break text lines and to repeat the comment leader
in new lines, so that a leading ">" for quotes is repeated.
You can also format quoted text with |gq|.

```
Local mappings:
<LocalLeader>q   or   \\MailQuote
        Quotes the text selected in Visual mode, or from the cursor position
        to the end of the file in Normal mode.  This means "> " is inserted in
        each line.

MAN                                      *ft-man-plugin* *:Man* *man.vim*

Displays a manual page in a nice way.  Also see the user manual
|find-manpage|.

To start using the ":Man" command before any manual page was loaded, source
this script from your startup vimrc file: >

        runtime ftplugin/man.vim

Options:
'iskeyword'     the '.' character is added to be able to use CTRL-] on the
                manual page name.

Commands:
Man {name}      Display the manual page for {name} in a window.
Man {number} {name}
                Display the manual page for {name} in a section {number}.

Global mapping:
<Leader>K       Displays the manual page for the word under the cursor.
<Plug>ManPreGetPage  idem, allows for using a mapping: >
                        nmap <F1> <Plug>ManPreGetPage<CR>

Local mappings:
CTRL-]          Jump to the manual page for the word under the cursor.
CTRL-T          Jump back to the previous manual page.
q               Same as ":quit"

To use a vertical split instead of horizontal: >
        let g:ft_man_open_mode = 'vert'
To use a new tab: >
        let g:ft_man_open_mode = 'tab'

To enable folding use this: >
        let g:ft_man_folding_enable = 1
If you do not like the default folding, use an autocommand to add your desired
folding style instead.  For example: >
        autocmd FileType man setlocal foldmethod=indent foldenable

You may also want to set 'keywordprg' to make the |K| command open a manual
page in a Vim window: >
        set keywordprg=:Man


MANPAGER                                          *manpager.vim*

The :Man command allows you to turn Vim into a manpager (that syntax highlights
manpages and follows linked manpages on hitting CTRL-]).

Works on:

  - Linux
  - Mac OS
  - FreeBSD
```

```
  - Cygwin
  - Win 10 under Bash

Untested:

  - Amiga OS
  - BeOS
  - OS/2

For bash,zsh,ksh or dash by adding to the config file (.bashrc,.zshrc, ...)

        export MANPAGER="env MAN_PN=1 vim -M +MANPAGER -"

For (t)csh by adding to the config file

        setenv MANPAGER "env MAN_PN=1 vim -M +MANPAGER -"

For fish by adding to the config file

        set -x MANPAGER "env MAN_PN=1 vim -M +MANPAGER -"

If man sets the $MAN_PN environment variable, like man-db, the most common
implementation on Linux and Mac OS, then the "env MAN_PN=1 " part above is
superfluous.

PDF                                                     *ft-pdf-plugin*

Two maps, <C-]> and <C-T>, are provided to simulate a tag stack for navigating
the PDF.  The following are treated as tags:

- The byte offset after "startxref" to the xref table
- The byte offset after the /Prev key in the trailer to an earlier xref table
- A line of the form "0123456789 00000 n" in the xref table
- An object reference like "1 0 R" anywhere in the PDF

These maps can be disabled with >
        :let g:no_pdf_maps = 1
<

PYTHON                                          *ft-python-plugin* *PEP8*

By default the following options are set, in accordance with PEP8: >

        setlocal expandtab shiftwidth=4 softtabstop=4 tabstop=8

To disable this behaviour, set the following variable in your vimrc: >

        let g:python_recommended_style = 0


RPM SPEC                                                *ft-spec-plugin*

Since the text for this plugin is rather long it has been put in a separate
file: |pi_spec.txt|.


RUST                                                    *ft-rust*

Since the text for this plugin is rather long it has been put in a separate
file: |ft_rust.txt|.
```

SQL *ft-sql*

Since the text for this plugin is rather long it has been put in a separate
file: |ft_sql.txt|.

TEX *ft-tex-plugin* *g:tex_flavor*

If the first line of a *.tex file has the form >
	%&<format>
then this determined the file type: plaintex (for plain TeX), context (for
ConTeXt), or tex (for LaTeX). Otherwise, the file is searched for keywords to
choose context or tex. If no keywords are found, it defaults to plaintex.
You can change the default by defining the variable g:tex_flavor to the format
(not the file type) you use most. Use one of these: >
	let g:tex_flavor = "plain"
	let g:tex_flavor = "context"
	let g:tex_flavor = "latex"
Currently no other formats are recognized.

vim:tw=78:ts=8:ft=help:norl:
eval.txt For Vim version 8.0. Last change: 2017 Sep 17

VIM REFERENCE MANUAL by Bram Moolenaar

Expression evaluation *expression* *expr* *E15* *eval*

Using expressions is introduced in chapter 41 of the user manual |usr_41.txt|.

Note: Expression evaluation can be disabled at compile time. If this has been
done, the features in this document are not available. See |+eval| and
|no-eval-feature|.

{Vi does not have any of these commands}

==
1. Variables *variables*

1.1 Variable types ~
E712

```
There are nine types of variables:

Number          A 32 or 64 bit signed number.  |expr-number| *Number*
                64-bit Numbers are available only when compiled with the
                |+num64| feature.
                Examples:  -123  0x10  0177  0b1011

Float           A floating point number. |floating-point-format| *Float*
                {only when compiled with the |+float| feature}
                Examples: 123.456  1.15e-6  -1.1e3

                                                        *E928*
String          A NUL terminated string of 8-bit unsigned characters (bytes).
                |expr-string| Examples: "ab\txx\"--"  'x-z''a,c'

List            An ordered sequence of items |List|.
                Example: [1, 2, ['a', 'b']]

Dictionary      An associative, unordered array: Each entry has a key and a
                value. |Dictionary|
                Example: {'blue': "#0000ff", 'red': "#ff0000"}

Funcref         A reference to a function |Funcref|.
                Example: function("strlen")
                It can be bound to a dictionary and arguments, it then works
                like a Partial.
                Example: function("Callback", [arg], myDict)

Special         |v:false|, |v:true|, |v:none| and |v:null|.  *Special*

Job             Used for a job, see |job_start()|. *Job* *Jobs*

Channel         Used for a channel, see |ch_open()|. *Channel* *Channels*

The Number and String types are converted automatically, depending on how they
are used.

Conversion from a Number to a String is by making the ASCII representation of
the Number.  Examples:
        Number 123      -->     String "123" ~
        Number 0        -->     String "0" ~
        Number -1       -->     String "-1" ~
                                                        *octal*
Conversion from a String to a Number is done by converting the first digits to
a number.  Hexadecimal "0xf9", Octal "017", and Binary "0b10" numbers are
recognized.  If the String doesn't start with digits, the result is zero.
Examples:
        String "456"    -->     Number 456 ~
        String "6bar"   -->     Number 6 ~
        String "foo"    -->     Number 0 ~
        String "0xf1"   -->     Number 241 ~
        String "0100"   -->     Number 64 ~
        String "0b101"  -->     Number 5 ~
        String "-8"     -->     Number -8 ~
        String "+8"     -->     Number 0 ~

To force conversion from String to Number, add zero to it: >
        :echo "0100" + 0
<       64 ~

To avoid a leading zero to cause octal conversion, or for using a different
base, use |str2nr()|.
```

TRUE *FALSE*
For boolean operators Numbers are used. Zero is FALSE, non-zero is TRUE.
You can also use |v:false| and |v:true|. When TRUE is returned from a
function it is the Number one, FALSE is the number zero.

Note that in the command: >
:if "foo"
:" NOT executed
"foo" is converted to 0, which means FALSE. If the string starts with a
non-zero number it means TRUE: >
:if "8foo"
:" executed
To test for a non-empty string, use empty(): >
:if !empty("foo")
<
non-zero-arg
Function arguments often behave slightly different from |TRUE|: If the
argument is present and it evaluates to a non-zero Number, |v:true| or a
non-empty String, then the value is considered to be TRUE.
Note that " " and "0" are also non-empty strings, thus cause the mode to be
cleared. A List, Dictionary or Float is not a Number or String, thus
evaluates to FALSE.

E745 *E728* *E703* *E729* *E730* *E731* *E908* *E910* *E913*
List, Dictionary, Funcref, Job and Channel types are not automatically
converted.

E805 *E806* *E808*
When mixing Number and Float the Number is converted to Float. Otherwise
there is no automatic conversion of Float. You can use str2float() for String
to Float, printf() for Float to String and float2nr() for Float to Number.

E891 *E892* *E893* *E894* *E907* *E911* *E914*
When expecting a Float a Number can also be used, but nothing else.

no-type-checking
You will not get an error if you try to change the type of a variable.

1.2 Function references ~
Funcref *E695* *E718*
A Funcref variable is obtained with the |function()| function, the |funcref()|
function or created with the lambda expression |expr-lambda|. It can be used
in an expression in the place of a function name, before the parenthesis
around the arguments, to invoke the function it refers to. Example: >

:let Fn = function("MyFunc")
:echo Fn()
<
E704 *E705* *E707*
A Funcref variable must start with a capital, "s:", "w:", "t:" or "b:". You
can use "g:" but the following name must still start with a capital. You
cannot have both a Funcref variable and a function with the same name.

A special case is defining a function and directly assigning its Funcref to a
Dictionary entry. Example: >
:function dict.init() dict
: let self.val = 0
:endfunction

The key of the Dictionary can start with a lower case letter. The actual
function name is not used here. Also see |numbered-function|.

A Funcref can also be used with the |:call| command: >
	:call Fn()
	:call dict.init()

The name of the referenced function can be obtained with |string()|. >
	:let func = string(Fn)

You can use |call()| to invoke a Funcref and use a list variable for the
arguments: >
	:let r = call(Fn, mylist)
<
Partial
A Funcref optionally binds a Dictionary and/or arguments. This is also called
a Partial. This is created by passing the Dictionary and/or arguments to
function() or funcref(). When calling the function the Dictionary and/or
arguments will be passed to the function. Example: >

	let Cb = function('Callback', ['foo'], myDict)
	call Cb()

This will invoke the function as if using: >
	call myDict.Callback('foo')

This is very useful when passing a function around, e.g. in the arguments of
|ch_open()|.

Note that binding a function to a Dictionary also happens when the function is
a member of the Dictionary: >

	let myDict.myFunction = MyFunction
	call myDict.myFunction()

Here MyFunction() will get myDict passed as "self". This happens when the
"myFunction" member is accessed. When making assigning "myFunction" to
otherDict and calling it, it will be bound to otherDict: >

	let otherDict.myFunction = myDict.myFunction
	call otherDict.myFunction()

Now "self" will be "otherDict". But when the dictionary was bound explicitly
this won't happen: >

	let myDict.myFunction = function(MyFunction, myDict)
	let otherDict.myFunction = myDict.myFunction
	call otherDict.myFunction()

Here "self" will be "myDict", because it was bound explicitly.

1.3 Lists ~

list *List* *Lists* *E686*
A List is an ordered sequence of items. An item can be of any type. Items
can be accessed by their index number. Items can be added and removed at any
position in the sequence.

List creation ~

E696 *E697*
A List is created with a comma separated list of items in square brackets.
Examples: >
	:let mylist = [1, two, 3, "four"]

```
        :let emptylist = []

An item can be any expression.  Using a List for an item creates a
List of Lists: >
        :let nestlist = [[11, 12], [21, 22], [31, 32]]

An extra comma after the last item is ignored.


List index ~
                                                        *list-index* *E684*
An item in the List can be accessed by putting the index in square brackets
after the List.  Indexes are zero-based, thus the first item has index zero. >
        :let item = mylist[0]           " get the first item: 1
        :let item = mylist[2]           " get the third item: 3

When the resulting item is a list this can be repeated: >
        :let item = nestlist[0][1]      " get the first list, second item: 12
<
A negative index is counted from the end.  Index -1 refers to the last item in
the List, -2 to the last but one item, etc. >
        :let last = mylist[-1]          " get the last item: "four"

To avoid an error for an invalid index use the |get()| function.  When an item
is not available it returns zero or the default value you specify: >
        :echo get(mylist, idx)
        :echo get(mylist, idx, "NONE")


List concatenation ~

Two lists can be concatenated with the "+" operator: >
        :let longlist = mylist + [5, 6]
        :let mylist += [7, 8]

To prepend or append an item turn the item into a list by putting [] around
it.  To change a list in-place see |list-modification| below.


Sublist ~
                                                        *sublist*
A part of the List can be obtained by specifying the first and last index,
separated by a colon in square brackets: >
        :let shortlist = mylist[2:-1]   " get List [3, "four"]

Omitting the first index is similar to zero.  Omitting the last index is
similar to -1. >
        :let endlist = mylist[2:]       " from item 2 to the end: [3, "four"]
        :let shortlist = mylist[2:2]    " List with one item: [3]
        :let otherlist = mylist[:]      " make a copy of the List

If the first index is beyond the last item of the List or the second item is
before the first item, the result is an empty list.  There is no error
message.

If the second index is equal to or greater than the length of the list the
length minus one is used: >
        :let mylist = [0, 1, 2, 3]
        :echo mylist[2:8]               " result: [2, 3]

NOTE: mylist[s:e] means using the variable "s:e" as index.  Watch out for
using a single letter variable before the ":".  Insert a space when needed:
```

```
mylist[s : e].


List identity ~
                                                              *list-identity*
When variable "aa" is a list and you assign it to another variable "bb", both
variables refer to the same list.  Thus changing the list "aa" will also
change "bb": >
        :let aa = [1, 2, 3]
        :let bb = aa
        :call add(aa, 4)
        :echo bb
<       [1, 2, 3, 4]

Making a copy of a list is done with the |copy()| function.  Using [:] also
works, as explained above.  This creates a shallow copy of the list: Changing
a list item in the list will also change the item in the copied list: >
        :let aa = [[1, 'a'], 2, 3]
        :let bb = copy(aa)
        :call add(aa, 4)
        :let aa[0][1] = 'aaa'
        :echo aa
<       [[1, aaa], 2, 3, 4] >
        :echo bb
<       [[1, aaa], 2, 3]

To make a completely independent list use |deepcopy()|.  This also makes a
copy of the values in the list, recursively.  Up to a hundred levels deep.

The operator "is" can be used to check if two variables refer to the same
List.  "isnot" does the opposite.  In contrast "==" compares if two lists have
the same value. >
        :let alist = [1, 2, 3]
        :let blist = [1, 2, 3]
        :echo alist is blist
<       0 >
        :echo alist == blist
<       1

Note about comparing lists: Two lists are considered equal if they have the
same length and all items compare equal, as with using "==".  There is one
exception: When comparing a number with a string they are considered
different.  There is no automatic type conversion, as with using "==" on
variables.  Example: >
        echo 4 == "4"
<       1 >
        echo [4] == ["4"]
<       0

Thus comparing Lists is more strict than comparing numbers and strings.  You
can compare simple values this way too by putting them in a list: >

        :let a = 5
        :let b = "5"
        :echo a == b
<       1 >
        :echo [a] == [b]
<       0


List unpack ~
```

```
To unpack the items in a list to individual variables, put the variables in
square brackets, like list items: >
        :let [var1, var2] = mylist

When the number of variables does not match the number of items in the list
this produces an error.  To handle any extra items from the list append ";"
and a variable name: >
        :let [var1, var2; rest] = mylist

This works like: >
        :let var1 = mylist[0]
        :let var2 = mylist[1]
        :let rest = mylist[2:]

Except that there is no error if there are only two items.  "rest" will be an
empty list then.


List modification ~
                                                        *list-modification*
To change a specific item of a list use |:let| this way: >
        :let list[4] = "four"
        :let listlist[0][3] = item

To change part of a list you can specify the first and last item to be
modified.  The value must at least have the number of items in the range: >
        :let list[3:5] = [3, 4, 5]

Adding and removing items from a list is done with functions.  Here are a few
examples: >
        :call insert(list, 'a')         " prepend item 'a'
        :call insert(list, 'a', 3)      " insert item 'a' before list[3]
        :call add(list, "new")          " append String item
        :call add(list, [1, 2])         " append a List as one new item
        :call extend(list, [1, 2])      " extend the list with two more items
        :let i = remove(list, 3)        " remove item 3
        :unlet list[3]                  " idem
        :let l = remove(list, 3, -1)    " remove items 3 to last item
        :unlet list[3 : ]               " idem
        :call filter(list, 'v:val !~ "x"')  " remove items with an 'x'

Changing the order of items in a list: >
        :call sort(list)                " sort a list alphabetically
        :call reverse(list)             " reverse the order of items
        :call uniq(sort(list))          " sort and remove duplicates


For loop ~

The |:for| loop executes commands for each item in a list.  A variable is set
to each item in the list in sequence.  Example: >
        :for item in mylist
        :   call Doit(item)
        :endfor

This works like: >
        :let index = 0
        :while index < len(mylist)
        :   let item = mylist[index]
        :   :call Doit(item)
        :   let index = index + 1
        :endwhile
```

```
If all you want to do is modify each item in the list then the |map()|
function will be a simpler method than a for loop.

Just like the |:let| command, |:for| also accepts a list of variables.  This
requires the argument to be a list of lists. >
        :for [lnum, col] in [[1, 3], [2, 8], [3, 0]]
        :   call Doit(lnum, col)
        :endfor

This works like a |:let| command is done for each list item.  Again, the types
must remain the same to avoid an error.

It is also possible to put remaining items in a List variable: >
        :for [i, j; rest] in listlist
        :   call Doit(i, j)
        :   if !empty(rest)
        :      echo "remainder: " . string(rest)
        :   endif
        :endfor


List functions ~
                                                *E714*
Functions that are useful with a List: >
        :let r = call(funcname, list)   " call a function with an argument list
        :if empty(list)                 " check if list is empty
        :let l = len(list)              " number of items in list
        :let big = max(list)            " maximum value in list
        :let small = min(list)          " minimum value in list
        :let xs = count(list, 'x')      " count nr of times 'x' appears in list
        :let i = index(list, 'x')       " index of first 'x' in list
        :let lines = getline(1, 10)     " get ten text lines from buffer
        :call append('$', lines)        " append text lines in buffer
        :let list = split("a b c")      " create list from items in a string
        :let string = join(list, ', ')  " create string from list items
        :let s = string(list)           " String representation of list
        :call map(list, '">> " . v:val')  " prepend ">> " to each item

Don't forget that a combination of features can make things simple.  For
example, to add up all the numbers in a list: >
        :exe 'let sum = ' . join(nrlist, '+')


1.4 Dictionaries ~
                                        *dict* *Dictionaries* *Dictionary*
A Dictionary is an associative array: Each entry has a key and a value.  The
entry can be located with the key.  The entries are stored without a specific
ordering.


Dictionary creation ~
                                                *E720* *E721* *E722* *E723*
A Dictionary is created with a comma separated list of entries in curly
braces.  Each entry has a key and a value, separated by a colon.  Each key can
only appear once.  Examples: >
        :let mydict = {1: 'one', 2: 'two', 3: 'three'}
        :let emptydict = {}
<                                                       *E713* *E716* *E717*
A key is always a String.  You can use a Number, it will be converted to a
String automatically.  Thus the String '4' and the number 4 will find the same
entry.  Note that the String '04' and the Number 04 are different, since the
```

```
Number will be converted to the String '4'.  The empty string can be used as a
key.

A value can be any expression.  Using a Dictionary for a value creates a
nested Dictionary: >
        :let nestdict = {1: {11: 'a', 12: 'b'}, 2: {21: 'c'}}

An extra comma after the last entry is ignored.


Accessing entries ~

The normal way to access an entry is by putting the key in square brackets: >
        :let val = mydict["one"]
        :let mydict["four"] = 4

You can add new entries to an existing Dictionary this way, unlike Lists.

For keys that consist entirely of letters, digits and underscore the following
form can be used |expr-entry|: >
        :let val = mydict.one
        :let mydict.four = 4

Since an entry can be any type, also a List and a Dictionary, the indexing and
key lookup can be repeated: >
        :echo dict.key[idx].key


Dictionary to List conversion ~

You may want to loop over the entries in a dictionary.  For this you need to
turn the Dictionary into a List and pass it to |:for|.

Most often you want to loop over the keys, using the |keys()| function: >
        :for key in keys(mydict)
        :   echo key . ': ' . mydict[key]
        :endfor

The List of keys is unsorted.  You may want to sort them first: >
        :for key in sort(keys(mydict))

To loop over the values use the |values()| function:  >
        :for v in values(mydict)
        :   echo "value: " . v
        :endfor

If you want both the key and the value use the |items()| function.  It returns
a List in which each item is a  List with two items, the key and the value: >
        :for [key, value] in items(mydict)
        :   echo key . ': ' . value
        :endfor


Dictionary identity ~
                                                        *dict-identity*
Just like Lists you need to use |copy()| and |deepcopy()| to make a copy of a
Dictionary.  Otherwise, assignment results in referring to the same
Dictionary: >
        :let onedict = {'a': 1, 'b': 2}
        :let adict = onedict
        :let adict['a'] = 11
        :echo onedict['a']
```

```
        11

Two Dictionaries compare equal if all the key-value pairs compare equal.  For
more info see |list-identity|.


Dictionary modification ~
                                                        *dict-modification*
To change an already existing entry of a Dictionary, or to add a new entry,
use |:let| this way: >
        :let dict[4] = "four"
        :let dict['one'] = item

Removing an entry from a Dictionary is done with |remove()| or |:unlet|.
Three ways to remove the entry with key "aaa" from dict: >
        :let i = remove(dict, 'aaa')
        :unlet dict.aaa
        :unlet dict['aaa']

Merging a Dictionary with another is done with |extend()|: >
        :call extend(adict, bdict)
This extends adict with all entries from bdict.  Duplicate keys cause entries
in adict to be overwritten.  An optional third argument can change this.
Note that the order of entries in a Dictionary is irrelevant, thus don't
expect ":echo adict" to show the items from bdict after the older entries in
adict.

Weeding out entries from a Dictionary can be done with |filter()|: >
        :call filter(dict, 'v:val =~ "x"')
This removes all entries from "dict" with a value not matching 'x'.


Dictionary function ~
                                *Dictionary-function* *self* *E725* *E862*
When a function is defined with the "dict" attribute it can be used in a
special way with a dictionary.  Example: >
        :function Mylen() dict
        :   return len(self.data)
        :endfunction
        :let mydict = {'data': [0, 1, 2, 3], 'len': function("Mylen")}
        :echo mydict.len()

This is like a method in object oriented programming.  The entry in the
Dictionary is a |Funcref|.  The local variable "self" refers to the dictionary
the function was invoked from.

It is also possible to add a function without the "dict" attribute as a
Funcref to a Dictionary, but the "self" variable is not available then.

                                *numbered-function* *anonymous-function*
To avoid the extra name for the function it can be defined and directly
assigned to a Dictionary in this way: >
        :let mydict = {'data': [0, 1, 2, 3]}
        :function mydict.len()
        :   return len(self.data)
        :endfunction
        :echo mydict.len()

The function will then get a number and the value of dict.len is a |Funcref|
that references this function.  The function can only be used through a
|Funcref|.  It will automatically be deleted when there is no |Funcref|
remaining that refers to it.
```

```
It is not necessary to use the "dict" attribute for a numbered function.

If you get an error for a numbered function, you can find out what it is with
a trick.  Assuming the function is 42, the command is: >
        :function {42}


Functions for Dictionaries ~
                                                        *E715*
Functions that can be used with a Dictionary: >
        :if has_key(dict, 'foo')        " TRUE if dict has entry with key "foo"
        :if empty(dict)                 " TRUE if dict is empty
        :let l = len(dict)              " number of items in dict
        :let big = max(dict)            " maximum value in dict
        :let small = min(dict)          " minimum value in dict
        :let xs = count(dict, 'x')      " count nr of times 'x' appears in dict
        :let s = string(dict)           " String representation of dict
        :call map(dict, '">> " . v:val')  " prepend ">> " to each item


1.5 More about variables ~
                                                        *more-variables*
If you need to know the type of a variable or expression, use the |type()|
function.

When the '!' flag is included in the 'viminfo' option, global variables that
start with an uppercase letter, and don't contain a lowercase letter, are
stored in the viminfo file |viminfo-file|.

When the 'sessionoptions' option contains "global", global variables that
start with an uppercase letter and contain at least one lowercase letter are
stored in the session file |session-file|.

variable name           can be stored where ~
my_var_6                not
My_Var_6                session file
MY_VAR_6                viminfo file


It's possible to form a variable name with curly braces, see
|curly-braces-names|.

==============================================================================
2. Expression syntax                                    *expression-syntax*

Expression syntax summary, from least to most significant:

|expr1| expr2
        expr2 ? expr1 : expr1   if-then-else

|expr2| expr3
        expr3 || expr3 ..       logical OR

|expr3| expr4
        expr4 && expr4 ..       logical AND

|expr4| expr5
        expr5 == expr5          equal
        expr5 != expr5          not equal
        expr5 >  expr5          greater than
        expr5 >= expr5          greater than or equal
```

```
        expr5 <  expr5          smaller than
        expr5 <= expr5          smaller than or equal
        expr5 =~ expr5          regexp matches
        expr5 !~ expr5          regexp doesn't match

        expr5 ==? expr5         equal, ignoring case
        expr5 ==# expr5         equal, match case
        etc.                    As above, append ? for ignoring case, # for
                                matching case

        expr5 is expr5          same |List| instance
        expr5 isnot expr5       different |List| instance

|expr5| expr6
        expr6 +  expr6 ..       number addition or list concatenation
        expr6 -  expr6 ..       number subtraction
        expr6 .  expr6 ..       string concatenation

|expr6| expr7
        expr7 *  expr7 ..       number multiplication
        expr7 /  expr7 ..       number division
        expr7 %  expr7 ..       number modulo

|expr7| expr8
        ! expr7                 logical NOT
        - expr7                 unary minus
        + expr7                 unary plus

|expr8| expr9
        expr8[expr1]            byte of a String or item of a |List|
        expr8[expr1 : expr1]    substring of a String or sublist of a |List|
        expr8.name              entry in a |Dictionary|
        expr8(expr1, ...)       function call with |Funcref| variable

|expr9| number                  number constant
        "string"                string constant, backslash is special
        'string'                string constant, ' is doubled
        [expr1, ...]            |List|
        {expr1: expr1, ...}     |Dictionary|
        &option                 option value
        (expr1)                 nested expression
        variable                internal variable
        va{ria}ble              internal variable with curly braces
        $VAR                    environment variable
        @r                      contents of register 'r'
        function(expr1, ...)    function call
        func{ti}on(expr1, ...)  function call with curly braces
        {args -> expr1}         lambda expression


".." indicates that the operations in this level can be concatenated.
Example: >
        &nu || &list && &shell == "csh"

All expressions within one level are parsed from left to right.


expr1                                                   *expr1* *E109*
-----

expr2 ? expr1 : expr1
```

```
The expression before the '?' is evaluated to a number.  If it evaluates to
|TRUE|, the result is the value of the expression between the '?' and ':',
otherwise the result is the value of the expression after the ':'.
Example: >
        :echo lnum == 1 ? "top" : lnum

Since the first expression is an "expr2", it cannot contain another ?:.  The
other two expressions can, thus allow for recursive use of ?:.
Example: >
        :echo lnum == 1 ? "top" : lnum == 1000 ? "last" : lnum

To keep this readable, using |line-continuation| is suggested: >
        :echo lnum == 1
        :\      ? "top"
        :\      : lnum == 1000
        :\              ? "last"
        :\              : lnum

You should always put a space before the ':', otherwise it can be mistaken for
use in a variable such as "a:1".


expr2 and expr3                                         *expr2* *expr3*
---------------

expr3 || expr3 ..       logical OR              *expr-barbar*
expr4 && expr4 ..       logical AND             *expr-&&*

The "||" and "&&" operators take one argument on each side.  The arguments
are (converted to) Numbers.  The result is:

    input                       output ~
n1      n2              n1 || n2        n1 && n2 ~
|FALSE| |FALSE|         |FALSE|         |FALSE|
|FALSE| |TRUE|          |TRUE|          |FALSE|
|TRUE|  |FALSE|         |TRUE|          |FALSE|
|TRUE|  |TRUE|          |TRUE|          |TRUE|

The operators can be concatenated, for example: >

        &nu || &list && &shell == "csh"

Note that "&&" takes precedence over "||", so this has the meaning of: >

        &nu || (&list && &shell == "csh")

Once the result is known, the expression "short-circuits", that is, further
arguments are not evaluated.  This is like what happens in C.  For example: >

        let a = 1
        echo a || b

This is valid even if there is no variable called "b" because "a" is |TRUE|,
so the result must be |TRUE|.  Similarly below: >

        echo exists("b") && b == "yes"

This is valid whether "b" has been defined or not.  The second clause will
only be evaluated if "b" has been defined.


expr4                                                   *expr4*
```

---

expr5 {cmp} expr5

Compare two expr5 expressions, resulting in a 0 if it evaluates to false, or 1
if it evaluates to true.

```
                        *expr-==*  *expr-!=*  *expr->*   *expr->=*
                        *expr-<*   *expr-<=*  *expr-=~*  *expr-!~*
                        *expr-==#* *expr-!=#* *expr->#*  *expr->=#*
                        *expr-<#*  *expr-<=#* *expr-=~#* *expr-!~#*
                        *expr-==?* *expr-!=?* *expr->?*  *expr->=?*
                        *expr-<?*  *expr-<=?* *expr-=~?* *expr-!~?*
                        *expr-is* *expr-isnot* *expr-is#* *expr-isnot#*
                        *expr-is?* *expr-isnot?*
                use 'ignorecase'    match case     ignore case ~
equal                       ==              ==#             ==?
not equal                   !=              !=#             !=?
greater than                >               >#              >?
greater than or equal       >=              >=#             >=?
smaller than                <               <#              <?
smaller than or equal       <=              <=#             <=?
regexp matches              =~              =~#             =~?
regexp doesn't match        !~              !~#             !~?
same instance               is              is#             is?
different instance          isnot           isnot#          isnot?
```

Examples:
```
"abc" ==# "Abc"   evaluates to 0
"abc" ==? "Abc"   evaluates to 1
"abc" == "Abc"    evaluates to 1 if 'ignorecase' is set, 0 otherwise
```

E691 *E692*

A |List| can only be compared with a |List| and only "equal", "not equal" and
"is" can be used. This compares the values of the list, recursively.
Ignoring case means case is ignored when comparing item values.

E735 *E736*

A |Dictionary| can only be compared with a |Dictionary| and only "equal", "not
equal" and "is" can be used. This compares the key/values of the |Dictionary|
recursively. Ignoring case means case is ignored when comparing item values.

E694

A |Funcref| can only be compared with a |Funcref| and only "equal", "not
equal", "is" and "isnot" can be used. Case is never ignored. Whether
arguments or a Dictionary are bound (with a partial) matters. The
Dictionaries must also be equal (or the same, in case of "is") and the
arguments must be equal (or the same).

To compare Funcrefs to see if they refer to the same function, ignoring bound
Dictionary and arguments, use |get()| to get the function name: >
```
        if get(Part1, 'name') == get(Part2, 'name')
           " Part1 and Part2 refer to the same function
```

When using "is" or "isnot" with a |List| or a |Dictionary| this checks if the
expressions are referring to the same |List| or |Dictionary| instance. A copy
of a |List| is different from the original |List|. When using "is" without
a |List| or a |Dictionary| it is equivalent to using "equal", using "isnot"
equivalent to using "not equal". Except that a different type means the
values are different: >
```
        echo 4 == '4'
        1
```

```
        echo 4 is '4'
        0
        echo 0 is []
        0
"is#"/"isnot#" and "is?"/"isnot?" can be used to match and ignore case.

When comparing a String with a Number, the String is converted to a Number,
and the comparison is done on Numbers.  This means that: >
        echo 0 == 'x'
        1
because 'x' converted to a Number is zero.  However: >
        echo [0] == ['x']
        0
Inside a List or Dictionary this conversion is not used.

When comparing two Strings, this is done with strcmp() or stricmp().  This
results in the mathematical difference (comparing byte values), not
necessarily the alphabetical difference in the local language.

When using the operators with a trailing '#', or the short version and
'ignorecase' is off, the comparing is done with strcmp(): case matters.

When using the operators with a trailing '?', or the short version and
'ignorecase' is set, the comparing is done with stricmp(): case is ignored.

'smartcase' is not used.

The "=~" and "!~" operators match the lefthand argument with the righthand
argument, which is used as a pattern.  See |pattern| for what a pattern is.
This matching is always done like 'magic' was set and 'cpoptions' is empty, no
matter what the actual value of 'magic' or 'cpoptions' is.  This makes scripts
portable.  To avoid backslashes in the regexp pattern to be doubled, use a
single-quote string, see |literal-string|.
Since a string is considered to be a single line, a multi-line pattern
(containing \n, backslash-n) will not match.  However, a literal NL character
can be matched like an ordinary character.  Examples:
        "foo\nbar" =~ "\n"      evaluates to 1
        "foo\nbar" =~ "\\n"     evaluates to 0


expr5 and expr6                                         *expr5* *expr6*
---------------
expr6 +  expr6 ..       Number addition or |List| concatenation *expr-+*
expr6 -  expr6 ..       Number subtraction                      *expr--*
expr6 .  expr6 ..       String concatenation                    *expr-.*

For |Lists| only "+" is possible and then both expr6 must be a list.  The
result is a new list with the two lists Concatenated.

expr7 *  expr7 ..       Number multiplication                   *expr-star*
expr7 /  expr7 ..       Number division                         *expr-/*
expr7 %  expr7 ..       Number modulo                           *expr-%*

For all, except ".", Strings are converted to Numbers.
For bitwise operators see |and()|, |or()| and |xor()|.

Note the difference between "+" and ".":
        "123" + "456" = 579
        "123" . "456" = "123456"

Since '.' has the same precedence as '+' and '-', you need to read: >
        1 . 90 + 90.0
```

```
As: >
        (1 . 90) + 90.0
That works, since the String "190" is automatically converted to the Number
190, which can be added to the Float 90.0.  However: >
        1 . 90 * 90.0
Should be read as: >
        1 . (90 * 90.0)
Since '.' has lower precedence than '*'.  This does NOT work, since this
attempts to concatenate a Float and a String.

When dividing a Number by zero the result depends on the value:
          0 / 0  = -0x80000000  (like NaN for Float)
         >0 / 0  =  0x7fffffff  (like positive infinity)
         <0 / 0  = -0x7fffffff  (like negative infinity)
        (before Vim 7.2 it was always 0x7fffffff)

When 64-bit Number support is enabled:
          0 / 0  = -0x8000000000000000  (like NaN for Float)
         >0 / 0  =  0x7fffffffffffffff  (like positive infinity)
         <0 / 0  = -0x7fffffffffffffff  (like negative infinity)

When the righthand side of '%' is zero, the result is 0.

None of these work for |Funcref|s.

. and % do not work for Float. *E804*


expr7                                                   *expr7*
-----
! expr7                logical NOT             *expr-!*
- expr7                unary minus             *expr-unary--*
+ expr7                unary plus              *expr-unary-+*

For '!' |TRUE| becomes |FALSE|, |FALSE| becomes |TRUE| (one).
For '-' the sign of the number is changed.
For '+' the number is unchanged.

A String will be converted to a Number first.

These three can be repeated and mixed.  Examples:
        !-1         == 0
        !!8         == 1
        --9         == 9


expr8                                                   *expr8*
-----
expr8[expr1]            item of String or |List|        *expr-[]* *E111*
                                                        *E909* *subscript*
If expr8 is a Number or String this results in a String that contains the
expr1'th single byte from expr8.  expr8 is used as a String, expr1 as a
Number.  This doesn't recognize multi-byte encodings, see `byteidx()` for
an alternative, or use `split()` to turn the string into a list of characters.

Index zero gives the first byte.  This is like it works in C.  Careful:
text column numbers start with one!  Example, to get the byte under the
cursor: >
        :let c = getline(".")[col(".") - 1]

If the length of the String is less than the index, the result is an empty
String.  A negative index always results in an empty string (reason: backward
```

```
compatibility).  Use [-1:] to get the last byte.

If expr8 is a |List| then it results the item at index expr1.  See |list-index|
for possible index values.  If the index is out of range this results in an
error.  Example: >
        :let item = mylist[-1]          " get last item

Generally, if a |List| index is equal to or higher than the length of the
|List|, or more negative than the length of the |List|, this results in an
error.


expr8[expr1a : expr1b]  substring or sublist            *expr-[:]*

If expr8 is a Number or String this results in the substring with the bytes
from expr1a to and including expr1b.  expr8 is used as a String, expr1a and
expr1b are used as a Number.  This doesn't recognize multi-byte encodings, see
|byteidx()| for computing the indexes.

If expr1a is omitted zero is used.  If expr1b is omitted the length of the
string minus one is used.

A negative number can be used to measure from the end of the string.  -1 is
the last character, -2 the last but one, etc.

If an index goes out of range for the string characters are omitted.  If
expr1b is smaller than expr1a the result is an empty string.

Examples: >
        :let c = name[-1:]              " last byte of a string
        :let c = name[-2:-2]            " last but one byte of a string
        :let s = line(".")[4:]          " from the fifth byte to the end
        :let s = s[:-3]                 " remove last two bytes
<
                                                        *slice*
If expr8 is a |List| this results in a new |List| with the items indicated by
the indexes expr1a and expr1b.  This works like with a String, as explained
just above. Also see |sublist| below.  Examples: >
        :let l = mylist[:3]             " first four items
        :let l = mylist[4:4]            " List with one item
        :let l = mylist[:]              " shallow copy of a List

Using expr8[expr1] or expr8[expr1a : expr1b] on a |Funcref| results in an
error.

Watch out for confusion between a namespace and a variable followed by a colon
for a sublist: >
        mylist[n:]     " uses variable n
        mylist[s:]     " uses namespace s:, error!


expr8.name              entry in a |Dictionary|         *expr-entry*

If expr8 is a |Dictionary| and it is followed by a dot, then the following
name will be used as a key in the |Dictionary|.  This is just like:
expr8[name].

The name must consist of alphanumeric characters, just like a variable name,
but it may start with a number.  Curly braces cannot be used.

There must not be white space before or after the dot.
```

```
Examples: >
        :let dict = {"one": 1, 2: "two"}
        :echo dict.one
        :echo dict .2

Note that the dot is also used for String concatenation.  To avoid confusion
always put spaces around the dot for String concatenation.


expr8(expr1, ...)       |Funcref| function call

When expr8 is a |Funcref| type variable, invoke the function it refers to.



                                                        *expr9*
number
------
number                  number constant                 *expr-number*
                                *hex-number* *octal-number* *binary-number*

Decimal, Hexadecimal (starting with 0x or 0X), Binary (starting with 0b or 0B)
and Octal (starting with 0).

                                                  *floating-point-format*
Floating point numbers can be written in two forms:

        [-+]{N}.{M}
        [-+]{N}.{M}[eE][-+]{exp}

{N} and {M} are numbers.  Both {N} and {M} must be present and can only
contain digits.
[-+] means there is an optional plus or minus sign.
{exp} is the exponent, power of 10.
Only a decimal point is accepted, not a comma.  No matter what the current
locale is.
{only when compiled with the |+float| feature}

Examples:
        123.456
        +0.0001
        55.0
        -0.123
        1.234e03
        1.0E-6
        -3.1416e+88

These are INVALID:
        3.              empty {M}
        1e40            missing .{M}

                                                        *float-pi* *float-e*
A few useful values to copy&paste: >
        :let pi = 3.14159265359
        :let e  = 2.71828182846

Rationale:
Before floating point was introduced, the text "123.456" was interpreted as
the two numbers "123" and "456", both converted to a string and concatenated,
resulting in the string "123456".  Since this was considered pointless, and we
could not find it intentionally being used in Vim scripts, this backwards
incompatibility was accepted in favor of being able to use the normal notation
```

```
for floating point numbers.

                                                    *floating-point-precision*
The precision and range of floating points numbers depends on what "double"
means in the library Vim was compiled with.  There is no way to change this at
runtime.

The default for displaying a |Float| is to use 6 decimal places, like using
printf("%g", f).  You can select something else when using the |printf()|
function.  Example: >
        :echo printf('%.15e', atan(1))
<       7.853981633974483e-01



string                                  *string* *String* *expr-string* *E114*
------
"string"                string constant         *expr-quote*

Note that double quotes are used.

A string constant accepts these special characters:
\...    three-digit octal number (e.g., "\316")
\..     two-digit octal number (must be followed by non-digit)
\.      one-digit octal number (must be followed by non-digit)
\x..    byte specified with two hex numbers (e.g., "\x1f")
\x.     byte specified with one hex number (must be followed by non-hex char)
\X..    same as \x..
\X.     same as \x.
\u....  character specified with up to 4 hex numbers, stored according to the
        current value of 'encoding' (e.g., "\u02a4")
\U....  same as \u but allows up to 8 hex numbers.
\b      backspace <BS>
\e      escape <Esc>
\f      formfeed <FF>
\n      newline <NL>
\r      return <CR>
\t      tab <Tab>
\\      backslash
\"      double quote
\<xxx>  Special key named "xxx".  e.g. "\<C-W>" for CTRL-W.  This is for use
        in mappings, the 0x80 byte is escaped.
        To use the double quote character it must be escaped: "<M-\">".
        Don't use <Char-xxxx> to get a utf-8 character, use \uxxxx as
        mentioned above.

Note that "\xff" is stored as the byte 255, which may be invalid in some
encodings.  Use "\u00ff" to store character 255 according to the current value
of 'encoding'.

Note that "\000" and "\x00" force the end of the string.


literal-string                                          *literal-string* *E115*
---------------
'string'                string constant                 *expr-'*

Note that single quotes are used.

This string is taken as it is.  No backslashes are removed or have a special
meaning.  The only exception is that two quotes stand for one quote.
```

```
Single quoted strings are useful for patterns, so that backslashes do not need
to be doubled.  These two commands are equivalent: >
        if a =~ "\\s*"
        if a =~ '\s*'


option                                          *expr-option* *E112* *E113*
------
&option                 option value, local value if possible
&g:option               global option value
&l:option               local option value

Examples: >
        echo "tabstop is " . &tabstop
        if &insertmode

Any option name can be used here.  See |options|.  When using the local value
and there is no buffer-local or window-local value, the global value is used
anyway.


register                                                *expr-register* *@r*
--------
@r                      contents of register 'r'

The result is the contents of the named register, as a single string.
Newlines are inserted where required.  To get the contents of the unnamed
register use @" or @@.  See |registers| for an explanation of the available
registers.

When using the '=' register you get the expression itself, not what it
evaluates to.  Use |eval()| to evaluate it.


nesting                                                 *expr-nesting* *E110*
-------
(expr1)                 nested expression


environment variable                                    *expr-env*
--------------------
$VAR                    environment variable

The String value of any environment variable.  When it is not defined, the
result is an empty string.
                                                    *expr-env-expand*
Note that there is a difference between using $VAR directly and using
expand("$VAR").  Using it directly will only expand environment variables that
are known inside the current Vim session.  Using expand() will first try using
the environment variables known inside the current Vim session.  If that
fails, a shell will be used to expand the variable.  This can be slow, but it
does expand all variables that the shell knows about.  Example: >
        :echo $shell
        :echo expand("$shell")
The first one probably doesn't echo anything, the second echoes the $shell
variable (if your shell supports it).


internal variable                                       *expr-variable*
-----------------
variable                internal variable
See below |internal-variables|.
```

```
function call           *expr-function* *E116* *E118* *E119* *E120*
-------------
function(expr1, ...)    function call
See below |functions|.


lambda expression                                *expr-lambda* *lambda*
-----------------
{args -> expr1}         lambda expression

A lambda expression creates a new unnamed function which returns the result of
evaluating |expr1|.  Lambda expressions differ from |user-functions| in
the following ways:

1. The body of the lambda expression is an |expr1| and not a sequence of |Ex|
   commands.
2. The prefix "a:" should not be used for arguments.  E.g.: >
        :let F = {arg1, arg2 -> arg1 - arg2}
        :echo F(5, 2)
<       3

The arguments are optional.  Example: >
        :let F = {-> 'error function'}
        :echo F()
<       error function
                                                          *closure*
Lambda expressions can access outer scope variables and arguments.  This is
often called a closure.  Example where "i" and "a:arg" are used in a lambda
while they exist in the function scope.  They remain valid even after the
function returns: >
        :function Foo(arg)
        :  let i = 3
        :  return {x -> x + i - a:arg}
        :endfunction
        :let Bar = Foo(4)
        :echo Bar(6)
<       5

See also |:func-closure|.  Lambda and closure support can be checked with: >
        if has('lambda')

Examples for using a lambda expression with |sort()|, |map()| and |filter()|: >
        :echo map([1, 2, 3], {idx, val -> val + 1})
<       [2, 3, 4] >
        :echo sort([3,7,2,1,4], {a, b -> a - b})
<       [1, 2, 3, 4, 7]

The lambda expression is also useful for Channel, Job and timer: >
        :let timer = timer_start(500,
                        \ {-> execute("echo 'Handler called'", "")},
                        \ {'repeat': 3})
<       Handler called
        Handler called
        Handler called

Note how execute() is used to execute an Ex command.  That's ugly though.


Lambda expressions have internal names like '<lambda>42'.  If you get an error
for a lambda expression, you can find what it is with the following command: >
```

```
        :function {'<lambda>42'}
See also: |numbered-function|

==============================================================================
3. Internal variable                                *internal-variables* *E461*

An internal variable name can be made up of letters, digits and '_'.  But it
cannot start with a digit.  It's also possible to use curly braces, see
|curly-braces-names|.

An internal variable is created with the ":let" command |:let|.
An internal variable is explicitly destroyed with the ":unlet" command
|:unlet|.
Using a name that is not an internal variable or refers to a variable that has
been destroyed results in an error.

There are several name spaces for variables.  Which one is to be used is
specified by what is prepended:

                (nothing) In a function: local to a function; otherwise: global
|buffer-variable|    b:   Local to the current buffer.
|window-variable|    w:   Local to the current window.
|tabpage-variable|   t:   Local to the current tab page.
|global-variable|    g:   Global.
|local-variable|     l:   Local to a function.
|script-variable|    s:   Local to a |:source|'ed Vim script.
|function-argument|  a:   Function argument (only inside a function).
|vim-variable|       v:   Global, predefined by Vim.

The scope name by itself can be used as a |Dictionary|.  For example, to
delete all script-local variables: >
        :for k in keys(s:)
        :    unlet s:[k]
        :endfor
<
                                                    *buffer-variable* *b:var* *b:*
A variable name that is preceded with "b:" is local to the current buffer.
Thus you can have several "b:foo" variables, one for each buffer.
This kind of variable is deleted when the buffer is wiped out or deleted with
|:bdelete|.

One local buffer variable is predefined:
                                        *b:changedtick* *changetick*
b:changedtick   The total number of changes to the current buffer.  It is
                incremented for each change.  An undo command is also a change
                in this case.  This can be used to perform an action only when
                the buffer has changed.  Example: >
                    :if my_changedtick != b:changedtick
                    :   let my_changedtick = b:changedtick
                    :   call My_Update()
                    :endif
<               You cannot change or delete the b:changedtick variable.

                                                    *window-variable* *w:var* *w:*
A variable name that is preceded with "w:" is local to the current window.  It
is deleted when the window is closed.

                                                    *tabpage-variable* *t:var* *t:*
A variable name that is preceded with "t:" is local to the current tab page,
It is deleted when the tab page is closed. {not available when compiled
without the |+windows| feature}
```

global-variable *g:var* *g:*
Inside functions global variables are accessed with "g:". Omitting this will
access a variable local to a function. But "g:" can also be used in any other
place if you like.

local-variable *l:var* *l:*
Inside functions local variables are accessed without prepending anything.
But you can also prepend "l:" if you like. However, without prepending "l:"
you may run into reserved variable names. For example "count". By itself it
refers to "v:count". Using "l:count" you can have a local variable with the
same name.

script-variable *s:var*
In a Vim script variables starting with "s:" can be used. They cannot be
accessed from outside of the scripts, thus are local to the script.

They can be used in:
- commands executed while the script is sourced
- functions defined in the script
- autocommands defined in the script
- functions and autocommands defined in functions and autocommands which were
  defined in the script (recursively)
- user defined commands defined in the script
Thus not in:
- other scripts sourced from this one
- mappings
- menus
- etc.

Script variables can be used to avoid conflicts with global variable names.
Take this example: >

```
        let s:counter = 0
        function MyCounter()
          let s:counter = s:counter + 1
          echo s:counter
        endfunction
        command Tick call MyCounter()
```

You can now invoke "Tick" from any script, and the "s:counter" variable in
that script will not be changed, only the "s:counter" in the script where
"Tick" was defined is used.

Another example that does the same: >

```
        let s:counter = 0
        command Tick let s:counter = s:counter + 1 | echo s:counter
```

When calling a function and invoking a user-defined command, the context for
script variables is set to the script where the function or command was
defined.

The script variables are also available when a function is defined inside a
function that is defined in a script. Example: >

```
        let s:counter = 0
        function StartCounting(incr)
          if a:incr
            function MyCounter()
              let s:counter = s:counter + 1
            endfunction
          else
```

```
            function MyCounter()
              let s:counter = s:counter - 1
            endfunction
          endif
        endfunction
```

This defines the MyCounter() function either for counting up or counting down
when calling StartCounting(). It doesn't matter from where StartCounting() is
called, the s:counter variable will be accessible in MyCounter().

When the same script is sourced again it will use the same script variables.
They will remain valid as long as Vim is running. This can be used to
maintain a counter: >

```
        if !exists("s:counter")
          let s:counter = 1
          echo "script executed for the first time"
        else
          let s:counter = s:counter + 1
          echo "script executed " . s:counter . " times now"
        endif
```

Note that this means that filetype plugins don't get a different set of script
variables for each buffer. Use local buffer variables instead |b:var|.

Predefined Vim variables: *vim-variable* *v:var* *v:*

v:beval_col *beval_col-variable*
v:beval_col The number of the column, over which the mouse pointer is.
This is the byte index in the |v:beval_lnum| line.
Only valid while evaluating the 'balloonexpr' option.

v:beval_bufnr *beval_bufnr-variable*
v:beval_bufnr The number of the buffer, over which the mouse pointer is. Only
valid while evaluating the 'balloonexpr' option.

v:beval_lnum *beval_lnum-variable*
v:beval_lnum The number of the line, over which the mouse pointer is. Only
valid while evaluating the 'balloonexpr' option.

v:beval_text *beval_text-variable*
v:beval_text The text under or after the mouse pointer. Usually a word as
it is useful for debugging a C program. 'iskeyword' applies,
but a dot and "->" before the position is included. When on a
']' the text before it is used, including the matching '[' and
word before it. When on a Visual area within one line the
highlighted text is used. Also see |<cexpr>|.
Only valid while evaluating the 'balloonexpr' option.

v:beval_winnr *beval_winnr-variable*
v:beval_winnr The number of the window, over which the mouse pointer is. Only
valid while evaluating the 'balloonexpr' option. The first
window has number zero (unlike most other places where a
window gets a number).

v:beval_winid *beval_winid-variable*
v:beval_winid The |window-ID| of the window, over which the mouse pointer
is. Otherwise like v:beval_winnr.

v:char *char-variable*
v:char Argument for evaluating 'formatexpr' and used for the typed

```
                character when using <expr> in an abbreviation |:map-<expr>|.
                It is also used by the |InsertCharPre| and |InsertEnter| events.

                        *v:charconvert_from* *charconvert_from-variable*
v:charconvert_from
                The name of the character encoding of a file to be converted.
                Only valid while evaluating the 'charconvert' option.

                        *v:charconvert_to* *charconvert_to-variable*
v:charconvert_to
                The name of the character encoding of a file after conversion.
                Only valid while evaluating the 'charconvert' option.

                                        *v:cmdarg* *cmdarg-variable*
v:cmdarg        This variable is used for two purposes:
                1. The extra arguments given to a file read/write command.
                   Currently these are "++enc=" and "++ff=".  This variable is
                   set before an autocommand event for a file read/write
                   command is triggered.  There is a leading space to make it
                   possible to append this variable directly after the
                   read/write command.  Note: The "+cmd" argument isn't
                   included here, because it will be executed anyway.
                2. When printing a PostScript file with ":hardcopy" this is
                   the argument for the ":hardcopy" command.  This can be used
                   in 'printexpr'.

                                        *v:cmdbang* *cmdbang-variable*
v:cmdbang       Set like v:cmdarg for a file read/write command.  When a "!"
                was used the value is 1, otherwise it is 0.  Note that this
                can only be used in autocommands.  For user commands |<bang>|
                can be used.

                                *v:completed_item* *completed_item-variable*
v:completed_item
                |Dictionary| containing the |complete-items| for the most
                recently completed word after |CompleteDone|.  The
                |Dictionary| is empty if the completion failed.

                                        *v:count* *count-variable*
v:count         The count given for the last Normal mode command.  Can be used
                to get the count before a mapping.  Read-only.  Example: >
        :map _x :<C-U>echo "the count is " . v:count<CR>
<               Note: The <C-U> is required to remove the line range that you
                get when typing ':' after a count.
                When there are two counts, as in "3d2w", they are multiplied,
                just like what happens in the command, "d6w" for the example.
                Also used for evaluating the 'formatexpr' option.
                "count" also works, for backwards compatibility.

                                        *v:count1* *count1-variable*
v:count1        Just like "v:count", but defaults to one when no count is
                used.

                                                *v:ctype* *ctype-variable*
v:ctype         The current locale setting for characters of the runtime
                environment.  This allows Vim scripts to be aware of the
                current locale encoding.  Technical: it's the value of
                LC_CTYPE.  When not using a locale the value is "C".
                This variable can not be set directly, use the |:language|
                command.
                See |multi-lang|.
```

```
                                        *v:dying* *dying-variable*
v:dying         Normally zero.  When a deadly signal is caught it's set to
                one.  When multiple signals are caught the number increases.
                Can be used in an autocommand to check if Vim didn't
                terminate normally. {only works on Unix}
                Example: >
        :au VimLeave * if v:dying | echo "\nAAAAaaaarrrgggghhhh!!!\n" | endif
<               Note: if another deadly signal is caught when v:dying is one,
                VimLeave autocommands will not be executed.

                                        *v:errmsg* *errmsg-variable*
v:errmsg        Last given error message.  It's allowed to set this variable.
                Example: >
        :let v:errmsg = ""
        :silent! next
        :if v:errmsg != ""
        :  ... handle error
<               "errmsg" also works, for backwards compatibility.

                                        *v:errors* *errors-variable*
v:errors        Errors found by assert functions, such as |assert_true()|.
                This is a list of strings.
                The assert functions append an item when an assert fails.
                To remove old results make it empty: >
        :let v:errors = []
<               If v:errors is set to anything but a list it is made an empty
                list by the assert function.

                                        *v:exception* *exception-variable*
v:exception     The value of the exception most recently caught and not
                finished.  See also |v:throwpoint| and |throw-variables|.
                Example: >
        :try
        :  throw "oops"
        :catch /.*/
        :  echo "caught" v:exception
        :endtry
<               Output: "caught oops".

                                        *v:false* *false-variable*
v:false         A Number with value zero. Used to put "false" in JSON.  See
                |json_encode()|.
                When used as a string this evaluates to "v:false". >
                        echo v:false
<                       v:false ~
                That is so that eval() can parse the string back to the same
                value.  Read-only.

                                        *v:fcs_reason* *fcs_reason-variable*
v:fcs_reason    The reason why the |FileChangedShell| event was triggered.
                Can be used in an autocommand to decide what to do and/or what
                to set v:fcs_choice to.  Possible values:
                        deleted         file no longer exists
                        conflict        file contents, mode or timestamp was
                                        changed and buffer is modified
                        changed         file contents has changed
                        mode            mode of file changed
                        time            only file timestamp changed

                                        *v:fcs_choice* *fcs_choice-variable*
v:fcs_choice    What should happen after a |FileChangedShell| event was
                triggered.  Can be used in an autocommand to tell Vim what to
```

```
                do with the affected buffer:
                        reload          Reload the buffer (does not work if
                                        the file was deleted).
                        ask             Ask the user what to do, as if there
                                        was no autocommand.  Except that when
                                        only the timestamp changed nothing
                                        will happen.
                        <empty>         Nothing, the autocommand should do
                                        everything that needs to be done.
                The default is empty.  If another (invalid) value is used then
                Vim behaves like it is empty, there is no warning message.

                                        *v:fname_in* *fname_in-variable*
v:fname_in      The name of the input file.  Valid while evaluating:
                        option          used for ~
                        'charconvert'   file to be converted
                        'diffexpr'      original file
                        'patchexpr'     original file
                        'printexpr'     file to be printed
                And set to the swap file name for |SwapExists|.

                                        *v:fname_out* *fname_out-variable*
v:fname_out     The name of the output file.  Only valid while
                evaluating:
                        option          used for ~
                        'charconvert'   resulting converted file (*)
                        'diffexpr'      output of diff
                        'patchexpr'     resulting patched file
                (*) When doing conversion for a write command (e.g., ":w
                file") it will be equal to v:fname_in.  When doing conversion
                for a read command (e.g., ":e file") it will be a temporary
                file and different from v:fname_in.

                                        *v:fname_new* *fname_new-variable*
v:fname_new     The name of the new version of the file.  Only valid while
                evaluating 'diffexpr'.

                                        *v:fname_diff* *fname_diff-variable*
v:fname_diff    The name of the diff (patch) file.  Only valid while
                evaluating 'patchexpr'.

                                        *v:folddashes* *folddashes-variable*
v:folddashes    Used for 'foldtext': dashes representing foldlevel of a closed
                fold.
                Read-only in the |sandbox|. |fold-foldtext|

                                        *v:foldlevel* *foldlevel-variable*
v:foldlevel     Used for 'foldtext': foldlevel of closed fold.
                Read-only in the |sandbox|. |fold-foldtext|

                                        *v:foldend* *foldend-variable*
v:foldend       Used for 'foldtext': last line of closed fold.
                Read-only in the |sandbox|. |fold-foldtext|

                                        *v:foldstart* *foldstart-variable*
v:foldstart     Used for 'foldtext': first line of closed fold.
                Read-only in the |sandbox|. |fold-foldtext|

                                        *v:hlsearch* *hlsearch-variable*
v:hlsearch      Variable that indicates whether search highlighting is on.
                Setting it makes sense only if 'hlsearch' is enabled which
                requires |+extra_search|. Setting this variable to zero acts
```

```
                like the |:nohlsearch| command, setting it to one acts like >
                        let &hlsearch = &hlsearch
<               Note that the value is restored when returning from a
                function. |function-search-undo|.

                                        *v:insertmode* *insertmode-variable*
v:insertmode    Used for the |InsertEnter| and |InsertChange| autocommand
                events.  Values:
                        i       Insert mode
                        r       Replace mode
                        v       Virtual Replace mode

                                        *v:key* *key-variable*
v:key           Key of the current item of a |Dictionary|.  Only valid while
                evaluating the expression used with |map()| and |filter()|.
                Read-only.

                                        *v:lang* *lang-variable*
v:lang          The current locale setting for messages of the runtime
                environment.  This allows Vim scripts to be aware of the
                current language.  Technical: it's the value of LC_MESSAGES.
                The value is system dependent.
                This variable can not be set directly, use the |:language|
                command.
                It can be different from |v:ctype| when messages are desired
                in a different language than what is used for character
                encoding.  See |multi-lang|.

                                        *v:lc_time* *lc_time-variable*
v:lc_time       The current locale setting for time messages of the runtime
                environment.  This allows Vim scripts to be aware of the
                current language.  Technical: it's the value of LC_TIME.
                This variable can not be set directly, use the |:language|
                command.  See |multi-lang|.

                                        *v:lnum* *lnum-variable*
v:lnum          Line number for the 'foldexpr' |fold-expr|, 'formatexpr' and
                'indentexpr' expressions, tab page number for 'guitablabel'
                and 'guitabtooltip'.  Only valid while one of these
                expressions is being evaluated.  Read-only when in the
                |sandbox|.

                                        *v:mouse_win* *mouse_win-variable*
v:mouse_win     Window number for a mouse click obtained with |getchar()|.
                First window has number 1, like with |winnr()|.  The value is
                zero when there was no mouse button click.

                                        *v:mouse_winid* *mouse_winid-variable*
v:mouse_winid   Window ID for a mouse click obtained with |getchar()|.
                The value is zero when there was no mouse button click.

                                        *v:mouse_lnum* *mouse_lnum-variable*
v:mouse_lnum    Line number for a mouse click obtained with |getchar()|.
                This is the text line number, not the screen line number.  The
                value is zero when there was no mouse button click.

                                        *v:mouse_col* *mouse_col-variable*
v:mouse_col     Column number for a mouse click obtained with |getchar()|.
                This is the screen column number, like with |virtcol()|.  The
                value is zero when there was no mouse button click.

                                        *v:none* *none-variable*
```

```
v:none          An empty String. Used to put an empty item in JSON.  See
                |json_encode()|.
                When used as a number this evaluates to zero.
                When used as a string this evaluates to "v:none". >
                        echo v:none
<                       v:none ~
                That is so that eval() can parse the string back to the same
                value.  Read-only.

                                        *v:null* *null-variable*
v:null          An empty String. Used to put "null" in JSON.  See
                |json_encode()|.
                When used as a number this evaluates to zero.
                When used as a string this evaluates to "v:null". >
                        echo v:null
<                       v:null ~
                That is so that eval() can parse the string back to the same
                value.  Read-only.

                                        *v:oldfiles* *oldfiles-variable*
v:oldfiles      List of file names that is loaded from the |viminfo| file on
                startup.  These are the files that Vim remembers marks for.
                The length of the List is limited by the ' argument of the
                'viminfo' option (default is 100).
                When the |viminfo| file is not used the List is empty.
                Also see |:oldfiles| and |c_#<|.
                The List can be modified, but this has no effect on what is
                stored in the |viminfo| file later.  If you use values other
                than String this will cause trouble.
                {only when compiled with the |+viminfo| feature}

                                                    *v:option_new*
v:option_new    New value of the option. Valid while executing an |OptionSet|
                autocommand.
                                                    *v:option_old*
v:option_old    Old value of the option. Valid while executing an |OptionSet|
                autocommand.
                                                    *v:option_type*
v:option_type   Scope of the set command. Valid while executing an
                |OptionSet| autocommand. Can be either "global" or "local"
                                        *v:operator* *operator-variable*
v:operator      The last operator given in Normal mode.  This is a single
                character except for commands starting with <g> or <z>,
                in which case it is two characters.  Best used alongside
                |v:prevcount| and |v:register|.  Useful if you want to cancel
                Operator-pending mode and then use the operator, e.g.: >
                        :omap O <Esc>:call MyMotion(v:operator)<CR>
<               The value remains set until another operator is entered, thus
                don't expect it to be empty.
                v:operator is not set for |:delete|, |:yank| or other Ex
                commands.
                Read-only.

                                        *v:prevcount* *prevcount-variable*
v:prevcount     The count given for the last but one Normal mode command.
                This is the v:count value of the previous command.  Useful if
                you want to cancel Visual or Operator-pending mode and then
                use the count, e.g.: >
                        :vmap % <Esc>:call MyFilter(v:prevcount)<CR>
<               Read-only.

                                        *v:profiling* *profiling-variable*
```

```
v:profiling     Normally zero.  Set to one after using ":profile start".
                See |profiling|.

                                        *v:progname* *progname-variable*
v:progname      Contains the name (with path removed) with which Vim was
                invoked.  Allows you to do special initialisations for |view|,
                |evim| etc., or any other name you might symlink to Vim.
                Read-only.

                                        *v:progpath* *progpath-variable*
v:progpath      Contains the command with which Vim was invoked, including the
                path.  Useful if you want to message a Vim server using a
                |--remote-expr|.
                To get the full path use: >
                        echo exepath(v:progpath)
<               If the path is relative it will be expanded to the full path,
                so that it still works after `:cd`. Thus starting "./vim"
                results in "/home/user/path/to/vim/src/vim".
                On MS-Windows the executable may be called "vim.exe", but the
                ".exe" is not added to v:progpath.
                Read-only.

                                        *v:register* *register-variable*
v:register      The name of the register in effect for the current normal mode
                command (regardless of whether that command actually used a
                register).  Or for the currently executing normal mode mapping
                (use this in custom commands that take a register).
                If none is supplied it is the default register '"', unless
                'clipboard' contains "unnamed" or "unnamedplus", then it is
                '*' or '+'.
                Also see |getreg()| and |setreg()|

                                        *v:scrollstart* *scrollstart-variable*
v:scrollstart   String describing the script or function that caused the
                screen to scroll up.  It's only set when it is empty, thus the
                first reason is remembered.  It is set to "Unknown" for a
                typed command.
                This can be used to find out why your script causes the
                hit-enter prompt.

                                        *v:servername* *servername-variable*
v:servername    The resulting registered |client-server-name| if any.
                Read-only.


v:searchforward                 *v:searchforward* *searchforward-variable*
                Search direction:  1 after a forward search, 0 after a
                backward search.  It is reset to forward when directly setting
                the last search pattern, see |quote/|.
                Note that the value is restored when returning from a
                function. |function-search-undo|.
                Read-write.

                                        *v:shell_error* *shell_error-variable*
v:shell_error   Result of the last shell command.  When non-zero, the last
                shell command had an error.  When zero, there was no problem.
                This only works when the shell returns the error code to Vim.
                The value -1 is often used when the command could not be
                executed.  Read-only.
                Example: >
        :!mv foo bar
        :if v:shell_error
```

```
        :  echo 'could not rename "foo" to "bar"!'
        :endif
<               "shell_error" also works, for backwards compatibility.

                                        *v:statusmsg* *statusmsg-variable*
v:statusmsg     Last given status message.  It's allowed to set this variable.

                                        *v:swapname* *swapname-variable*
v:swapname      Only valid when executing |SwapExists| autocommands: Name of
                the swap file found.  Read-only.

                                        *v:swapchoice* *swapchoice-variable*
v:swapchoice    |SwapExists| autocommands can set this to the selected choice
                for handling an existing swap file:
                        'o'     Open read-only
                        'e'     Edit anyway
                        'r'     Recover
                        'd'     Delete swapfile
                        'q'     Quit
                        'a'     Abort
                The value should be a single-character string.  An empty value
                results in the user being asked, as would happen when there is
                no SwapExists autocommand.  The default is empty.

                                        *v:swapcommand* *swapcommand-variable*
v:swapcommand   Normal mode command to be executed after a file has been
                opened.  Can be used for a |SwapExists| autocommand to have
                another Vim open the file and jump to the right place.  For
                example, when jumping to a tag the value is ":tag tagname\r".
                For ":edit +cmd file" the value is ":cmd\r".

                                *v:t_TYPE* *v:t_bool* *t_bool-variable*
v:t_bool        Value of Boolean type.  Read-only.  See: |type()|
                                        *v:t_channel* *t_channel-variable*
v:t_channel     Value of Channel type.  Read-only.  See: |type()|
                                        *v:t_dict* *t_dict-variable*
v:t_dict        Value of Dictionary type.  Read-only.  See: |type()|
                                        *v:t_float* *t_float-variable*
v:t_float       Value of Float type.  Read-only.  See: |type()|
                                        *v:t_func* *t_func-variable*
v:t_func        Value of Funcref type.  Read-only.  See: |type()|
                                        *v:t_job* *t_job-variable*
v:t_job         Value of Job type.  Read-only.  See: |type()|
                                        *v:t_list* *t_list-variable*
v:t_list        Value of List type.  Read-only.  See: |type()|
                                        *v:t_none* *t_none-variable*
v:t_none        Value of None type.  Read-only.  See: |type()|
                                        *v:t_number* *t_number-variable*
v:t_number      Value of Number type.  Read-only.  See: |type()|
                                        *v:t_string* *t_string-variable*
v:t_string      Value of String type.  Read-only.  See: |type()|

                                *v:termresponse* *termresponse-variable*
v:termresponse  The escape sequence returned by the terminal for the |t_RV|
                termcap entry.  It is set when Vim receives an escape sequence
                that starts with ESC [ or CSI and ends in a 'c', with only
                digits, ';' and '.' in between.
                When this option is set, the TermResponse autocommand event is
                fired, so that you can react to the response from the
                terminal.
                The response from a new xterm is: "<Esc>[ Pp ; Pv ; Pc c".  Pp
                is the terminal type: 0 for vt100 and 1 for vt220.  Pv is the
```

```
                patch level (since this was introduced in patch 95, it's
                always 95 or bigger).  Pc is always zero.
                {only when compiled with |+termresponse| feature}

                                                    *v:termblinkresp*
v:termblinkresp The escape sequence returned by the terminal for the |t_RC|
                termcap entry.  This is used to find out whether the terminal
                cursor is blinking. This is used by |term_getcursor()|.

                                                    *v:termstyleresp*
v:termstyleresp The escape sequence returned by the terminal for the |t_RS|
                termcap entry.  This is used to find out what the shape of the
                cursor is.  This is used by |term_getcursor()|.

                                                    *v:termrgbresp*
v:termrgbresp   The escape sequence returned by the terminal for the |t_RB|
                termcap entry.  This is used to find out what the terminal
                background color is, see 'background'.

                                                    *v:termu7resp*
v:termu7resp    The escape sequence returned by the terminal for the |t_u7|
                termcap entry.  This is used to find out what the terminal
                does with ambiguous width characters, see 'ambiwidth'.

                                          *v:testing* *testing-variable*
v:testing       Must be set before using `test_garbagecollect_now()`.
                Also, when set certain error messages won't be shown for 2
                seconds. (e.g. "'dictionary' option is empty")

                                  *v:this_session* *this_session-variable*
v:this_session  Full filename of the last loaded or saved session file.  See
                |:mksession|.  It is allowed to set this variable.  When no
                session file has been saved, this variable is empty.
                "this_session" also works, for backwards compatibility.

                                          *v:throwpoint* *throwpoint-variable*
v:throwpoint    The point where the exception most recently caught and not
                finished was thrown.  Not set when commands are typed.  See
                also |v:exception| and |throw-variables|.
                Example: >
        :try
        :  throw "oops"
        :catch /.*/
        :  echo "Exception from" v:throwpoint
        :endtry
<               Output: "Exception from test.vim, line 2"

                                                    *v:true* *true-variable*
v:true          A Number with value one. Used to put "true" in JSON.  See
                |json_encode()|.
                When used as a string this evaluates to "v:true". >
                        echo v:true
<                       v:true ~
                That is so that eval() can parse the string back to the same
                value.  Read-only.
                                                    *v:val* *val-variable*
v:val           Value of the current item of a |List| or |Dictionary|.  Only
                valid while evaluating the expression used with |map()| and
                |filter()|.  Read-only.

                                          *v:version* *version-variable*
v:version       Version number of Vim: Major version number times 100 plus
```

```
                minor version number.  Version 5.0 is 500.  Version 5.1 (5.01)
                is 501.  Read-only.  "version" also works, for backwards
                compatibility.
                Use |has()| to check if a certain patch was included, e.g.: >
                        if has("patch-7.4.123")
<               Note that patch numbers are specific to the version, thus both
                version 5.0 and 5.1 may have a patch 123, but these are
                completely different.

                                *v:vim_did_enter* *vim_did_enter-variable*
v:vim_did_enter Zero until most of startup is done.  It is set to one just
                before |VimEnter| autocommands are triggered.

                                        *v:warningmsg* *warningmsg-variable*
v:warningmsg    Last given warning message.  It's allowed to set this variable.

                                        *v:windowid* *windowid-variable*
v:windowid      When any X11 based GUI is running or when running in a
                terminal and Vim connects to the X server (|-X|) this will be
                set to the window ID.
                When an MS-Windows GUI is running this will be set to the
                window handle.
                Otherwise the value is zero.
                Note: for windows inside Vim use |winnr()| or |win_getid()|,
                see |window-ID|.

==============================================================================
4. Builtin Functions                                    *functions*

See |function-list| for a list grouped by what the function is used for.

(Use CTRL-] on the function name to jump to the full explanation.)

USAGE                           RESULT  DESCRIPTION     ~

abs({expr})                     Float or Number  absolute value of {expr}
acos({expr})                    Float   arc cosine of {expr}
add({list}, {item})             List    append {item} to |List| {list}
and({expr}, {expr})             Number  bitwise AND
append({lnum}, {string})        Number  append {string} below line {lnum}
append({lnum}, {list})          Number  append lines {list} below line {lnum}
argc()                          Number  number of files in the argument list
argidx()                        Number  current index in the argument list
arglistid([{winnr} [, {tabnr}]]) Number argument list id
argv({nr})                      String  {nr} entry of the argument list
argv()                          List    the argument list
assert_equal({exp}, {act} [, {msg}])
                                none    assert {exp} is equal to {act}
assert_exception({error} [, {msg}])
                                none    assert {error} is in v:exception
assert_fails({cmd} [, {error}]) none    assert {cmd} fails
assert_false({actual} [, {msg}])
                                none    assert {actual} is false
assert_inrange({lower}, {upper}, {actual} [, {msg}])
                                none    assert {actual} is inside the range
assert_match({pat}, {text} [, {msg}])
                                none    assert {pat} matches {text}
assert_notequal({exp}, {act} [, {msg}])
                                none    assert {exp} is not equal {act}
assert_notmatch({pat}, {text} [, {msg}])
                                none    assert {pat} not matches {text}
assert_report({msg})            none    report a test failure
```

```
assert_true({actual} [, {msg}]) none    assert {actual} is true
asin({expr})                    Float   arc sine of {expr}
atan({expr})                    Float   arc tangent of {expr}
atan2({expr1}, {expr2})         Float   arc tangent of {expr1} / {expr2}
balloon_show({msg})             none    show {msg} inside the balloon
browse({save}, {title}, {initdir}, {default})
                                String  put up a file requester
browsedir({title}, {initdir})   String  put up a directory requester
bufexists({expr})               Number  |TRUE| if buffer {expr} exists
buflisted({expr})               Number  |TRUE| if buffer {expr} is listed
bufloaded({expr})               Number  |TRUE| if buffer {expr} is loaded
bufname({expr})                 String  Name of the buffer {expr}
bufnr({expr} [, {create}])      Number  Number of the buffer {expr}
bufwinid({expr})                Number  window ID of buffer {expr}
bufwinnr({expr})                Number  window number of buffer {expr}
byte2line({byte})               Number  line number at byte count {byte}
byteidx({expr}, {nr})           Number  byte index of {nr}'th char in {expr}
byteidxcomp({expr}, {nr})       Number  byte index of {nr}'th char in {expr}
call({func}, {arglist} [, {dict}])
                                any     call {func} with arguments {arglist}
ceil({expr})                    Float   round {expr} up
ch_canread({handle})            Number  check if there is something to read
ch_close({handle})              none    close {handle}
ch_close_in({handle})           none    close in part of {handle}
ch_evalexpr({handle}, {expr} [, {options}])
                                any     evaluate {expr} on JSON {handle}
ch_evalraw({handle}, {string} [, {options}])
                                any     evaluate {string} on raw {handle}
ch_getbufnr({handle}, {what})   Number  get buffer number for {handle}/{what}
ch_getjob({channel})            Job     get the Job of {channel}
ch_info({handle})               String  info about channel {handle}
ch_log({msg} [, {handle}])      none    write {msg} in the channel log file
ch_logfile({fname} [, {mode}])  none    start logging channel activity
ch_open({address} [, {options}])
                                Channel open a channel to {address}
ch_read({handle} [, {options}]) String  read from {handle}
ch_readraw({handle} [, {options}])
                                String  read raw from {handle}
ch_sendexpr({handle}, {expr} [, {options}])
                                any     send {expr} over JSON {handle}
ch_sendraw({handle}, {string} [, {options}])
                                any     send {string} over raw {handle}
ch_setoptions({handle}, {options})
                                none    set options for {handle}
ch_status({handle} [, {options}])
                                String  status of channel {handle}
changenr()                      Number  current change number
char2nr({expr}[, {utf8}])       Number  ASCII/UTF8 value of first char in {expr}
cindent({lnum})                 Number  C indent for line {lnum}
clearmatches()                  none    clear all matches
col({expr})                     Number  column nr of cursor or mark
complete({startcol}, {matches}) none    set Insert mode completion
complete_add({expr})            Number  add completion match
complete_check()                Number  check for key typed during completion
confirm({msg} [, {choices} [, {default} [, {type}]]])
                                Number  number of choice picked by user
copy({expr})                    any     make a shallow copy of {expr}
cos({expr})                     Float   cosine of {expr}
cosh({expr})                    Float   hyperbolic cosine of {expr}
count({list}, {expr} [, {ic} [, {start}]])
                                Number  count how many {expr} are in {list}
cscope_connection([{num}, {dbpath} [, {prepend}]])
```

```
                                Number  checks existence of cscope connection
cursor({lnum}, {col} [, {off}])
                                Number  move cursor to {lnum}, {col}, {off}
cursor({list})                  Number  move cursor to position in {list}
deepcopy({expr} [, {noref}])    any     make a full copy of {expr}
delete({fname} [, {flags}])     Number  delete the file or directory {fname}
did_filetype()                  Number  |TRUE| if FileType autocmd event used
diff_filler({lnum})             Number  diff filler lines about {lnum}
diff_hlID({lnum}, {col})        Number  diff highlighting at {lnum}/{col}
empty({expr})                   Number  |TRUE| if {expr} is empty
escape({string}, {chars})       String  escape {chars} in {string} with '\'
eval({string})                  any     evaluate {string} into its value
eventhandler()                  Number  |TRUE| if inside an event handler
executable({expr})              Number  1 if executable {expr} exists
execute({command})              String  execute {command} and get the output
exepath({expr})                 String  full path of the command {expr}
exists({expr})                  Number  |TRUE| if {expr} exists
extend({expr1}, {expr2} [, {expr3}])
                                List/Dict insert items of {expr2} into {expr1}
exp({expr})                     Float   exponential of {expr}
expand({expr} [, {nosuf} [, {list}]])
                                any     expand special keywords in {expr}
feedkeys({string} [, {mode}])   Number  add key sequence to typeahead buffer
filereadable({file})            Number  |TRUE| if {file} is a readable file
filewritable({file})            Number  |TRUE| if {file} is a writable file
filter({expr1}, {expr2})        List/Dict  remove items from {expr1} where
                                        {expr2} is 0
finddir({name}[, {path}[, {count}]])
                                String  find directory {name} in {path}
findfile({name}[, {path}[, {count}]])
                                String  find file {name} in {path}
float2nr({expr})                Number  convert Float {expr} to a Number
floor({expr})                   Float   round {expr} down
fmod({expr1}, {expr2})          Float   remainder of {expr1} / {expr2}
fnameescape({fname})            String  escape special characters in {fname}
fnamemodify({fname}, {mods})    String  modify file name
foldclosed({lnum})              Number  first line of fold at {lnum} if closed
foldclosedend({lnum})           Number  last line of fold at {lnum} if closed
foldlevel({lnum})               Number  fold level at {lnum}
foldtext()                      String  line displayed for closed fold
foldtextresult({lnum})          String  text for closed fold at {lnum}
foreground()                    Number  bring the Vim window to the foreground
funcref({name} [, {arglist}] [, {dict}])
                                Funcref reference to function {name}
function({name} [, {arglist}] [, {dict}])
                                Funcref named reference to function {name}
garbagecollect([{atexit}])      none    free memory, breaking cyclic references
get({list}, {idx} [, {def}])    any     get item {idx} from {list} or {def}
get({dict}, {key} [, {def}])    any     get item {key} from {dict} or {def}
get({func}, {what})             any     get property of funcref/partial {func}
getbufinfo([{expr}])            List    information about buffers
getbufline({expr}, {lnum} [, {end}])
                                List    lines {lnum} to {end} of buffer {expr}
getbufvar({expr}, {varname} [, {def}])
                                any     variable {varname} in buffer {expr}
getchar([expr])                 Number  get one character from the user
getcharmod()                    Number  modifiers for the last typed character
getcharsearch()                 Dict    last character search
getcmdline()                    String  return the current command-line
getcmdpos()                     Number  return cursor position in command-line
getcmdtype()                    String  return current command-line type
getcmdwintype()                 String  return current command-line window type
```

```
getcompletion({pat}, {type} [, {filtered}])
                                List    list of cmdline completion matches
getcurpos()                     List    position of the cursor
getcwd([{winnr} [, {tabnr}]])   String  get the current working directory
getfontname([{name}])           String  name of font being used
getfperm({fname})               String  file permissions of file {fname}
getfsize({fname})               Number  size in bytes of file {fname}
getftime({fname})               Number  last modification time of file
getftype({fname})               String  description of type of file {fname}
getline({lnum})                 String  line {lnum} of current buffer
getline({lnum}, {end})          List    lines {lnum} to {end} of current buffer
getloclist({nr}[, {what}])      List    list of location list items
getmatches()                    List    list of current matches
getpid()                        Number  process ID of Vim
getpos({expr})                  List    position of cursor, mark, etc.
getqflist([{what}])             List    list of quickfix items
getreg([{regname} [, 1 [, {list}]]])
                                String or List   contents of register
getregtype([{regname}])         String  type of register
gettabinfo([{expr}])            List    list of tab pages
gettabvar({nr}, {varname} [, {def}])
                                any     variable {varname} in tab {nr} or {def}
gettabwinvar({tabnr}, {winnr}, {name} [, {def}])
                                any     {name} in {winnr} in tab page {tabnr}
getwininfo([{winid}])           List    list of windows
getwinposx()                    Number  X coord in pixels of GUI Vim window
getwinposy()                    Number  Y coord in pixels of GUI Vim window
getwinvar({nr}, {varname} [, {def}])
                                any     variable {varname} in window {nr}
glob({expr} [, {nosuf} [, {list} [, {alllinks}]]])
                                any     expand file wildcards in {expr}
glob2regpat({expr})             String  convert a glob pat into a search pat
globpath({path}, {expr} [, {nosuf} [, {list} [, {alllinks}]]])
                                String  do glob({expr}) for all dirs in {path}
has({feature})                  Number  |TRUE| if feature {feature} supported
has_key({dict}, {key})          Number  |TRUE| if {dict} has entry {key}
haslocaldir([{winnr} [, {tabnr}]])
                                Number  |TRUE| if the window executed |:lcd|
hasmapto({what} [, {mode} [, {abbr}]])
                                Number  |TRUE| if mapping to {what} exists
histadd({history}, {item})      String  add an item to a history
histdel({history} [, {item}])   String  remove an item from a history
histget({history} [, {index}])  String  get the item {index} from a history
histnr({history})               Number  highest index of a history
hlexists({name})                Number  |TRUE| if highlight group {name} exists
hlID({name})                    Number  syntax ID of highlight group {name}
hostname()                      String  name of the machine Vim is running on
iconv({expr}, {from}, {to})     String  convert encoding of {expr}
indent({lnum})                  Number  indent of line {lnum}
index({list}, {expr} [, {start} [, {ic}]])
                                Number  index in {list} where {expr} appears
input({prompt} [, {text} [, {completion}]])
                                String  get input from the user
inputdialog({prompt} [, {text} [, {completion}]])
                                String  like input() but in a GUI dialog
inputlist({textlist})           Number  let the user pick from a choice list
inputrestore()                  Number  restore typeahead
inputsave()                     Number  save and clear typeahead
inputsecret({prompt} [, {text}]) String like input() but hiding the text
insert({list}, {item} [, {idx}]) List   insert {item} in {list} [before {idx}]
invert({expr})                  Number  bitwise invert
isdirectory({directory})        Number  |TRUE| if {directory} is a directory
```

```
islocked({expr})                Number  |TRUE| if {expr} is locked
isnan({expr})                   Number  |TRUE| if {expr} is NaN
items({dict})                   List    key-value pairs in {dict}
job_getchannel({job})           Channel get the channel handle for {job}
job_info({job})                 Dict    get information about {job}
job_setoptions({job}, {options}) none   set options for {job}
job_start({command} [, {options}])
                                Job     start a job
job_status({job})               String  get the status of {job}
job_stop({job} [, {how}])       Number  stop {job}
join({list} [, {sep}])          String  join {list} items into one String
js_decode({string})             any     decode JS style JSON
js_encode({expr})               String  encode JS style JSON
json_decode({string})           any     decode JSON
json_encode({expr})             String  encode JSON
keys({dict})                    List    keys in {dict}
len({expr})                     Number  the length of {expr}
libcall({lib}, {func}, {arg})   String  call {func} in library {lib} with {arg}
libcallnr({lib}, {func}, {arg}) Number  idem, but return a Number
line({expr})                    Number  line nr of cursor, last line or mark
line2byte({lnum})               Number  byte count of line {lnum}
lispindent({lnum})              Number  Lisp indent for line {lnum}
localtime()                     Number  current time
log({expr})                     Float   natural logarithm (base e) of {expr}
log10({expr})                   Float   logarithm of Float {expr} to base 10
luaeval({expr}[, {expr}])       any     evaluate |Lua| expression
map({expr1}, {expr2})           List/Dict  change each item in {expr1} to {expr}
maparg({name}[, {mode} [, {abbr} [, {dict}]]])
                                String or Dict
                                        rhs of mapping {name} in mode {mode}
mapcheck({name}[, {mode} [, {abbr}]])
                                String  check for mappings matching {name}
match({expr}, {pat}[, {start}[, {count}]])
                                Number  position where {pat} matches in {expr}
matchadd({group}, {pattern}[, {priority}[, {id} [, {dict}]]])
                                Number  highlight {pattern} with {group}
matchaddpos({group}, {pos}[, {priority}[, {id}[, {dict}]]])
                                Number  highlight positions with {group}
matcharg({nr})                  List    arguments of |:match|
matchdelete({id})               Number  delete match identified by {id}
matchend({expr}, {pat}[, {start}[, {count}]])
                                Number  position where {pat} ends in {expr}
matchlist({expr}, {pat}[, {start}[, {count}]])
                                List    match and submatches of {pat} in {expr}
matchstr({expr}, {pat}[, {start}[, {count}]])
                                String  {count}'th match of {pat} in {expr}
matchstrpos({expr}, {pat}[, {start}[, {count}]])
                                List    {count}'th match of {pat} in {expr}
max({expr})                     Number  maximum value of items in {expr}
min({expr})                     Number  minimum value of items in {expr}
mkdir({name} [, {path} [, {prot}]])
                                Number  create directory {name}
mode([expr])                    String  current editing mode
mzeval({expr})                  any     evaluate |MzScheme| expression
nextnonblank({lnum})            Number  line nr of non-blank line >= {lnum}
nr2char({expr}[, {utf8}])       String  single char with ASCII/UTF8 value {expr}
or({expr}, {expr})              Number  bitwise OR
pathshorten({expr})             String  shorten directory names in a path
perleval({expr})                any     evaluate |Perl| expression
pow({x}, {y})                   Float   {x} to the power of {y}
prevnonblank({lnum})            Number  line nr of non-blank line <= {lnum}
printf({fmt}, {expr1}...)       String  format text
```

```
pumvisible()                    Number  whether popup menu is visible
pyeval({expr})                  any     evaluate |Python| expression
py3eval({expr})                 any     evaluate |python3| expression
pyxeval({expr})                 any     evaluate |python_x| expression
range({expr} [, {max} [, {stride}]])
                                List    items from {expr} to {max}
readfile({fname} [, {binary} [, {max}]])
                                List    get list of lines from file {fname}
reltime([{start} [, {end}]])    List    get time value
reltimefloat({time})            Float   turn the time value into a Float
reltimestr({time})              String  turn time value into a String
remote_expr({server}, {string} [, {idvar} [, {timeout}]])
                                String  send expression
remote_foreground({server})     Number  bring Vim server to the foreground
remote_peek({serverid} [, {retvar}])
                                Number  check for reply string
remote_read({serverid} [, {timeout}])
                                String  read reply string
remote_send({server}, {string} [, {idvar}])
                                String  send key sequence
remote_startserver({name})      none    become server {name}
                                String  send key sequence
remove({list}, {idx} [, {end}]) any     remove items {idx}-{end} from {list}
remove({dict}, {key})           any     remove entry {key} from {dict}
rename({from}, {to})            Number  rename (move) file from {from} to {to}
repeat({expr}, {count})         String  repeat {expr} {count} times
resolve({filename})             String  get filename a shortcut points to
reverse({list})                 List    reverse {list} in-place
round({expr})                   Float   round off {expr}
screenattr({row}, {col})        Number  attribute at screen position
screenchar({row}, {col})        Number  character at screen position
screencol()                     Number  current cursor column
screenrow()                     Number  current cursor row
search({pattern} [, {flags} [, {stopline} [, {timeout}]]])
                                Number  search for {pattern}
searchdecl({name} [, {global} [, {thisblock}]])
                                Number  search for variable declaration
searchpair({start}, {middle}, {end} [, {flags} [, {skip} [...]]])
                                Number  search for other end of start/end pair
searchpairpos({start}, {middle}, {end} [, {flags} [, {skip} [...]]])
                                List    search for other end of start/end pair
searchpos({pattern} [, {flags} [, {stopline} [, {timeout}]]])
                                List    search for {pattern}
server2client({clientid}, {string})
                                Number  send reply string
serverlist()                    String  get a list of available servers
setbufline( {expr}, {lnum}, {line})
                                Number  set line {lnum} to {line} in buffer
                                        {expr}
setbufvar({expr}, {varname}, {val})
                                none    set {varname} in buffer {expr} to {val}
setcharsearch({dict})           Dict    set character search from {dict}
setcmdpos({pos})                Number  set cursor position in command-line
setfperm({fname}, {mode})       Number  set {fname} file permissions to {mode}
setline({lnum}, {line})         Number  set line {lnum} to {line}
setloclist({nr}, {list}[, {action}[, {what}]])
                                Number  modify location list using {list}
setmatches({list})              Number  restore a list of matches
setpos({expr}, {list})          Number  set the {expr} position to {list}
setqflist({list}[, {action}[, {what}]])
                                Number  modify quickfix list using {list}
setreg({n}, {v}[, {opt}])       Number  set register to value and type
```

```
settabvar({nr}, {varname}, {val}) none  set {varname} in tab page {nr} to {val}
settabwinvar({tabnr}, {winnr}, {varname}, {val})
                                none    set {varname} in window {winnr} in tab
                                        page {tabnr} to {val}
setwinvar({nr}, {varname}, {val}) none  set {varname} in window {nr} to {val}
sha256({string})                String  SHA256 checksum of {string}
shellescape({string} [, {special}])
                                String  escape {string} for use as shell
                                        command argument
shiftwidth()                    Number  effective value of 'shiftwidth'
simplify({filename})            String  simplify filename as much as possible
sin({expr})                     Float   sine of {expr}
sinh({expr})                    Float   hyperbolic sine of {expr}
sort({list} [, {func} [, {dict}]])
                                List    sort {list}, using {func} to compare
soundfold({word})               String  sound-fold {word}
spellbadword()                  String  badly spelled word at cursor
spellsuggest({word} [, {max} [, {capital}]])
                                List    spelling suggestions
split({expr} [, {pat} [, {keepempty}]])
                                List    make |List| from {pat} separated {expr}
sqrt({expr})                    Float   square root of {expr}
str2float({expr})               Float   convert String to Float
str2nr({expr} [, {base}])       Number  convert String to Number
strchars({expr} [, {skipcc}])   Number  character length of the String {expr}
strcharpart({str}, {start}[, {len}])
                                String  {len} characters of {str} at {start}
strdisplaywidth({expr} [, {col}]) Number display length of the String {expr}
strftime({format}[, {time}])    String  time in specified format
strgetchar({str}, {index})      Number  get char {index} from {str}
stridx({haystack}, {needle}[, {start}])
                                Number  index of {needle} in {haystack}
string({expr})                  String  String representation of {expr} value
strlen({expr})                  Number  length of the String {expr}
strpart({str}, {start}[, {len}])
                                String  {len} characters of {str} at {start}
strridx({haystack}, {needle} [, {start}])
                                Number  last index of {needle} in {haystack}
strtrans({expr})                String  translate string to make it printable
strwidth({expr})                Number  display cell length of the String {expr}
submatch({nr}[, {list}])        String or List
                                        specific match in ":s" or substitute()
substitute({expr}, {pat}, {sub}, {flags})
                                String  all {pat} in {expr} replaced with {sub}
synID({lnum}, {col}, {trans})   Number  syntax ID at {lnum} and {col}
synIDattr({synID}, {what} [, {mode}])
                                String  attribute {what} of syntax ID {synID}
synIDtrans({synID})             Number  translated syntax ID of {synID}
synconcealed({lnum}, {col})     List    info about concealing
synstack({lnum}, {col})         List    stack of syntax IDs at {lnum} and {col}
system({expr} [, {input}])      String  output of shell command/filter {expr}
systemlist({expr} [, {input}])  List    output of shell command/filter {expr}
tabpagebuflist([{arg}])         List    list of buffer numbers in tab page
tabpagenr([{arg}])              Number  number of current or last tab page
tabpagewinnr({tabarg}[, {arg}]) Number  number of current window in tab page
taglist({expr}[, {filename}])   List    list of tags matching {expr}
tagfiles()                      List    tags files used
tan({expr})                     Float   tangent of {expr}
tanh({expr})                    Float   hyperbolic tangent of {expr}
tempname()                      String  name for a temporary file
term_getaltscreen({buf})        Number  get the alternate screen flag
term_getattr({attr}, {what})    Number  get the value of attribute {what}
```

```
term_getcursor({buf})           List    get the cursor position of a terminal
term_getjob({buf})              Job     get the job associated with a terminal
term_getline({buf}, {row})      String  get a line of text from a terminal
term_getscrolled({buf})         Number  get the scroll count of a terminal
term_getsize({buf})             List    get the size of a terminal
term_getstatus({buf})           String  get the status of a terminal
term_gettitle({buf})            String  get the title of a terminal
term_getttty({buf}, [{input}])  String  get the tty name of a terminal
term_list()                     List    get the list of terminal buffers
term_scrape({buf}, {row})       List    get row of a terminal screen
term_sendkeys({buf}, {keys})    none    send keystrokes to a terminal
term_start({cmd}, {options})    Job     open a terminal window and run a job
term_wait({buf} [, {time}])     Number  wait for screen to be updated
test_alloc_fail({id}, {countdown}, {repeat})
                                none    make memory allocation fail
test_autochdir()                none    enable 'autochdir' during startup
test_feedinput()                none    add key sequence to input buffer
test_garbagecollect_now()       none    free memory right now for testing
test_ignore_error({expr})       none    ignore a specific error
test_null_channel()             Channel null value for testing
test_null_dict()                Dict    null value for testing
test_null_job()                 Job     null value for testing
test_null_list()                List    null value for testing
test_null_partial()             Funcref null value for testing
test_null_string()              String  null value for testing
test_override({expr}, {val})    none    test with Vim internal overrides
test_settime({expr})            none    set current time for testing
timer_info([{id}])              List    information about timers
timer_pause({id}, {pause})      none    pause or unpause a timer
timer_start({time}, {callback} [, {options}])
                                Number  create a timer
timer_stop({timer})             none    stop a timer
timer_stopall()                 none    stop all timers
tolower({expr})                 String  the String {expr} switched to lowercase
toupper({expr})                 String  the String {expr} switched to uppercase
tr({src}, {fromstr}, {tostr})   String  translate chars of {src} in {fromstr}
                                        to chars in {tostr}
trunc({expr})                   Float   truncate Float {expr}
type({name})                    Number  type of variable {name}
undofile({name})                String  undo file name for {name}
undotree()                      List    undo file tree
uniq({list} [, {func} [, {dict}]])
                                List    remove adjacent duplicates from a list
values({dict})                  List    values in {dict}
virtcol({expr})                 Number  screen column of cursor or mark
visualmode([expr])              String  last visual mode used
wildmenumode()                  Number  whether 'wildmenu' mode is active
win_findbuf({bufnr})            List    find windows containing {bufnr}
win_getid([{win} [, {tab}]])    Number  get window ID for {win} in {tab}
win_gotoid({expr})              Number  go to window with ID {expr}
win_id2tabwin({expr})           List    get tab and window nr from window ID
win_id2win({expr})              Number  get window nr from window ID
winbufnr({nr})                  Number  buffer number of window {nr}
wincol()                        Number  window column of the cursor
winheight({nr})                 Number  height of window {nr}
winline()                       Number  window line of the cursor
winnr([{expr}])                 Number  number of current window
winrestcmd()                    String  returns command to restore window sizes
winrestview({dict})             none    restore view of current window
winsaveview()                   Dict    save view of current window
winwidth({nr})                  Number  width of window {nr}
wordcount()                     Dict    get byte/char/word statistics
```

```
writefile({list}, {fname} [, {flags}])
                                Number  write list of lines to file {fname}
xor({expr}, {expr})             Number  bitwise XOR


abs({expr})                                                     *abs()*
                Return the absolute value of {expr}.  When {expr} evaluates to
                a |Float| abs() returns a |Float|.  When {expr} can be
                converted to a |Number| abs() returns a |Number|.  Otherwise
                abs() gives an error message and returns -1.
                Examples: >
                        echo abs(1.456)
<                       1.456  >
                        echo abs(-5.456)
<                       5.456  >
                        echo abs(-4)
<                       4
                {only available when compiled with the |+float| feature}


acos({expr})                                                    *acos()*
                Return the arc cosine of {expr} measured in radians, as a
                |Float| in the range of [0, pi].
                {expr} must evaluate to a |Float| or a |Number| in the range
                [-1, 1].
                Examples: >
                        :echo acos(0)
<                       1.570796 >
                        :echo acos(-0.5)
<                       2.094395
                {only available when compiled with the |+float| feature}


add({list}, {expr})                                             *add()*
                Append the item {expr} to |List| {list}.  Returns the
                resulting |List|.  Examples: >
                        :let alist = add([1, 2, 3], item)
                        :call add(mylist, "woodstock")
<               Note that when {expr} is a |List| it is appended as a single
                item.  Use |extend()| to concatenate |Lists|.
                Use |insert()| to add an item at another position.


and({expr}, {expr})                                             *and()*
                Bitwise AND on the two arguments.  The arguments are converted
                to a number.  A List, Dict or Float argument causes an error.
                Example: >
                        :let flag = and(bits, 0x80)


append({lnum}, {expr})                                          *append()*
                When {expr} is a |List|: Append each item of the |List| as a
                text line below line {lnum} in the current buffer.
                Otherwise append {expr} as one text line below line {lnum} in
                the current buffer.
                {lnum} can be zero to insert a line before the first one.
                Returns 1 for failure ({lnum} out of range or out of memory),
                0 for success.  Example: >
                        :let failed = append(line('$'), "# THE END")
                        :let failed = append(0, ["Chapter 1", "the beginning"])
<
                                                                *argc()*
```

```
argc()          The result is the number of files in the argument list of the
                current window.  See |arglist|.

                                                        *argidx()*
argidx()        The result is the current index in the argument list.  0 is
                the first file.  argc() - 1 is the last one.  See |arglist|.

                                                        *arglistid()*
arglistid([{winnr} [, {tabnr}]])
                Return the argument list ID.  This is a number which
                identifies the argument list being used.  Zero is used for the
                global argument list.  See |arglist|.
                Return -1 if the arguments are invalid.

                Without arguments use the current window.
                With {winnr} only use this window in the current tab page.
                With {winnr} and {tabnr} use the window in the specified tab
                page.
                {winnr} can be the window number or the |window-ID|.

                                                        *argv()*
argv([{nr}])    The result is the {nr}th file in the argument list of the
                current window.  See |arglist|.  "argv(0)" is the first one.
                Example: >
        :let i = 0
        :while i < argc()
        :  let f = escape(fnameescape(argv(i)), '.')
        :  exe 'amenu Arg.' . f . ' :e ' . f . '<CR>'
        :  let i = i + 1
        :endwhile
<               Without the {nr} argument a |List| with the whole |arglist| is
                returned.

                                                        *assert_equal()*
assert_equal({expected}, {actual} [, {msg}])
                When {expected} and {actual} are not equal an error message is
                added to |v:errors|.
                There is no automatic conversion, the String "4" is different
                from the Number 4.  And the number 4 is different from the
                Float 4.0.  The value of 'ignorecase' is not used here, case
                always matters.
                When {msg} is omitted an error in the form "Expected
                {expected} but got {actual}" is produced.
                Example: >
        assert_equal('foo', 'bar')
<               Will result in a string to be added to |v:errors|:
        test.vim line 12: Expected 'foo' but got 'bar' ~

assert_exception({error} [, {msg}])                     *assert_exception()*
                When v:exception does not contain the string {error} an error
                message is added to |v:errors|.
                This can be used to assert that a command throws an exception.
                Using the error number, followed by a colon, avoids problems
                with translations: >
                        try
                          commandthatfails
                          call assert_false(1, 'command should have failed')
                        catch
                          call assert_exception('E492:')
                        endtry

assert_fails({cmd} [, {error}])                                 *assert_fails()*
```

```
                Run {cmd} and add an error message to |v:errors| if it does
                NOT produce an error.
                When {error} is given it must match in |v:errmsg|.

assert_false({actual} [, {msg}])                                *assert_false()*
                When {actual} is not false an error message is added to
                |v:errors|, like with |assert_equal()|.
                A value is false when it is zero. When {actual} is not a
                number the assert fails.
                When {msg} is omitted an error in the form
                "Expected False but got {actual}" is produced.

assert_inrange({lower}, {upper}, {actual} [, {msg}])     *assert_inrange()*
                This asserts number values.  When {actual}  is lower than
                {lower} or higher than {upper} an error message is added to
                |v:errors|.
                When {msg} is omitted an error in the form
                "Expected range {lower} - {upper}, but got {actual}" is
                produced.

                                                                *assert_match()*
assert_match({pattern}, {actual} [, {msg}])
                When {pattern} does not match {actual} an error message is
                added to |v:errors|.

                {pattern} is used as with |=~|: The matching is always done
                like 'magic' was set and 'cpoptions' is empty, no matter what
                the actual value of 'magic' or 'cpoptions' is.

                {actual} is used as a string, automatic conversion applies.
                Use "^" and "$" to match with the start and end of the text.
                Use both to match the whole text.

                When {msg} is omitted an error in the form
                "Pattern {pattern} does not match {actual}" is produced.
                Example: >
        assert_match('^f.*o$', 'foobar')
<               Will result in a string to be added to |v:errors|:
        test.vim line 12: Pattern '^f.*o$' does not match 'foobar' ~

                                                        *assert_notequal()*
assert_notequal({expected}, {actual} [, {msg}])
                The opposite of `assert_equal()`: add an error message to
                |v:errors| when {expected} and {actual} are equal.

                                                        *assert_notmatch()*
assert_notmatch({pattern}, {actual} [, {msg}])
                The opposite of `assert_match()`: add an error message to
                |v:errors| when {pattern} matches {actual}.

assert_report({msg})                                    *assert_report()*
                Report a test failure directly, using {msg}.

assert_true({actual} [, {msg}])                         *assert_true()*
                When {actual} is not true an error message is added to
                |v:errors|, like with |assert_equal()|.
                A value is TRUE when it is a non-zero number.  When {actual}
                is not a number the assert fails.
                When {msg} is omitted an error in the form "Expected True but
                got {actual}" is produced.

asin({expr})                                            *asin()*
```

```
                Return the arc sine of {expr} measured in radians, as a |Float|
                in the range of [-pi/2, pi/2].
                {expr} must evaluate to a |Float| or a |Number| in the range
                [-1, 1].
                Examples: >
                        :echo asin(0.8)
<                       0.927295 >
                        :echo asin(-0.5)
<                       -0.523599
                {only available when compiled with the |+float| feature}


atan({expr})                                            *atan()*
                Return the principal value of the arc tangent of {expr}, in
                the range [-pi/2, +pi/2] radians, as a |Float|.
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo atan(100)
<                       1.560797 >
                        :echo atan(-4.01)
<                       -1.326405
                {only available when compiled with the |+float| feature}


atan2({expr1}, {expr2})                                 *atan2()*
                Return the arc tangent of {expr1} / {expr2}, measured in
                radians, as a |Float| in the range [-pi, pi].
                {expr1} and {expr2} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo atan2(-1, 1)
<                       -0.785398 >
                        :echo atan2(1, -1)
<                       2.356194
                {only available when compiled with the |+float| feature}

balloon_show({msg})                                     *balloon_show()*
                Show {msg} inside the balloon.
                Example: >
                        func GetBalloonContent()
                           " initiate getting the content
                           return ''
                        endfunc
                        set balloonexpr=GetBalloonContent()

                        func BalloonCallback(result)
                          call balloon_show(a:result)
                        endfunc
<
                The intended use is that fetching the content of the balloon
                is initiated from 'balloonexpr'.  It will invoke an
                asynchronous method, in which a callback invokes
                balloon_show().  The 'balloonexpr' itself can return an
                empty string or a placeholder.

                When showing a balloon is not possible nothing happens, no
                error message.
                {only available when compiled with the +balloon_eval feature}

                                                        *browse()*
browse({save}, {title}, {initdir}, {default})
                Put up a file requester.  This only works when "has("browse")"
                returns |TRUE| (only in some GUI versions).
```

```
                The input fields are:
                    {save}      when |TRUE|, select file to write
                    {title}     title for the requester
                    {initdir}   directory to start browsing in
                    {default}   default file name
                When the "Cancel" button is hit, something went wrong, or
                browsing is not possible, an empty string is returned.

                                                        *browsedir()*
browsedir({title}, {initdir})
                Put up a directory requester.  This only works when
                "has("browse")" returns |TRUE| (only in some GUI versions).
                On systems where a directory browser is not supported a file
                browser is used.  In that case: select a file in the directory
                to be used.
                The input fields are:
                    {title}     title for the requester
                    {initdir}   directory to start browsing in
                When the "Cancel" button is hit, something went wrong, or
                browsing is not possible, an empty string is returned.

bufexists({expr})                                       *bufexists()*
                The result is a Number, which is |TRUE| if a buffer called
                {expr} exists.
                If the {expr} argument is a number, buffer numbers are used.
                If the {expr} argument is a string it must match a buffer name
                exactly.  The name can be:
                - Relative to the current directory.
                - A full path.
                - The name of a buffer with 'buftype' set to "nofile".
                - A URL name.
                Unlisted buffers will be found.
                Note that help files are listed by their short name in the
                output of |:buffers|, but bufexists() requires using their
                long name to be able to find them.
                bufexists() may report a buffer exists, but to use the name
                with a |:buffer| command you may need to use |expand()|.  Esp
                for MS-Windows 8.3 names in the form "c:\DOCUME~1"
                Use "bufexists(0)" to test for the existence of an alternate
                file name.
                                                        *buffer_exists()*
                Obsolete name: buffer_exists().

buflisted({expr})                                       *buflisted()*
                The result is a Number, which is |TRUE| if a buffer called
                {expr} exists and is listed (has the 'buflisted' option set).
                The {expr} argument is used like with |bufexists()|.

bufloaded({expr})                                       *bufloaded()*
                The result is a Number, which is |TRUE| if a buffer called
                {expr} exists and is loaded (shown in a window or hidden).
                The {expr} argument is used like with |bufexists()|.

bufname({expr})                                         *bufname()*
                The result is the name of a buffer, as it is displayed by the
                ":ls" command.
                If {expr} is a Number, that buffer number's name is given.
                Number zero is the alternate buffer for the current window.
                If {expr} is a String, it is used as a |file-pattern| to match
                with the buffer names.  This is always done like 'magic' is
                set and 'cpoptions' is empty.  When there is more than one
                match an empty string is returned.
```

```
                "" or "%" can be used for the current buffer, "#" for the
                alternate buffer.
                A full match is preferred, otherwise a match at the start, end
                or middle of the buffer name is accepted.  If you only want a
                full match then put "^" at the start and "$" at the end of the
                pattern.
                Listed buffers are found first.  If there is a single match
                with a listed buffer, that one is returned.  Next unlisted
                buffers are searched for.
                If the {expr} is a String, but you want to use it as a buffer
                number, force it to be a Number by adding zero to it: >
                        :echo bufname("3" + 0)
<               If the buffer doesn't exist, or doesn't have a name, an empty
                string is returned. >
        bufname("#")            alternate buffer name
        bufname(3)              name of buffer 3
        bufname("%")            name of current buffer
        bufname("file2")        name of buffer where "file2" matches.
<                                                       *buffer_name()*
                Obsolete name: buffer_name().

                                                        *bufnr()*
bufnr({expr} [, {create}])
                The result is the number of a buffer, as it is displayed by
                the ":ls" command.  For the use of {expr}, see |bufname()|
                above.
                If the buffer doesn't exist, -1 is returned.  Or, if the
                {create} argument is present and not zero, a new, unlisted,
                buffer is created and its number is returned.
                bufnr("$") is the last buffer: >
        :let last_buffer = bufnr("$")
<               The result is a Number, which is the highest buffer number
                of existing buffers.  Note that not all buffers with a smaller
                number necessarily exist, because ":bwipeout" may have removed
                them.  Use bufexists() to test for the existence of a buffer.
                                                        *buffer_number()*
                Obsolete name: buffer_number().
                                                        *last_buffer_nr()*
                Obsolete name for bufnr("$"): last_buffer_nr().

bufwinid({expr})                                        *bufwinid()*
                The result is a Number, which is the |window-ID| of the first
                window associated with buffer {expr}.  For the use of {expr},
                see |bufname()| above.  If buffer {expr} doesn't exist or
                there is no such window, -1 is returned.  Example: >

        echo "A window containing buffer 1 is " . (bufwinid(1))
<
                Only deals with the current tab page.

bufwinnr({expr})                                        *bufwinnr()*
                The result is a Number, which is the number of the first
                window associated with buffer {expr}.  For the use of {expr},
                see |bufname()| above.  If buffer {expr} doesn't exist or
                there is no such window, -1 is returned.  Example: >

        echo "A window containing buffer 1 is " . (bufwinnr(1))

<               The number can be used with |CTRL-W_w| and ":wincmd w"
                |:wincmd|.
                Only deals with the current tab page.
```

```
byte2line({byte})                                       *byte2line()*
                Return the line number that contains the character at byte
                count {byte} in the current buffer.  This includes the
                end-of-line character, depending on the 'fileformat' option
                for the current buffer.  The first character has byte count
                one.
                Also see |line2byte()|, |go| and |:goto|.
                {not available when compiled without the |+byte_offset|
                feature}

byteidx({expr}, {nr})                                   *byteidx()*
                Return byte index of the {nr}'th character in the string
                {expr}.  Use zero for the first character, it returns zero.
                This function is only useful when there are multibyte
                characters, otherwise the returned value is equal to {nr}.
                Composing characters are not counted separately, their byte
                length is added to the preceding base character.  See
                |byteidxcomp()| below for counting composing characters
                separately.
                Example : >
                        echo matchstr(str, ".", byteidx(str, 3))
<               will display the fourth character.  Another way to do the
                same: >
                        let s = strpart(str, byteidx(str, 3))
                        echo strpart(s, 0, byteidx(s, 1))
<               Also see |strgetchar()| and |strcharpart()|.

                If there are less than {nr} characters -1 is returned.
                If there are exactly {nr} characters the length of the string
                in bytes is returned.

byteidxcomp({expr}, {nr})                                       *byteidxcomp()*
                Like byteidx(), except that a composing character is counted
                as a separate character.  Example: >
                        let s = 'e' . nr2char(0x301)
                        echo byteidx(s, 1)
                        echo byteidxcomp(s, 1)
                        echo byteidxcomp(s, 2)
<               The first and third echo result in 3 ('e' plus composing
                character is 3 bytes), the second echo results in 1 ('e' is
                one byte).
                Only works different from byteidx() when 'encoding' is set to
                a Unicode encoding.

call({func}, {arglist} [, {dict}])                      *call()* *E699*
                Call function {func} with the items in |List| {arglist} as
                arguments.
                {func} can either be a |Funcref| or the name of a function.
                a:firstline and a:lastline are set to the cursor line.
                Returns the return value of the called function.
                {dict} is for functions with the "dict" attribute.  It will be
                used to set the local variable "self". |Dictionary-function|

ceil({expr})                                                    *ceil()*
                Return the smallest integral value greater than or equal to
                {expr} as a |Float| (round up).
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        echo ceil(1.456)
<                       2.0  >
                        echo ceil(-5.456)
<                       -5.0  >
```

```
                        echo ceil(4.0)
<                       4.0
                {only available when compiled with the |+float| feature}

ch_canread({handle})                                            *ch_canread()*
                Return non-zero when there is something to read from {handle}.
                {handle} can be a Channel or a Job that has a Channel.

                This is useful to read from a channel at a convenient time,
                e.g. from a timer.

                Note that messages are dropped when the channel does not have
                a callback.  Add a close callback to avoid that.

                {only available when compiled with the |+channel| feature}

ch_close({handle})                                              *ch_close()*
                Close {handle}.  See |channel-close|.
                {handle} can be a Channel or a Job that has a Channel.
                A close callback is not invoked.

                {only available when compiled with the |+channel| feature}

ch_close_in({handle})                                           *ch_close_in()*
                Close the "in" part of {handle}.  See |channel-close-in|.
                {handle} can be a Channel or a Job that has a Channel.
                A close callback is not invoked.

                {only available when compiled with the |+channel| feature}

ch_evalexpr({handle}, {expr} [, {options}])                     *ch_evalexpr()*
                Send {expr} over {handle}.  The {expr} is encoded
                according to the type of channel.  The function cannot be used
                with a raw channel.  See |channel-use|.
                {handle} can be a Channel or a Job that has a Channel.
                                                                *E917*
                {options} must be a Dictionary.  It must not have a "callback"
                entry.  It can have a "timeout" entry to specify the timeout
                for this specific request.

                ch_evalexpr() waits for a response and returns the decoded
                expression.  When there is an error or timeout it returns an
                empty string.

                {only available when compiled with the |+channel| feature}

ch_evalraw({handle}, {string} [, {options}])            *ch_evalraw()*
                Send {string} over {handle}.
                {handle} can be a Channel or a Job that has a Channel.

                Works like |ch_evalexpr()|, but does not encode the request or
                decode the response.  The caller is responsible for the
                correct contents.  Also does not add a newline for a channel
                in NL mode, the caller must do that.  The NL in the response
                is removed.
                See |channel-use|.

                {only available when compiled with the |+channel| feature}

ch_getbufnr({handle}, {what})                               *ch_getbufnr()*
                Get the buffer number that {handle} is using for {what}.
                {handle} can be a Channel or a Job that has a Channel.
```

```
                {what} can be "err" for stderr, "out" for stdout or empty for
                socket output.
                Returns -1 when there is no buffer.
                {only available when compiled with the |+channel| feature}

ch_getjob({channel})                                            *ch_getjob()*
                Get the Job associated with {channel}.
                If there is no job calling |job_status()| on the returned Job
                will result in "fail".

                {only available when compiled with the |+channel| and
                |+job| features}

ch_info({handle})                                               *ch_info()*
                Returns a Dictionary with information about {handle}.  The
                items are:
                   "id"           number of the channel
                   "status"       "open", "buffered" or "closed", like
                                  ch_status()
                When opened with ch_open():
                   "hostname"     the hostname of the address
                   "port"         the port of the address
                   "sock_status"  "open" or "closed"
                   "sock_mode"    "NL", "RAW", "JSON" or "JS"
                   "sock_io"      "socket"
                   "sock_timeout" timeout in msec
                When opened with job_start():
                   "out_status"   "open", "buffered" or "closed"
                   "out_mode"     "NL", "RAW", "JSON" or "JS"
                   "out_io"       "null", "pipe", "file" or "buffer"
                   "out_timeout"  timeout in msec
                   "err_status"   "open", "buffered" or "closed"
                   "err_mode"     "NL", "RAW", "JSON" or "JS"
                   "err_io"       "out", "null", "pipe", "file" or "buffer"
                   "err_timeout"  timeout in msec
                   "in_status"    "open" or "closed"
                   "in_mode"      "NL", "RAW", "JSON" or "JS"
                   "in_io"        "null", "pipe", "file" or "buffer"
                   "in_timeout"   timeout in msec

ch_log({msg} [, {handle}])                                      *ch_log()*
                Write {msg} in the channel log file, if it was opened with
                |ch_logfile()|.
                When {handle} is passed the channel number is used for the
                message.
                {handle} can be a Channel or a Job that has a Channel.  The
                Channel must be open for the channel number to be used.

ch_logfile({fname} [, {mode}])                                  *ch_logfile()*
                Start logging channel activity to {fname}.
                When {fname} is an empty string: stop logging.

                When {mode} is omitted or "a" append to the file.
                When {mode} is "w" start with an empty file.

                The file is flushed after every message, on Unix you can use
                "tail -f" to see what is going on in real time.

                This function is not available in the |sandbox|.
                NOTE: the channel communication is stored in the file, be
                aware that this may contain confidential and privacy sensitive
                information, e.g. a password you type in a terminal window.
```

```
ch_open({address} [, {options}])                                    *ch_open()*
                Open a channel to {address}.  See |channel|.
                Returns a Channel.  Use |ch_status()| to check for failure.

                {address} has the form "hostname:port", e.g.,
                "localhost:8765".

                If {options} is given it must be a |Dictionary|.
                See |channel-open-options|.

                {only available when compiled with the |+channel| feature}

ch_read({handle} [, {options}])                                     *ch_read()*
                Read from {handle} and return the received message.
                {handle} can be a Channel or a Job that has a Channel.
                See |channel-more|.
                {only available when compiled with the |+channel| feature}

ch_readraw({handle} [, {options}])                          *ch_readraw()*
                Like ch_read() but for a JS and JSON channel does not decode
                the message.  See |channel-more|.
                {only available when compiled with the |+channel| feature}

ch_sendexpr({handle}, {expr} [, {options}])                         *ch_sendexpr()*
                Send {expr} over {handle}.  The {expr} is encoded
                according to the type of channel.  The function cannot be used
                with a raw channel.
                See |channel-use|.                                  *E912*
                {handle} can be a Channel or a Job that has a Channel.

                {only available when compiled with the |+channel| feature}

ch_sendraw({handle}, {string} [, {options}])                *ch_sendraw()*
                Send {string} over {handle}.
                Works like |ch_sendexpr()|, but does not encode the request or
                decode the response.  The caller is responsible for the
                correct contents.  Also does not add a newline for a channel
                in NL mode, the caller must do that.  The NL in the response
                is removed.
                See |channel-use|.

                {only available when compiled with the |+channel| feature}

ch_setoptions({handle}, {options})                          *ch_setoptions()*
                Set options on {handle}:
                        "callback"      the channel callback
                        "timeout"       default read timeout in msec
                        "mode"          mode for the whole channel
                See |ch_open()| for more explanation.
                {handle} can be a Channel or a Job that has a Channel.

                Note that changing the mode may cause queued messages to be
                lost.

                These options cannot be changed:
                        "waittime"      only applies to |ch_open()|

ch_status({handle} [, {options}])                                   *ch_status()*
                Return the status of {handle}:
                        "fail"          failed to open the channel
```

```
                        "open"          channel can be used
                        "buffered"      channel can be read, not written to
                        "closed"        channel can not be used
                    {handle} can be a Channel or a Job that has a Channel.
                    "buffered" is used when the channel was closed but there is
                    still data that can be obtained with |ch_read()|.

                    If {options} is given it can contain a "part" entry to specify
                    the part of the channel to return the status for: "out" or
                    "err".  For example, to get the error status: >
                            ch_status(job, {"part": "err"})
<
changenr()                                                  *changenr()*
                    Return the number of the most recent change.  This is the same
                    number as what is displayed with |:undolist| and can be used
                    with the |:undo| command.
                    When a change was made it is the number of that change.  After
                    redo it is the number of the redone change.  After undo it is
                    one less than the number of the undone change.

char2nr({expr}[, {utf8}])                                   *char2nr()*
                    Return number value of the first char in {expr}.  Examples: >
                            char2nr(" ")            returns 32
                            char2nr("ABC")          returns 65
<                   When {utf8} is omitted or zero, the current 'encoding' is used.
                    Example for "utf-8": >
                            char2nr("\E1")          returns 225
                            char2nr("\E1"[0])               returns 195
<                   With {utf8} set to 1, always treat as utf-8 characters.
                    A combining character is a separate character.
                    |nr2char()| does the opposite.

cindent({lnum})                                             *cindent()*
                    Get the amount of indent for line {lnum} according the C
                    indenting rules, as with 'cindent'.
                    The indent is counted in spaces, the value of 'tabstop' is
                    relevant.  {lnum} is used just like in |getline()|.
                    When {lnum} is invalid or Vim was not compiled the |+cindent|
                    feature, -1 is returned.
                    See |C-indenting|.

clearmatches()                                              *clearmatches()*
                    Clears all matches previously defined by |matchadd()| and the
                    |:match| commands.

                                                            *col()*
col({expr})         The result is a Number, which is the byte index of the column
                    position given with {expr}.  The accepted positions are:
                        .       the cursor position
                        $       the end of the cursor line (the result is the
                                number of bytes in the cursor line plus one)
                        'x      position of mark x (if the mark is not set, 0 is
                                returned)
                        v       In Visual mode: the start of the Visual area (the
                                cursor is the end).  When not in Visual mode
                                returns the cursor position.  Differs from |'<| in
                                that it's updated right away.
                    Additionally {expr} can be [lnum, col]: a |List| with the line
                    and column number. Most useful when the column is "$", to get
                    the last column of a specific line.  When "lnum" or "col" is
                    out of range then col() returns zero.
                    To get the line number use |line()|.  To get both use
```

```
                |getpos()|.
                For the screen column position use |virtcol()|.
                Note that only marks in the current file can be used.
                Examples: >
                        col(".")                column of cursor
                        col("$")                length of cursor line plus one
                        col("'t")               column of mark t
                        col("'" . markname)     column of mark markname
<               The first column is 1.  0 is returned for an error.
                For an uppercase mark the column may actually be in another
                buffer.
                For the cursor position, when 'virtualedit' is active, the
                column is one higher if the cursor is after the end of the
                line.  This can be used to obtain the column in Insert mode: >
                        :imap <F2> <C-O>:let save_ve = &ve<CR>
                                \<C-O>:set ve=all<CR>
                                \<C-O>:echo col(".") . "\n" <Bar>
                                \let &ve = save_ve<CR>
<

complete({startcol}, {matches})                 *complete()* *E785*
                Set the matches for Insert mode completion.
                Can only be used in Insert mode.  You need to use a mapping
                with CTRL-R = (see |i_CTRL-R|).  It does not work after CTRL-O
                or with an expression mapping.
                {startcol} is the byte offset in the line where the completed
                text start.  The text up to the cursor is the original text
                that will be replaced by the matches.  Use col('.') for an
                empty string.  "col('.') - 1" will replace one character by a
                match.
                {matches} must be a |List|.  Each |List| item is one match.
                See |complete-items| for the kind of items that are possible.
                Note that the after calling this function you need to avoid
                inserting anything that would cause completion to stop.
                The match can be selected with CTRL-N and CTRL-P as usual with
                Insert mode completion.  The popup menu will appear if
                specified, see |ins-completion-menu|.
                Example: >
        inoremap <F5> <C-R>=ListMonths()<CR>

        func! ListMonths()
          call complete(col('.'), ['January', 'February', 'March',
                \ 'April', 'May', 'June', 'July', 'August', 'September',
                \ 'October', 'November', 'December'])
          return ''
        endfunc
<               This isn't very useful, but it shows how it works.  Note that
                an empty string is returned to avoid a zero being inserted.

complete_add({expr})                            *complete_add()*
                Add {expr} to the list of matches.  Only to be used by the
                function specified with the 'completefunc' option.
                Returns 0 for failure (empty string or out of memory),
                1 when the match was added, 2 when the match was already in
                the list.
                See |complete-functions| for an explanation of {expr}.  It is
                the same as one item in the list that 'omnifunc' would return.

complete_check()                                *complete_check()*
                Check for a key typed while looking for completion matches.
                This is to be used when looking for matches takes some time.
                Returns |TRUE| when searching for matches is to be aborted,
```

```
                zero otherwise.
                Only to be used by the function specified with the
                'completefunc' option.

                                                    *confirm()*
confirm({msg} [, {choices} [, {default} [, {type}]]])
                Confirm() offers the user a dialog, from which a choice can be
                made.  It returns the number of the choice.  For the first
                choice this is 1.
                Note: confirm() is only supported when compiled with dialog
                support, see |+dialog_con| and |+dialog_gui|.

                {msg} is displayed in a |dialog| with {choices} as the
                alternatives.  When {choices} is missing or empty, "&OK" is
                used (and translated).
                {msg} is a String, use '\n' to include a newline.  Only on
                some systems the string is wrapped when it doesn't fit.

                {choices} is a String, with the individual choices separated
                by '\n', e.g. >
                        confirm("Save changes?", "&Yes\n&No\n&Cancel")
<               The letter after the '&' is the shortcut key for that choice.
                Thus you can type 'c' to select "Cancel".  The shortcut does
                not need to be the first letter: >
                        confirm("file has been modified", "&Save\nSave &All")
<               For the console, the first letter of each choice is used as
                the default shortcut key.

                The optional {default} argument is the number of the choice
                that is made if the user hits <CR>.  Use 1 to make the first
                choice the default one.  Use 0 to not set a default.  If
                {default} is omitted, 1 is used.

                The optional {type} argument gives the type of dialog.  This
                is only used for the icon of the GTK, Mac, Motif and Win32
                GUI.  It can be one of these values: "Error", "Question",
                "Info", "Warning" or "Generic".  Only the first character is
                relevant.  When {type} is omitted, "Generic" is used.

                If the user aborts the dialog by pressing <Esc>, CTRL-C,
                or another valid interrupt key, confirm() returns 0.

                An example: >
   :let choice = confirm("What do you want?", "&Apples\n&Oranges\n&Bananas", 2)
   :if choice == 0
   :    echo "make up your mind!"
   :elseif choice == 3
   :    echo "tasteful"
   :else
   :    echo "I prefer bananas myself."
   :endif
<               In a GUI dialog, buttons are used.  The layout of the buttons
                depends on the 'v' flag in 'guioptions'.  If it is included,
                the buttons are always put vertically.  Otherwise,  confirm()
                tries to put the buttons in one horizontal line.  If they
                don't fit, a vertical layout is used anyway.  For some systems
                the horizontal layout is always used.

                                                        *copy()*
copy({expr})    Make a copy of {expr}.  For Numbers and Strings this isn't
                different from using {expr} directly.
                When {expr} is a |List| a shallow copy is created.  This means
```

```
                that the original |List| can be changed without changing the
                copy, and vice versa.  But the items are identical, thus
                changing an item changes the contents of both |Lists|.
                A |Dictionary| is copied in a similar way as a |List|.
                Also see |deepcopy()|.

cos({expr})                                                     *cos()*
                Return the cosine of {expr}, measured in radians, as a |Float|.
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo cos(100)
<                       0.862319 >
                        :echo cos(-4.01)
<                       -0.646043
                {only available when compiled with the |+float| feature}


cosh({expr})                                                    *cosh()*
                Return the hyperbolic cosine of {expr} as a |Float| in the range
                [1, inf].
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo cosh(0.5)
<                       1.127626 >
                        :echo cosh(-0.5)
<                       -1.127626
                {only available when compiled with the |+float| feature}


count({comp}, {expr} [, {ic} [, {start}]])                      *count()*
                Return the number of times an item with value {expr} appears
                in |String|, |List| or |Dictionary| {comp}.

                If {start} is given then start with the item with this index.
                {start} can only be used with a |List|.

                When {ic} is given and it's |TRUE| then case is ignored.

                When {comp} is a string then the number of not overlapping
                occurrences of {expr} is returned.


                                                        *cscope_connection()*
cscope_connection([{num} , {dbpath} [, {prepend}]])
                Checks for the existence of a |cscope| connection.  If no
                parameters are specified, then the function returns:
                        0, if cscope was not available (not compiled in), or
                           if there are no cscope connections;
                        1, if there is at least one cscope connection.

                If parameters are specified, then the value of {num}
                determines how existence of a cscope connection is checked:

                {num}   Description of existence check
                -----   ------------------------------
                0       Same as no parameters (e.g., "cscope_connection()").
                1       Ignore {prepend}, and use partial string matches for
                        {dbpath}.
                2       Ignore {prepend}, and use exact string matches for
                        {dbpath}.
                3       Use {prepend}, use partial string matches for both
                        {dbpath} and {prepend}.
```

```
                4       Use {prepend}, use exact string matches for both
                        {dbpath} and {prepend}.

                Note: All string comparisons are case sensitive!

                Examples.  Suppose we had the following (from ":cs show"): >

  # pid    database name                       prepend path
  0 27664  cscope.out                         /usr/local
<
                Invocation                                      Return Val ~
                ----------                                      ---------- >
                cscope_connection()                                    1
                cscope_connection(1, "out")                            1
                cscope_connection(2, "out")                            0
                cscope_connection(3, "out")                            0
                cscope_connection(3, "out", "local")                   1
                cscope_connection(4, "out")                            0
                cscope_connection(4, "out", "local")                   0
                cscope_connection(4, "cscope.out", "/usr/local")       1
<
cursor({lnum}, {col} [, {off}])                         *cursor()*
cursor({list})
                Positions the cursor at the column (byte count) {col} in the
                line {lnum}.  The first column is one.

                When there is one argument {list} this is used as a |List|
                with two, three or four item:
                        [{lnum}, {col}]
                        [{lnum}, {col}, {off}]
                        [{lnum}, {col}, {off}, {curswant}]
                This is like the return value of |getpos()| or |getcurpos()|,
                but without the first item.

                Does not change the jumplist.
                If {lnum} is greater than the number of lines in the buffer,
                the cursor will be positioned at the last line in the buffer.
                If {lnum} is zero, the cursor will stay in the current line.
                If {col} is greater than the number of bytes in the line,
                the cursor will be positioned at the last character in the
                line.
                If {col} is zero, the cursor will stay in the current column.
                If {curswant} is given it is used to set the preferred column
                for vertical movement.  Otherwise {col} is used.

                When 'virtualedit' is used {off} specifies the offset in
                screen columns from the start of the character.  E.g., a
                position within a <Tab> or after the last character.
                Returns 0 when the position could be set, -1 otherwise.


deepcopy({expr}[, {noref}])                             *deepcopy()* *E698*
                Make a copy of {expr}.  For Numbers and Strings this isn't
                different from using {expr} directly.
                When {expr} is a |List| a full copy is created.  This means
                that the original |List| can be changed without changing the
                copy, and vice versa.  When an item is a |List| or
                |Dictionary|, a copy for it is made, recursively.  Thus
                changing an item in the copy does not change the contents of
                the original |List|.
                A |Dictionary| is copied in a similar way as a |List|.
                When {noref} is omitted or zero a contained |List| or
```

```
                |Dictionary| is only copied once.  All references point to
                this single copy.  With {noref} set to 1 every occurrence of a
                |List| or |Dictionary| results in a new copy.  This also means
                that a cyclic reference causes deepcopy() to fail.
                                                                *E724*
                Nesting is possible up to 100 levels.  When there is an item
                that refers back to a higher level making a deep copy with
                {noref} set to 1 will fail.
                Also see |copy()|.

delete({fname} [, {flags}])                                     *delete()*
                Without {flags} or with {flags} empty: Deletes the file by the
                name {fname}.  This also works when {fname} is a symbolic link.

                When {flags} is "d": Deletes the directory by the name
                {fname}.  This fails when directory {fname} is not empty.

                When {flags} is "rf": Deletes the directory by the name
                {fname} and everything in it, recursively.  BE CAREFUL!
                Note: on MS-Windows it is not possible to delete a directory
                that is being used.

                A symbolic link itself is deleted, not what it points to.

                The result is a Number, which is 0 if the delete operation was
                successful and -1 when the deletion failed or partly failed.

                Use |remove()| to delete an item from a |List|.
                To delete a line from the buffer use |:delete|.  Use |:exe|
                when the line number is in a variable.

                                                        *did_filetype()*
did_filetype()  Returns |TRUE| when autocommands are being executed and the
                FileType event has been triggered at least once.  Can be used
                to avoid triggering the FileType event again in the scripts
                that detect the file type. |FileType|
                Returns |FALSE| when `:setf FALLBACK` was used.
                When editing another file, the counter is reset, thus this
                really checks if the FileType event has been triggered for the
                current buffer.  This allows an autocommand that starts
                editing another buffer to set 'filetype' and load a syntax
                file.

diff_filler({lnum})                                     *diff_filler()*
                Returns the number of filler lines above line {lnum}.
                These are the lines that were inserted at this point in
                another diff'ed window.  These filler lines are shown in the
                display but don't exist in the buffer.
                {lnum} is used like with |getline()|.  Thus "." is the current
                line, "'m" mark m, etc.
                Returns 0 if the current window is not in diff mode.

diff_hlID({lnum}, {col})                                    *diff_hlID()*
                Returns the highlight ID for diff mode at line {lnum} column
                {col} (byte index).  When the current line does not have a
                diff change zero is returned.
                {lnum} is used like with |getline()|.  Thus "." is the current
                line, "'m" mark m, etc.
                {col} is 1 for the leftmost column, {lnum} is 1 for the first
                line.
                The highlight ID can be used with |synIDattr()| to obtain
                syntax information about the highlighting.
```

```
empty({expr})                                           *empty()*
                Return the Number 1 if {expr} is empty, zero otherwise.
                - A |List| or |Dictionary| is empty when it does not have any
                  items.
                - A Number and Float is empty when its value is zero.
                - |v:false|, |v:none| and |v:null| are empty, |v:true| is not.
                - A Job is empty when it failed to start.
                - A Channel is empty when it is closed.

                For a long |List| this is much faster than comparing the
                length with zero.

escape({string}, {chars})                               *escape()*
                Escape the characters in {chars} that occur in {string} with a
                backslash.  Example: >
                        :echo escape('c:\program files\vim', ' \')
<               results in: >
                        c:\\program\ files\\vim
<               Also see |shellescape()| and |fnameescape()|.

                                                        *eval()*
eval({string})  Evaluate {string} and return the result.  Especially useful to
                turn the result of |string()| back into the original value.
                This works for Numbers, Floats, Strings and composites of
                them.  Also works for |Funcref|s that refer to existing
                functions.

eventhandler()                                          *eventhandler()*
                Returns 1 when inside an event handler.  That is that Vim got
                interrupted while waiting for the user to type a character,
                e.g., when dropping a file on Vim.  This means interactive
                commands cannot be used.  Otherwise zero is returned.

executable({expr})                                      *executable()*
                This function checks if an executable with the name {expr}
                exists.  {expr} must be the name of the program without any
                arguments.
                executable() uses the value of $PATH and/or the normal
                searchpath for programs.                *PATHEXT*
                On MS-DOS and MS-Windows the ".exe", ".bat", etc. can
                optionally be included.  Then the extensions in $PATHEXT are
                tried.  Thus if "foo.exe" does not exist, "foo.exe.bat" can be
                found.  If $PATHEXT is not set then ".exe;.com;.bat;.cmd" is
                used.  A dot by itself can be used in $PATHEXT to try using
                the name without an extension.  When 'shell' looks like a
                Unix shell, then the name is also tried without adding an
                extension.
                On MS-DOS and MS-Windows it only checks if the file exists and
                is not a directory, not if it's really executable.
                On MS-Windows an executable in the same directory as Vim is
                always found.  Since this directory is added to $PATH it
                should also work to execute it |win32-PATH|.
                The result is a Number:
                        1       exists
                        0       does not exist
                        -1      not implemented on this system

execute({command} [, {silent}])                                 *execute()*
                Execute an Ex command or commands and return the output as a
                string.
                {command} can be a string or a List.  In case of a List the
```

```
                lines are executed one by one.
                This is equivalent to: >
                        redir => var
                        {command}
                        redir END
<
                The optional {silent} argument can have these values:
                        ""              no `:silent` used
                        "silent"        `:silent` used
                        "silent!"       `:silent!` used
                The default is "silent".  Note that with "silent!", unlike
                `:redir`, error messages are dropped.  When using an external
                command the screen may be messed up, use `system()` instead.
                                                        *E930*
                It is not possible to use `:redir` anywhere in {command}.

                To get a list of lines use |split()| on the result: >
                        split(execute('args'), "\n")

<               When used recursively the output of the recursive call is not
                included in the output of the higher level call.

exepath({expr})                                         *exepath()*
                If {expr} is an executable and is either an absolute path, a
                relative path or found in $PATH, return the full path.
                Note that the current directory is used when {expr} starts
                with "./", which may be a problem for Vim: >
                        echo exepath(v:progpath)
<               If {expr} cannot be found in $PATH or is not executable then
                an empty string is returned.

                                                        *exists()*
exists({expr})  The result is a Number, which is |TRUE| if {expr} is defined,
                zero otherwise.

                For checking for a supported feature use |has()|.
                For checking if a file exists use |filereadable()|.

                The {expr} argument is a string, which contains one of these:
                        &option-name    Vim option (only checks if it exists,
                                        not if it really works)
                        +option-name    Vim option that works.
                        $ENVNAME        environment variable (could also be
                                        done by comparing with an empty
                                        string)
                        *funcname       built-in function (see |functions|)
                                        or user defined function (see
                                        |user-functions|). Also works for a
                                        variable that is a Funcref.
                        varname         internal variable (see
                                        |internal-variables|).  Also works
                                        for |curly-braces-names|, |Dictionary|
                                        entries, |List| items, etc.  Beware
                                        that evaluating an index may cause an
                                        error message for an invalid
                                        expression.  E.g.: >
                                           :let l = [1, 2, 3]
                                           :echo exists("l[5]")
<                                          0 >
                                           :echo exists("l[xx]")
<                                          E121: Undefined variable: xx
                                           0
```

```
                        :cmdname        Ex command: built-in command, user
                                        command or command modifier |:command|.
                                        Returns:
                                        1  for match with start of a command
                                        2  full match with a command
                                        3  matches several user commands
                                        To check for a supported command
                                        always check the return value to be 2.
                        :2match         The |:2match| command.
                        :3match         The |:3match| command.
                        #event          autocommand defined for this event
                        #event#pattern  autocommand defined for this event and
                                        pattern (the pattern is taken
                                        literally and compared to the
                                        autocommand patterns character by
                                        character)
                        #group          autocommand group exists
                        #group#event    autocommand defined for this group and
                                        event.
                        #group#event#pattern
                                        autocommand defined for this group,
                                        event and pattern.
                        ##event         autocommand for this event is
                                        supported.

                Examples: >
                        exists("&shortname")
                        exists("$HOSTNAME")
                        exists("*strftime")
                        exists("*s:MyFunc")
                        exists("bufcount")
                        exists(":Make")
                        exists("#CursorHold")
                        exists("#BufReadPre#*.gz")
                        exists("#filetypeindent")
                        exists("#filetypeindent#FileType")
                        exists("#filetypeindent#FileType#*")
                        exists("##ColorScheme")
<               There must be no space between the symbol (&/$/*/#) and the
                name.
                There must be no extra characters after the name, although in
                a few cases this is ignored.  That may become more strict in
                the future, thus don't count on it!
                Working example: >
                        exists(":make")
<               NOT working example: >
                        exists(":make install")

<               Note that the argument must be a string, not the name of the
                variable itself.  For example: >
                        exists(bufcount)
<               This doesn't check for existence of the "bufcount" variable,
                but gets the value of "bufcount", and checks if that exists.

exp({expr})                                             *exp()*
                Return the exponential of {expr} as a |Float| in the range
                [0, inf].
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo exp(2)
<                       7.389056 >
                        :echo exp(-1)
```

```
<                       0.367879
                {only available when compiled with the |+float| feature}


expand({expr} [, {nosuf} [, {list}]])                           *expand()*
                Expand wildcards and the following special keywords in {expr}.
                'wildignorecase' applies.

                If {list} is given and it is |TRUE|, a List will be returned.
                Otherwise the result is a String and when there are several
                matches, they are separated by <NL> characters.  [Note: in
                version 5.0 a space was used, which caused problems when a
                file name contains a space]

                If the expansion fails, the result is an empty string.  A name
                for a non-existing file is not included, unless {expr} does
                not start with '%', '#' or '<', see below.

                When {expr} starts with '%', '#' or '<', the expansion is done
                like for the |cmdline-special| variables with their associated
                modifiers.  Here is a short overview:

                        %               current file name
                        #               alternate file name
                        #n              alternate file name n
                        <cfile>         file name under the cursor
                        <afile>         autocmd file name
                        <abuf>          autocmd buffer number (as a String!)
                        <amatch>        autocmd matched name
                        <sfile>         sourced script file or function name
                        <slnum>         sourced script file line number
                        <cword>         word under the cursor
                        <cWORD>         WORD under the cursor
                        <client>        the {clientid} of the last received
                                        message |server2client()|
                Modifiers:
                        :p              expand to full path
                        :h              head (last path component removed)
                        :t              tail (last path component only)
                        :r              root (one extension removed)
                        :e              extension only

                Example: >
                        :let &tags = expand("%:p:h") . "/tags"
<               Note that when expanding a string that starts with '%', '#' or
                '<', any following text is ignored.  This does NOT work: >
                        :let doesntwork = expand("%:h.bak")
<               Use this: >
                        :let doeswork = expand("%:h") . ".bak"
<               Also note that expanding "<cfile>" and others only returns the
                referenced file name without further expansion.  If "<cfile>"
                is "~/.cshrc", you need to do another expand() to have the
                "~/" expanded into the path of the home directory: >
                        :echo expand(expand("<cfile>"))
<
                There cannot be white space between the variables and the
                following modifier.  The |fnamemodify()| function can be used
                to modify normal file names.

                When using '%' or '#', and the current or alternate file name
                is not defined, an empty string is used.  Using "%:p" in a
                buffer with no name, results in the current directory, with a
```

```
                '/' added.

                When {expr} does not start with '%', '#' or '<', it is
                expanded like a file name is expanded on the command line.
                'suffixes' and 'wildignore' are used, unless the optional
                {nosuf} argument is given and it is |TRUE|.
                Names for non-existing files are included.  The "**" item can
                be used to search in a directory tree.  For example, to find
                all "README" files in the current directory and below: >
                        :echo expand("**/README")
<
                Expand() can also be used to expand variables and environment
                variables that are only known in a shell.  But this can be
                slow, because a shell may be used to do the expansion.  See
                |expr-env-expand|.
                The expanded variable is still handled like a list of file
                names.  When an environment variable cannot be expanded, it is
                left unchanged.  Thus ":echo expand('$FOOBAR')" results in
                "$FOOBAR".

                See |glob()| for finding existing files.  See |system()| for
                getting the raw output of an external command.

extend({expr1}, {expr2} [, {expr3}])                    *extend()*
                {expr1} and {expr2} must be both |Lists| or both
                |Dictionaries|.

                If they are |Lists|: Append {expr2} to {expr1}.
                If {expr3} is given insert the items of {expr2} before item
                {expr3} in {expr1}.  When {expr3} is zero insert before the
                first item.  When {expr3} is equal to len({expr1}) then
                {expr2} is appended.
                Examples: >
                        :echo sort(extend(mylist, [7, 5]))
                        :call extend(mylist, [2, 3], 1)
<               When {expr1} is the same List as {expr2} then the number of
                items copied is equal to the original length of the List.
                E.g., when {expr3} is 1 you get N new copies of the first item
                (where N is the original length of the List).
                Use |add()| to concatenate one item to a list.  To concatenate
                two lists into a new list use the + operator: >
                        :let newlist = [1, 2, 3] + [4, 5]
<
                If they are |Dictionaries|:
                Add all entries from {expr2} to {expr1}.
                If a key exists in both {expr1} and {expr2} then {expr3} is
                used to decide what to do:
                {expr3} = "keep": keep the value of {expr1}
                {expr3} = "force": use the value of {expr2}
                {expr3} = "error": give an error message               *E737*
                When {expr3} is omitted then "force" is assumed.

                {expr1} is changed when {expr2} is not empty.  If necessary
                make a copy of {expr1} first.
                {expr2} remains unchanged.
                When {expr1} is locked and {expr2} is not empty the operation
                fails.
                Returns {expr1}.


feedkeys({string} [, {mode}])                           *feedkeys()*
                Characters in {string} are queued for processing as if they
```

```
                come from a mapping or were typed by the user.
                By default the string is added to the end of the typeahead
                buffer, thus if a mapping is still being executed the
                characters come after them.  Use the 'i' flag to insert before
                other characters, they will be executed next, before any
                characters from a mapping.
                The function does not wait for processing of keys contained in
                {string}.
                To include special keys into {string}, use double-quotes
                and "\..." notation |expr-quote|. For example,
                feedkeys("\<CR>") simulates pressing of the <Enter> key. But
                feedkeys('\<CR>') pushes 5 characters.
                If {mode} is absent, keys are remapped.
                {mode} is a String, which can contain these character flags:
                'm'     Remap keys. This is default.
                'n'     Do not remap keys.
                't'     Handle keys as if typed; otherwise they are handled as
                        if coming from a mapping.  This matters for undo,
                        opening folds, etc.
                'i'     Insert the string instead of appending (see above).
                'x'     Execute commands until typeahead is empty.  This is
                        similar to using ":normal!".  You can call feedkeys()
                        several times without 'x' and then one time with 'x'
                        (possibly with an empty {string}) to execute all the
                        typeahead.  Note that when Vim ends in Insert mode it
                        will behave as if <Esc> is typed, to avoid getting
                        stuck, waiting for a character to be typed before the
                        script continues.
                '!'     When used with 'x' will not end Insert mode. Can be
                        used in a test when a timer is set to exit Insert mode
                        a little later.  Useful for testing CursorHoldI.

                Return value is always 0.

filereadable({file})                                    *filereadable()*
                The result is a Number, which is |TRUE| when a file with the
                name {file} exists, and can be read.  If {file} doesn't exist,
                or is a directory, the result is |FALSE|.  {file} is any
                expression, which is used as a String.
                If you don't care about the file being readable you can use
                |glob()|.
                                                        *file_readable()*
                Obsolete name: file_readable().


filewritable({file})                                    *filewritable()*
                The result is a Number, which is 1 when a file with the
                name {file} exists, and can be written.  If {file} doesn't
                exist, or is not writable, the result is 0.  If {file} is a
                directory, and we can write to it, the result is 2.


filter({expr1}, {expr2})                                *filter()*
                {expr1} must be a |List| or a |Dictionary|.
                For each item in {expr1} evaluate {expr2} and when the result
                is zero remove the item from the |List| or |Dictionary|.
                {expr2} must be a |string| or |Funcref|.

                If {expr2} is a |string|, inside {expr2} |v:val| has the value
                of the current item.  For a |Dictionary| |v:key| has the key
                of the current item and for a |List| |v:key| has the index of
                the current item.
```

```
                Examples: >
                        call filter(mylist, 'v:val !~ "OLD"')
<               Removes the items where "OLD" appears. >
                        call filter(mydict, 'v:key >= 8')
<               Removes the items with a key below 8. >
                        call filter(var, 0)
<               Removes all the items, thus clears the |List| or |Dictionary|.

                Note that {expr2} is the result of expression and is then
                used as an expression again.  Often it is good to use a
                |literal-string| to avoid having to double backslashes.

                If {expr2} is a |Funcref| it must take two arguments:
                        1. the key or the index of the current item.
                        2. the value of the current item.
                The function must return |TRUE| if the item should be kept.
                Example that keeps the odd items of a list: >
                        func Odd(idx, val)
                          return a:idx % 2 == 1
                        endfunc
                        call filter(mylist, function('Odd'))
<               It is shorter when using a |lambda|: >
                        call filter(myList, {idx, val -> idx * val <= 42})
<               If you do not use "val" you can leave it out: >
                        call filter(myList, {idx -> idx % 2 == 1})
<
                The operation is done in-place.  If you want a |List| or
                |Dictionary| to remain unmodified make a copy first: >
                        :let l = filter(copy(mylist), 'v:val =~ "KEEP"')

<               Returns {expr1}, the |List| or |Dictionary| that was filtered.
                When an error is encountered while evaluating {expr2} no
                further items in {expr1} are processed.  When {expr2} is a
                Funcref errors inside a function are ignored, unless it was
                defined with the "abort" flag.


finddir({name}[, {path}[, {count}]])                            *finddir()*
                Find directory {name} in {path}.  Supports both downwards and
                upwards recursive directory searches.  See |file-searching|
                for the syntax of {path}.
                Returns the path of the first found match.  When the found
                directory is below the current directory a relative path is
                returned.  Otherwise a full path is returned.
                If {path} is omitted or empty then 'path' is used.
                If the optional {count} is given, find {count}'s occurrence of
                {name} in {path} instead of the first one.
                When {count} is negative return all the matches in a |List|.
                This is quite similar to the ex-command |:find|.
                {only available when compiled with the |+file_in_path|
                feature}

findfile({name}[, {path}[, {count}]])                           *findfile()*
                Just like |finddir()|, but find a file instead of a directory.
                Uses 'suffixesadd'.
                Example: >
                        :echo findfile("tags.vim", ".;")
<               Searches from the directory of the current file upwards until
                it finds the file "tags.vim".

float2nr({expr})                                        *float2nr()*
                Convert {expr} to a Number by omitting the part after the
```

```
                decimal point.
                {expr} must evaluate to a |Float| or a Number.
                When the value of {expr} is out of range for a |Number| the
                result is truncated to 0x7fffffff or -0x7fffffff (or when
                64-bit Number support is enabled, 0x7fffffffffffffff or
                -0x7fffffffffffffff).  NaN results in -0x80000000 (or when
                64-bit Number support is enabled, -0x8000000000000000).
                Examples: >
                        echo float2nr(3.95)
<                       3  >
                        echo float2nr(-23.45)
<                       -23  >
                        echo float2nr(1.0e100)
<                       2147483647  (or 9223372036854775807) >
                        echo float2nr(-1.0e150)
<                       -2147483647 (or -9223372036854775807) >
                        echo float2nr(1.0e-100)
<                       0
                {only available when compiled with the |+float| feature}


floor({expr})                                                   *floor()*
                Return the largest integral value less than or equal to
                {expr} as a |Float| (round down).
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        echo floor(1.856)
<                       1.0  >
                        echo floor(-5.456)
<                       -6.0  >
                        echo floor(4.0)
<                       4.0
                {only available when compiled with the |+float| feature}


fmod({expr1}, {expr2})                                      *fmod()*
                Return the remainder of {expr1} / {expr2}, even if the
                division is not representable.  Returns {expr1} - i * {expr2}
                for some integer i such that if {expr2} is non-zero, the
                result has the same sign as {expr1} and magnitude less than
                the magnitude of {expr2}.  If {expr2} is zero, the value
                returned is zero.  The value returned is a |Float|.
                {expr1} and {expr2} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo fmod(12.33, 1.22)
<                       0.13 >
                        :echo fmod(-12.33, 1.22)
<                       -0.13
                {only available when compiled with |+float| feature}


fnameescape({string})                                   *fnameescape()*
                Escape {string} for use as file name command argument.  All
                characters that have a special meaning, such as '%' and '|'
                are escaped with a backslash.
                For most systems the characters escaped are
                " \t\n*?[{`$\\%#'\"|!<".  For systems where a backslash
                appears in a filename, it depends on the value of 'isfname'.
                A leading '+' and '>' is also escaped (special after |:edit|
                and |:write|).  And a "-" by itself (special after |:cd|).
                Example: >
                        :let fname = '+some str%nge|name'
```

```
                        :exe "edit " . fnameescape(fname)
<               results in executing: >
                        edit \+some\ str\%nge\|name

fnamemodify({fname}, {mods})                            *fnamemodify()*
               Modify file name {fname} according to {mods}.  {mods} is a
               string of characters like it is used for file names on the
               command line.  See |filename-modifiers|.
               Example: >
                        :echo fnamemodify("main.c", ":p:h")
<              results in: >
                        /home/mool/vim/vim/src
<              Note: Environment variables don't work in {fname}, use
               |expand()| first then.

foldclosed({lnum})                                      *foldclosed()*
               The result is a Number.  If the line {lnum} is in a closed
               fold, the result is the number of the first line in that fold.
               If the line {lnum} is not in a closed fold, -1 is returned.

foldclosedend({lnum})                                   *foldclosedend()*
               The result is a Number.  If the line {lnum} is in a closed
               fold, the result is the number of the last line in that fold.
               If the line {lnum} is not in a closed fold, -1 is returned.

foldlevel({lnum})                                       *foldlevel()*
               The result is a Number, which is the foldlevel of line {lnum}
               in the current buffer.  For nested folds the deepest level is
               returned.  If there is no fold at line {lnum}, zero is
               returned.  It doesn't matter if the folds are open or closed.
               When used while updating folds (from 'foldexpr') -1 is
               returned for lines where folds are still to be updated and the
               foldlevel is unknown.  As a special case the level of the
               previous line is usually available.

                                                        *foldtext()*
foldtext()     Returns a String, to be displayed for a closed fold.  This is
               the default function used for the 'foldtext' option and should
               only be called from evaluating 'foldtext'.  It uses the
               |v:foldstart|, |v:foldend| and |v:folddashes| variables.
               The returned string looks like this: >
                        +-- 45 lines: abcdef
<              The number of leading dashes depends on the foldlevel.  The
               "45" is the number of lines in the fold.  "abcdef" is the text
               in the first non-blank line of the fold.  Leading white space,
               "//" or "/*" and the text from the 'foldmarker' and
               'commentstring' options is removed.
               When used to draw the actual foldtext, the rest of the line
               will be filled with the fold char from the 'fillchars'
               setting.
               {not available when compiled without the |+folding| feature}

foldtextresult({lnum})                                  *foldtextresult()*
               Returns the text that is displayed for the closed fold at line
               {lnum}.  Evaluates 'foldtext' in the appropriate context.
               When there is no closed fold at {lnum} an empty string is
               returned.
               {lnum} is used like with |getline()|.  Thus "." is the current
               line, "'m" mark m, etc.
               Useful when exporting folded text, e.g., to HTML.
               {not available when compiled without the |+folding| feature}
```

```
                                                        *foreground()*
foreground()    Move the Vim window to the foreground.  Useful when sent from
                a client to a Vim server. |remote_send()|
                On Win32 systems this might not work, the OS does not always
                allow a window to bring itself to the foreground.  Use
                |remote_foreground()| instead.
                {only in the Win32, Athena, Motif and GTK GUI versions and the
                Win32 console version}

                                                *funcref()*
funcref({name} [, {arglist}] [, {dict}])
                Just like |function()|, but the returned Funcref will lookup
                the function by reference, not by name.  This matters when the
                function {name} is redefined later.

                Unlike |function()|, {name} must be an existing user function.
                Also for autoloaded functions. {name} cannot be a builtin
                function.

                                        *function()* *E700* *E922* *E923*
function({name} [, {arglist}] [, {dict}])
                Return a |Funcref| variable that refers to function {name}.
                {name} can be the name of a user defined function or an
                internal function.

                {name} can also be a Funcref or a partial.  When it is a
                partial the dict stored in it will be used and the {dict}
                argument is not allowed. E.g.: >
                        let FuncWithArg = function(dict.Func, [arg])
                        let Broken = function(dict.Func, [arg], dict)
<
                When using the Funcref the function will be found by {name},
                also when it was redefined later.  Use |funcref()| to keep the
                same function.

                When {arglist} or {dict} is present this creates a partial.
                That means the argument list and/or the dictionary is stored in
                the Funcref and will be used when the Funcref is called.

                The arguments are passed to the function in front of other
                arguments.  Example: >
                        func Callback(arg1, arg2, name)
                        ...
                        let Func = function('Callback', ['one', 'two'])
                        ...
                        call Func('name')
<               Invokes the function as with: >
                        call Callback('one', 'two', 'name')

<               The function() call can be nested to add more arguments to the
                Funcref.  The extra arguments are appended to the list of
                arguments.  Example: >
                        func Callback(arg1, arg2, name)
                        ...
                        let Func = function('Callback', ['one'])
                        let Func2 = function(Func, ['two'])
                        ...
                        call Func2('name')
<               Invokes the function as with: >
                        call Callback('one', 'two', 'name')

<               The Dictionary is only useful when calling a "dict" function.
```

```
                In that case the {dict} is passed in as "self". Example: >
                        function Callback() dict
                           echo "called for " . self.name
                        endfunction
                        ...
                        let context = {"name": "example"}
                        let Func = function('Callback', context)
                        ...
                        call Func()     " will echo: called for example
<               The use of function() is not needed when there are no extra
                arguments, these two are equivalent: >
                        let Func = function('Callback', context)
                        let Func = context.Callback

<               The argument list and the Dictionary can be combined: >
                        function Callback(arg1, count) dict
                        ...
                        let context = {"name": "example"}
                        let Func = function('Callback', ['one'], context)
                        ...
                        call Func(500)
<               Invokes the function as with: >
                        call context.Callback('one', 500)


garbagecollect([{atexit}])                              *garbagecollect()*
                Cleanup unused |Lists|, |Dictionaries|, |Channels| and |Jobs|
                that have circular references.

                There is hardly ever a need to invoke this function, as it is
                automatically done when Vim runs out of memory or is waiting
                for the user to press a key after 'updatetime'.  Items without
                circular references are always freed when they become unused.
                This is useful if you have deleted a very big |List| and/or
                |Dictionary| with circular references in a script that runs
                for a long time.

                When the optional {atexit} argument is one, garbage
                collection will also be done when exiting Vim, if it wasn't
                done before.  This is useful when checking for memory leaks.

                The garbage collection is not done immediately but only when
                it's safe to perform.  This is when waiting for the user to
                type a character.  To force garbage collection immediately use
                |test_garbagecollect_now()|.

get({list}, {idx} [, {default}])                        *get()*
                Get item {idx} from |List| {list}.  When this item is not
                available return {default}.  Return zero when {default} is
                omitted.
get({dict}, {key} [, {default}])
                Get item with key {key} from |Dictionary| {dict}.  When this
                item is not available return {default}.  Return zero when
                {default} is omitted.
get({func}, {what})
                Get an item with from Funcref {func}.  Possible values for
                {what} are:
                        "name"  The function name
                        "func"  The function
                        "dict"  The dictionary
                        "args"  The list with arguments
```

getbufinfo()

```
getbufinfo([{expr}])
getbufinfo([{dict}])
```

Get information about buffers as a List of Dictionaries.

Without an argument information about all the buffers is returned.

When the argument is a Dictionary only the buffers matching the specified criteria are returned. The following keys can be specified in {dict}:

buflisted	include only listed buffers.
bufloaded	include only loaded buffers.

Otherwise, {expr} specifies a particular buffer to return information for. For the use of {expr}, see |bufname()| above. If the buffer is found the returned List has one item. Otherwise the result is an empty list.

Each returned List item is a dictionary with the following entries:

bufnr	buffer number.
changed	TRUE if the buffer is modified.
changedtick	number of changes made to the buffer.
hidden	TRUE if the buffer is hidden.
listed	TRUE if the buffer is listed.
lnum	current line number in buffer.
loaded	TRUE if the buffer is loaded.
name	full path to the file in the buffer.
signs	list of signs placed in the buffer. Each list item is a dictionary with the following fields: id sign identifier lnum line number name sign name
variables	a reference to the dictionary with buffer-local variables.
windows	list of |window-ID|s that display this buffer

Examples: >

```
    for buf in getbufinfo()
        echo buf.name
    endfor
    for buf in getbufinfo({'buflisted':1})
        if buf.changed
            ....
        endif
    endfor
<
```

To get buffer-local options use: >

```
    getbufvar({bufnr}, '&')

<
```

getbufline()

```
getbufline({expr}, {lnum} [, {end}])
```

Return a |List| with the lines starting from {lnum} to {end} (inclusive) in the buffer {expr}. If {end} is omitted, a |List| with only the line {lnum} is returned.

For the use of {expr}, see |bufname()| above.

```
                For {lnum} and {end} "$" can be used for the last line of the
                buffer.  Otherwise a number must be used.

                When {lnum} is smaller than 1 or bigger than the number of
                lines in the buffer, an empty |List| is returned.

                When {end} is greater than the number of lines in the buffer,
                it is treated as {end} is set to the number of lines in the
                buffer.  When {end} is before {lnum} an empty |List| is
                returned.

                This function works only for loaded buffers.  For unloaded and
                non-existing buffers, an empty |List| is returned.

                Example: >
                        :let lines = getbufline(bufnr("myfile"), 1, "$")

getbufvar({expr}, {varname} [, {def}])                          *getbufvar()*
                The result is the value of option or local buffer variable
                {varname} in buffer {expr}.  Note that the name without "b:"
                must be used.
                When {varname} is empty returns a dictionary with all the
                buffer-local variables.
                When {varname} is equal to "&" returns a dictionary with all
                the buffer-local options.
                Otherwise, when {varname} starts with "&" returns the value of
                a buffer-local option.
                This also works for a global or buffer-local option, but it
                doesn't work for a global variable, window-local variable or
                window-local option.
                For the use of {expr}, see |bufname()| above.
                When the buffer or variable doesn't exist {def} or an empty
                string is returned, there is no error message.
                Examples: >
                        :let bufmodified = getbufvar(1, "&mod")
                        :echo "todo myvar = " . getbufvar("todo", "myvar")
<
getchar([expr])                                             *getchar()*
                Get a single character from the user or input stream.
                If [expr] is omitted, wait until a character is available.
                If [expr] is 0, only get a character when one is available.
                        Return zero otherwise.
                If [expr] is 1, only check if a character is available, it is
                        not consumed.  Return zero if no character available.

                Without [expr] and when [expr] is 0 a whole character or
                special key is returned.  If it is a single character, the
                result is a number.  Use nr2char() to convert it to a String.
                Otherwise a String is returned with the encoded character.
                For a special key it's a String with a sequence of bytes
                starting with 0x80 (decimal: 128).  This is the same value as
                the String "\<Key>", e.g., "\<Left>".  The returned value is
                also a String when a modifier (shift, control, alt) was used
                that is not included in the character.

                When [expr] is 0 and Esc is typed, there will be a short delay
                while Vim waits to see if this is the start of an escape
                sequence.

                When [expr] is 1 only the first byte is returned.  For a
                one-byte character it is the character itself as a number.
                Use nr2char() to convert it to a String.
```

```
                Use getcharmod() to obtain any additional modifiers.

                When the user clicks a mouse button, the mouse event will be
                returned.  The position can then be found in |v:mouse_col|,
                |v:mouse_lnum|, |v:mouse_winid| and |v:mouse_win|.  This
                example positions the mouse as it would normally happen: >
                        let c = getchar()
                        if c == "\<LeftMouse>" && v:mouse_win > 0
                          exe v:mouse_win . "wincmd w"
                          exe v:mouse_lnum
                          exe "normal " . v:mouse_col . "|"
                        endif
<
                When using bracketed paste only the first character is
                returned, the rest of the pasted text is dropped.
                |xterm-bracketed-paste|.

                There is no prompt, you will somehow have to make clear to the
                user that a character has to be typed.
                There is no mapping for the character.
                Key codes are replaced, thus when the user presses the <Del>
                key you get the code for the <Del> key, not the raw character
                sequence.  Examples: >
                        getchar() == "\<Del>"
                        getchar() == "\<S-Left>"
<               This example redefines "f" to ignore case: >
                        :nmap f :call FindChar()<CR>
                        :function FindChar()
                        :  let c = nr2char(getchar())
                        :  while col('.') < col('$') - 1
                        :    normal l
                        :    if getline('.')[col('.') - 1] ==? c
                        :      break
                        :    endif
                        :  endwhile
                        :endfunction
<
                You may also receive synthetic characters, such as
                |<CursorHold>|. Often you will want to ignore this and get
                another character: >
                        :function GetKey()
                        :  let c = getchar()
                        :  while c == "\<CursorHold>"
                        :    let c = getchar()
                        :  endwhile
                        :  return c
                        :endfunction

getcharmod()                                            *getcharmod()*
                The result is a Number which is the state of the modifiers for
                the last obtained character with getchar() or in another way.
                These values are added together:
                        2       shift
                        4       control
                        8       alt (meta)
                        16      meta (when it's different from ALT)
                        32      mouse double click
                        64      mouse triple click
                        96      mouse quadruple click (== 32 + 64)
                        128     command (Macintosh only)
                Only the modifiers that have not been included in the
```

```
                character itself are obtained.  Thus Shift-a results in "A"
                without a modifier.

getcharsearch()                                         *getcharsearch()*
                Return the current character search information as a {dict}
                with the following entries:

                    char        character previously used for a character
                                search (|t|, |f|, |T|, or |F|); empty string
                                if no character search has been performed
                    forward     direction of character search; 1 for forward,
                                0 for backward
                    until       type of character search; 1 for a |t| or |T|
                                character search, 0 for an |f| or |F|
                                character search

                This can be useful to always have |;| and |,| search
                forward/backward regardless of the direction of the previous
                character search: >
                        :nnoremap <expr> ; getcharsearch().forward ? ';' : ','
                        :nnoremap <expr> , getcharsearch().forward ? ',' : ';'
<               Also see |setcharsearch()|.

getcmdline()                                            *getcmdline()*
                Return the current command-line.  Only works when the command
                line is being edited, thus requires use of |c_CTRL-\_e| or
                |c_CTRL-R_=|.
                Example: >
                        :cmap <F7> <C-\>eescape(getcmdline(), ' \')<CR>
<               Also see |getcmdtype()|, |getcmdpos()| and |setcmdpos()|.

getcmdpos()                                             *getcmdpos()*
                Return the position of the cursor in the command line as a
                byte count.  The first column is 1.
                Only works when editing the command line, thus requires use of
                |c_CTRL-\_e| or |c_CTRL-R_=| or an expression mapping.
                Returns 0 otherwise.
                Also see |getcmdtype()|, |setcmdpos()| and |getcmdline()|.

getcmdtype()                                            *getcmdtype()*
                Return the current command-line type. Possible return values
                are:
                    :   normal Ex command
                    >   debug mode command |debug-mode|
                    /   forward search command
                    ?   backward search command
                    @   |input()| command
                    -   |:insert| or |:append| command
                    =   |i_CTRL-R_=|
                Only works when editing the command line, thus requires use of
                |c_CTRL-\_e| or |c_CTRL-R_=| or an expression mapping.
                Returns an empty string otherwise.
                Also see |getcmdpos()|, |setcmdpos()| and |getcmdline()|.

getcmdwintype()                                         *getcmdwintype()*
                Return the current |command-line-window| type. Possible return
                values are the same as |getcmdtype()|. Returns an empty string
                when not in the command-line window.

getcompletion({pat}, {type} [, {filtered}])             *getcompletion()*
                Return a list of command-line completion matches. {type}
                specifies what for.  The following completion types are
```

```
                supported:

                augroup         autocmd groups
                buffer          buffer names
                behave          :behave suboptions
                color           color schemes
                command         Ex command (and arguments)
                compiler        compilers
                cscope          |:cscope| suboptions
                dir             directory names
                environment     environment variable names
                event           autocommand events
                expression      Vim expression
                file            file and directory names
                file_in_path    file and directory names in |'path'|
                filetype        filetype names |'filetype'|
                function        function name
                help            help subjects
                highlight       highlight groups
                history         :history suboptions
                locale          locale names (as output of locale -a)
                mapclear        buffer argument
                mapping         mapping name
                menu            menus
                messages        |:messages| suboptions
                option          options
                packadd         optional package |pack-add| names
                shellcmd        Shell command
                sign            |:sign| suboptions
                syntax          syntax file names |'syntax'|
                syntime         |:syntime| suboptions
                tag             tags
                tag_listfiles   tags, file names
                user            user names
                var             user variables

                If {pat} is an empty string, then all the matches are returned.
                Otherwise only items matching {pat} are returned. See
                |wildcards| for the use of special characters in {pat}.

                If the optional {filtered} flag is set to 1, then 'wildignore'
                is applied to filter the results.  Otherwise all the matches
                are returned. The 'wildignorecase' option always applies.

                If there are no matches, an empty list is returned.  An
                invalid value for {type} produces an error.

                                                        *getcurpos()*
getcurpos()     Get the position of the cursor.  This is like getpos('.'), but
                includes an extra item in the list:
                    [bufnum, lnum, col, off, curswant] ~
                The "curswant" number is the preferred column when moving the
                cursor vertically.  Also see |getpos()|.

                This can be used to save and restore the cursor position: >
                        let save_cursor = getcurpos()
                        MoveTheCursorAround
                        call setpos('.', save_cursor)
<               Note that this only works within the window.  See
                |winrestview()| for restoring more state.
                                                        *getcwd()*
getcwd([{winnr} [, {tabnr}]])
```

```
                The result is a String, which is the name of the current
                working directory.
                Without arguments, for the current window.

                With {winnr} return the local current directory of this window
                in the current tab page.
                With {winnr} and {tabnr} return the local current directory of
                the window in the specified tab page.
                {winnr} can be the window number or the |window-ID|.
                Return an empty string if the arguments are invalid.

getfsize({fname})                                       *getfsize()*
                The result is a Number, which is the size in bytes of the
                given file {fname}.
                If {fname} is a directory, 0 is returned.
                If the file {fname} can't be found, -1 is returned.
                If the size of {fname} is too big to fit in a Number then -2
                is returned.

getfontname([{name}])                                   *getfontname()*
                Without an argument returns the name of the normal font being
                used.  Like what is used for the Normal highlight group
                |hl-Normal|.
                With an argument a check is done whether {name} is a valid
                font name.  If not then an empty string is returned.
                Otherwise the actual font name is returned, or {name} if the
                GUI does not support obtaining the real name.
                Only works when the GUI is running, thus not in your vimrc or
                gvimrc file.  Use the |GUIEnter| autocommand to use this
                function just after the GUI has started.
                Note that the GTK GUI accepts any font name, thus checking for
                a valid name does not work.

getfperm({fname})                                       *getfperm()*
                The result is a String, which is the read, write, and execute
                permissions of the given file {fname}.
                If {fname} does not exist or its directory cannot be read, an
                empty string is returned.
                The result is of the form "rwxrwxrwx", where each group of
                "rwx" flags represent, in turn, the permissions of the owner
                of the file, the group the file belongs to, and other users.
                If a user does not have a given permission the flag for this
                is replaced with the string "-".  Examples: >
                        :echo getfperm("/etc/passwd")
                        :echo getfperm(expand("~/.vimrc"))
<               This will hopefully (from a security point of view) display
                the string "rw-r--r--" or even "rw-------".

                For setting permissions use |setfperm()|.

getftime({fname})                                       *getftime()*
                The result is a Number, which is the last modification time of
                the given file {fname}.  The value is measured as seconds
                since 1st Jan 1970, and may be passed to strftime().  See also
                |localtime()| and |strftime()|.
                If the file {fname} can't be found -1 is returned.

getftype({fname})                                       *getftype()*
                The result is a String, which is a description of the kind of
                file of the given file {fname}.
                If {fname} does not exist an empty string is returned.
                Here is a table over different kinds of files and their
```

```
                results:
                        Normal file             "file"
                        Directory               "dir"
                        Symbolic link           "link"
                        Block device            "bdev"
                        Character device        "cdev"
                        Socket                  "socket"
                        FIFO                    "fifo"
                        All other               "other"
                Example: >
                        getftype("/home")
<               Note that a type such as "link" will only be returned on
                systems that support it.  On some systems only "dir" and
                "file" are returned.  On MS-Windows a symbolic link to a
                directory returns "dir" instead of "link".

                                                        *getline()*
getline({lnum} [, {end}])
                Without {end} the result is a String, which is line {lnum}
                from the current buffer.  Example: >
                        getline(1)
<               When {lnum} is a String that doesn't start with a
                digit, line() is called to translate the String into a Number.
                To get the line under the cursor: >
                        getline(".")
<               When {lnum} is smaller than 1 or bigger than the number of
                lines in the buffer, an empty string is returned.

                When {end} is given the result is a |List| where each item is
                a line from the current buffer in the range {lnum} to {end},
                including line {end}.
                {end} is used in the same way as {lnum}.
                Non-existing lines are silently omitted.
                When {end} is before {lnum} an empty |List| is returned.
                Example: >
                        :let start = line('.')
                        :let end = search("^$") - 1
                        :let lines = getline(start, end)

<               To get lines from another buffer see |getbufline()|

getloclist({nr}[, {what}])                              *getloclist()*
                Returns a list with all the entries in the location list for
                window {nr}.  {nr} can be the window number or the |window-ID|.
                When {nr} is zero the current window is used.

                For a location list window, the displayed location list is
                returned.  For an invalid window number {nr}, an empty list is
                returned. Otherwise, same as |getqflist()|.

                If the optional {what} dictionary argument is supplied, then
                returns the items listed in {what} as a dictionary. Refer to
                |getqflist()| for the supported items in {what}.

getmatches()                                            *getmatches()*
                Returns a |List| with all matches previously defined by
                |matchadd()| and the |:match| commands.  |getmatches()| is
                useful in combination with |setmatches()|, as |setmatches()|
                can restore a list of matches saved by |getmatches()|.
                Example: >
                        :echo getmatches()
<                       [{'group': 'MyGroup1', 'pattern': 'TODO',
```

```
                        'priority': 10, 'id': 1}, {'group': 'MyGroup2',
                        'pattern': 'FIXME', 'priority': 10, 'id': 2}] >
                        :let m = getmatches()
                        :call clearmatches()
                        :echo getmatches()
<                       [] >
                        :call setmatches(m)
                        :echo getmatches()
<                       [{'group': 'MyGroup1', 'pattern': 'TODO',
                        'priority': 10, 'id': 1}, {'group': 'MyGroup2',
                        'pattern': 'FIXME', 'priority': 10, 'id': 2}] >
                        :unlet m
<
                                                        *getpid()*
getpid()        Return a Number which is the process ID of the Vim process.
                On Unix and MS-Windows this is a unique number, until Vim
                exits.  On MS-DOS it's always zero.

                                                        *getpos()*
getpos({expr})  Get the position for {expr}.  For possible values of {expr}
                see |line()|.  For getting the cursor position see
                |getcurpos()|.
                The result is a |List| with four numbers:
                    [bufnum, lnum, col, off]
                "bufnum" is zero, unless a mark like '0 or 'A is used, then it
                is the buffer number of the mark.
                "lnum" and "col" are the position in the buffer.  The first
                column is 1.
                The "off" number is zero, unless 'virtualedit' is used.  Then
                it is the offset in screen columns from the start of the
                character.  E.g., a position within a <Tab> or after the last
                character.
                Note that for '< and '> Visual mode matters: when it is "V"
                (visual line mode) the column of '< is zero and the column of
                '> is a large number.
                This can be used to save and restore the position of a mark: >
                        let save_a_mark = getpos("'a")
                        ...
                        call setpos("'a", save_a_mark)
<               Also see |getcurpos()| and |setpos()|.


getqflist([{what}])                                     *getqflist()*
                Returns a list with all the current quickfix errors.  Each
                list item is a dictionary with these entries:
                        bufnr   number of buffer that has the file name, use
                                bufname() to get the name
                        lnum    line number in the buffer (first line is 1)
                        col     column number (first column is 1)
                        vcol    |TRUE|: "col" is visual column
                                |FALSE|: "col" is byte index
                        nr      error number
                        pattern search pattern used to locate the error
                        text    description of the error
                        type    type of the error, 'E', '1', etc.
                        valid   |TRUE|: recognized error message

                When there is no error list or it's empty, an empty list is
                returned. Quickfix list entries with non-existing buffer
                number are returned with "bufnr" set to zero.

                Useful application: Find pattern matches in multiple files and
```

```
                do something with them: >
                        :vimgrep /theword/jg *.c
                        :for d in getqflist()
                        :   echo bufname(d.bufnr) ':' d.lnum '=' d.text
                        :endfor
<
                If the optional {what} dictionary argument is supplied, then
                returns only the items listed in {what} as a dictionary. The
                following string items are supported in {what}:
                        context get the context stored with |setqflist()|
                        efm     errorformat to use when parsing "lines". If
                                not present, then the 'erroformat' option
                                value is used.
                        id      get information for the quickfix list with
                                |quickfix-ID|; zero means the id for the
                                current list or the list specifed by "nr"
                        idx     index of the current entry in the list
                        items   quickfix list entries
                        lines   use 'errorformat' to extract items from a list
                                of lines and return the resulting entries.
                                Only a |List| type is accepted.  The current
                                quickfix list is not modified.
                        nr      get information for this quickfix list; zero
                                means the current quickfix list and "$" means
                                the last quickfix list
                        size    number of entries in the quickfix list
                        title   get the list title
                        winid   get the |window-ID| (if opened)
                        all     all of the above quickfix properties
                Non-string items in {what} are ignored.
                If "nr" is not present then the current quickfix list is used.
                If both "nr" and a non-zero "id" are specified, then the list
                specified by "id" is used.
                To get the number of lists in the quickfix stack, set "nr" to
                "$" in {what}. The "nr" value in the returned dictionary
                contains the quickfix stack size.
                When "lines" is specified, all the other items except "efm"
                are ignored.  The returned dictionary contains the entry
                "items" with the list of entries.
                In case of error processing {what}, an empty dictionary is
                returned.

                The returned dictionary contains the following entries:
                        context context information stored with |setqflist()|
                        id      quickfix list ID |quickfix-ID|
                        idx     index of the current entry in the list
                        items   quickfix list entries
                        nr      quickfix list number
                        size    number of entries in the quickfix list
                        title   quickfix list title text
                        winid   quickfix |window-ID| (if opened)

                Examples: >
                        :echo getqflist({'all': 1})
                        :echo getqflist({'nr': 2, 'title': 1})
                        :echo getqflist({'lines' : ["F1:10:L10"]})
<

getreg([{regname} [, 1 [, {list}]]])                    *getreg()*
                The result is a String, which is the contents of register
                {regname}.  Example: >
                        :let cliptext = getreg('*')
```

```
<               When {regname} was not set the result is an empty string.

                getreg('=') returns the last evaluated value of the expression
                register.  (For use in maps.)
                getreg('=', 1) returns the expression itself, so that it can
                be restored with |setreg()|.  For other registers the extra
                argument is ignored, thus you can always give it.

                If {list} is present and |TRUE|, the result type is changed
                to |List|. Each list item is one text line. Use it if you care
                about zero bytes possibly present inside register: without
                third argument both NLs and zero bytes are represented as NLs
                (see |NL-used-for-Nul|).
                When the register was not set an empty list is returned.

                If {regname} is not specified, |v:register| is used.


getregtype([{regname}])                                 *getregtype()*
                The result is a String, which is type of register {regname}.
                The value will be one of:
                    "v"                 for |characterwise| text
                    "V"                 for |linewise| text
                    "<CTRL-V>{width}"   for |blockwise-visual| text
                    ""                  for an empty or unknown register
                <CTRL-V> is one character with value 0x16.
                If {regname} is not specified, |v:register| is used.

gettabinfo([{arg}])                                     *gettabinfo()*
                If {arg} is not specified, then information about all the tab
                pages is returned as a List. Each List item is a Dictionary.
                Otherwise, {arg} specifies the tab page number and information
                about that one is returned.  If the tab page does not exist an
                empty List is returned.

                Each List item is a Dictionary with the following entries:
                        tabnr           tab page number.
                        variables       a reference to the dictionary with
                                        tabpage-local variables
                        windows         List of |window-ID|s in the tag page.

gettabvar({tabnr}, {varname} [, {def}])                         *gettabvar()*
                Get the value of a tab-local variable {varname} in tab page
                {tabnr}. |t:var|
                Tabs are numbered starting with one.
                When {varname} is empty a dictionary with all tab-local
                variables is returned.
                Note that the name without "t:" must be used.
                When the tab or variable doesn't exist {def} or an empty
                string is returned, there is no error message.

gettabwinvar({tabnr}, {winnr}, {varname} [, {def}])             *gettabwinvar()*
                Get the value of window-local variable {varname} in window
                {winnr} in tab page {tabnr}.
                When {varname} is empty a dictionary with all window-local
                variables is returned.
                When {varname} is equal to "&" get the values of all
                window-local options in a Dictionary.
                Otherwise, when {varname} starts with "&" get the value of a
                window-local option.
                Note that {varname} must be the name without "w:".
                Tabs are numbered starting with one.  For the current tabpage
```

```
                use |getwinvar()|.
                {winnr} can be the window number or the |window-ID|.
                When {winnr} is zero the current window is used.
                This also works for a global option, buffer-local option and
                window-local option, but it doesn't work for a global variable
                or buffer-local variable.
                When the tab, window or variable doesn't exist {def} or an
                empty string is returned, there is no error message.
                Examples: >
                        :let list_is_on = gettabwinvar(1, 2, '&list')
                        :echo "myvar = " . gettabwinvar(3, 1, 'myvar')
<
                                                        *getwinposx()*
getwinposx()    The result is a Number, which is the X coordinate in pixels of
                the left hand side of the GUI Vim window. Also works for an
                xterm.
                The result will be -1 if the information is not available.
                The value can be used with `:winpos`.

                                                        *getwinposy()*
getwinposy()    The result is a Number, which is the Y coordinate in pixels of
                the top of the GUI Vim window.  Also works for an xterm.
                The result will be -1 if the information is not available.
                The value can be used with `:winpos`.

getwininfo([{winid}])                                   *getwininfo()*
                Returns information about windows as a List with Dictionaries.

                If {winid} is given Information about the window with that ID
                is returned.  If the window does not exist the result is an
                empty list.

                Without {winid} information about all the windows in all the
                tab pages is returned.

                Each List item is a Dictionary with the following entries:
                        bufnr           number of buffer in the window
                        height          window height (excluding winbar)
                        winbar          1 if the window has a toolbar, 0
                                        otherwise
                        loclist         1 if showing a location list
                                        {only with the +quickfix feature}
                        quickfix        1 if quickfix or location list window
                                        {only with the +quickfix feature}
                        terminal        1 if a terminal window
                                        {only with the +terminal feature}
                        tabnr           tab page number
                        variables       a reference to the dictionary with
                                        window-local variables
                        width           window width
                        winid           |window-ID|
                        winnr           window number

                To obtain all window-local variables use: >
                        gettabwinvar({tabnr}, {winnr}, '&')

getwinvar({winnr}, {varname} [, {def}])                         *getwinvar()*
                Like |gettabwinvar()| for the current tabpage.
                Examples: >
                        :let list_is_on = getwinvar(2, '&list')
                        :echo "myvar = " . getwinvar(1, 'myvar')
<
```

```
glob({expr} [, {nosuf} [, {list} [, {alllinks}]]])              *glob()*
                Expand the file wildcards in {expr}.  See |wildcards| for the
                use of special characters.

                Unless the optional {nosuf} argument is given and is |TRUE|,
                the 'suffixes' and 'wildignore' options apply: Names matching
                one of the patterns in 'wildignore' will be skipped and
                'suffixes' affect the ordering of matches.
                'wildignorecase' always applies.

                When {list} is present and it is |TRUE| the result is a List
                with all matching files. The advantage of using a List is,
                you also get filenames containing newlines correctly.
                Otherwise the result is a String and when there are several
                matches, they are separated by <NL> characters.

                If the expansion fails, the result is an empty String or List.

                A name for a non-existing file is not included.  A symbolic
                link is only included if it points to an existing file.
                However, when the {alllinks} argument is present and it is
                |TRUE| then all symbolic links are included.

                For most systems backticks can be used to get files names from
                any external command.  Example: >
                        :let tagfiles = glob("`find . -name tags -print`")
                        :let &tags = substitute(tagfiles, "\n", ",", "g")
<               The result of the program inside the backticks should be one
                item per line.  Spaces inside an item are allowed.

                See |expand()| for expanding special Vim variables.  See
                |system()| for getting the raw output of an external command.

glob2regpat({expr})                                        *glob2regpat()*
                Convert a file pattern, as used by glob(), into a search
                pattern.  The result can be used to match with a string that
                is a file name.  E.g. >
                        if filename =~ glob2regpat('Make*.mak')
<               This is equivalent to: >
                        if filename =~ '^Make.*\.mak$'
<               When {expr} is an empty string the result is "^$", match an
                empty string.
                Note that the result depends on the system.  On MS-Windows
                a backslash usually means a path separator.

                                                                *globpath()*
globpath({path}, {expr} [, {nosuf} [, {list} [, {alllinks}]]])
                Perform glob() on all directories in {path} and concatenate
                the results.  Example: >
                        :echo globpath(&rtp, "syntax/c.vim")
<
                {path} is a comma-separated list of directory names.  Each
                directory name is prepended to {expr} and expanded like with
                |glob()|.  A path separator is inserted when needed.
                To add a comma inside a directory name escape it with a
                backslash.  Note that on MS-Windows a directory may have a
                trailing backslash, remove it if you put a comma after it.
                If the expansion fails for one of the directories, there is no
                error message.

                Unless the optional {nosuf} argument is given and is |TRUE|,
                the 'suffixes' and 'wildignore' options apply: Names matching
```

```
                one of the patterns in 'wildignore' will be skipped and
                'suffixes' affect the ordering of matches.

                When {list} is present and it is |TRUE| the result is a List
                with all matching files. The advantage of using a List is, you
                also get filenames containing newlines correctly. Otherwise
                the result is a String and when there are several matches,
                they are separated by <NL> characters.  Example: >
                        :echo globpath(&rtp, "syntax/c.vim", 0, 1)
<
                {alllinks} is used as with |glob()|.

                The "**" item can be used to search in a directory tree.
                For example, to find all "README.txt" files in the directories
                in 'runtimepath' and below: >
                        :echo globpath(&rtp, "**/README.txt")
<               Upwards search and limiting the depth of "**" is not
                supported, thus using 'path' will not always work properly.

                                                        *has()*
has({feature})  The result is a Number, which is 1 if the feature {feature} is
                supported, zero otherwise.  The {feature} argument is a
                string.  See |feature-list| below.
                Also see |exists()|.


has_key({dict}, {key})                                  *has_key()*
                The result is a Number, which is 1 if |Dictionary| {dict} has
                an entry with key {key}.  Zero otherwise.

haslocaldir([{winnr} [, {tabnr}]])                      *haslocaldir()*
                The result is a Number, which is 1 when the window has set a
                local path via |:lcd|, and 0 otherwise.

                Without arguments use the current window.
                With {winnr} use this window in the current tab page.
                With {winnr} and {tabnr} use the window in the specified tab
                page.
                {winnr} can be the window number or the |window-ID|.
                Return 0 if the arguments are invalid.

hasmapto({what} [, {mode} [, {abbr}]])                  *hasmapto()*
                The result is a Number, which is 1 if there is a mapping that
                contains {what} in somewhere in the rhs (what it is mapped to)
                and this mapping exists in one of the modes indicated by
                {mode}.
                When {abbr} is there and it is |TRUE| use abbreviations
                instead of mappings.  Don't forget to specify Insert and/or
                Command-line mode.
                Both the global mappings and the mappings local to the current
                buffer are checked for a match.
                If no matching mapping is found 0 is returned.
                The following characters are recognized in {mode}:
                        n       Normal mode
                        v       Visual mode
                        o       Operator-pending mode
                        i       Insert mode
                        l       Language-Argument ("r", "f", "t", etc.)
                        c       Command-line mode
                When {mode} is omitted, "nvo" is used.

                This function is useful to check if a mapping already exists
```

```
                to a function in a Vim script.  Example: >
                        :if !hasmapto('\ABCdoit')
                        :   map <Leader>d \ABCdoit
                        :endif
<               This installs the mapping to "\ABCdoit" only if there isn't
                already a mapping to "\ABCdoit".

histadd({history}, {item})                              *histadd()*
                Add the String {item} to the history {history} which can be
                one of:                                 *hist-names*
                        "cmd"    or ":"   command line history
                        "search" or "/"   search pattern history
                        "expr"   or "="   typed expression history
                        "input"  or "@"   input line history
                        "debug"  or ">"   debug command history
                        empty             the current or last used history
                The {history} string does not need to be the whole name, one
                character is sufficient.
                If {item} does already exist in the history, it will be
                shifted to become the newest entry.
                The result is a Number: 1 if the operation was successful,
                otherwise 0 is returned.

                Example: >
                        :call histadd("input", strftime("%Y %b %d"))
                        :let date=input("Enter date: ")
<               This function is not available in the |sandbox|.

histdel({history} [, {item}])                           *histdel()*
                Clear {history}, i.e. delete all its entries.  See |hist-names|
                for the possible values of {history}.

                If the parameter {item} evaluates to a String, it is used as a
                regular expression.  All entries matching that expression will
                be removed from the history (if there are any).
                Upper/lowercase must match, unless "\c" is used |/\c|.
                If {item} evaluates to a Number, it will be interpreted as
                an index, see |:history-indexing|.  The respective entry will
                be removed if it exists.

                The result is a Number: 1 for a successful operation,
                otherwise 0 is returned.

                Examples:
                Clear expression register history: >
                        :call histdel("expr")
<
                Remove all entries starting with "*" from the search history: >
                        :call histdel("/", '^\*')
<
                The following three are equivalent: >
                        :call histdel("search", histnr("search"))
                        :call histdel("search", -1)
                        :call histdel("search", '^'.histget("search", -1).'$')
<
                To delete the last search pattern and use the last-but-one for
                the "n" command and 'hlsearch': >
                        :call histdel("search", -1)
                        :let @/ = histget("search", -1)

histget({history} [, {index}])                          *histget()*
                The result is a String, the entry with Number {index} from
```

```
                {history}.  See |hist-names| for the possible values of
                {history}, and |:history-indexing| for {index}.  If there is
                no such entry, an empty String is returned.  When {index} is
                omitted, the most recent item from the history is used.

                Examples:
                Redo the second last search from history. >
                        :execute '/' . histget("search", -2)

<               Define an Ex command ":H {num}" that supports re-execution of
                the {num}th entry from the output of |:history|. >
                        :command -nargs=1 H execute histget("cmd", 0+<args>)
<
histnr({history})                                       *histnr()*
                The result is the Number of the current entry in {history}.
                See |hist-names| for the possible values of {history}.
                If an error occurred, -1 is returned.

                Example: >
                        :let inp_index = histnr("expr")
<
hlexists({name})                                        *hlexists()*
                The result is a Number, which is non-zero if a highlight group
                called {name} exists.  This is when the group has been
                defined in some way.  Not necessarily when highlighting has
                been defined for it, it may also have been used for a syntax
                item.
                                                        *highlight_exists()*
                Obsolete name: highlight_exists().

                                                        *hlID()*
hlID({name})    The result is a Number, which is the ID of the highlight group
                with name {name}.  When the highlight group doesn't exist,
                zero is returned.
                This can be used to retrieve information about the highlight
                group.  For example, to get the background color of the
                "Comment" group: >
        :echo synIDattr(synIDtrans(hlID("Comment")), "bg")
<                                                       *highlightID()*
                Obsolete name: highlightID().

hostname()                                              *hostname()*
                The result is a String, which is the name of the machine on
                which Vim is currently running.  Machine names greater than
                256 characters long are truncated.

iconv({expr}, {from}, {to})                             *iconv()*
                The result is a String, which is the text {expr} converted
                from encoding {from} to encoding {to}.
                When the conversion completely fails an empty string is
                returned.  When some characters could not be converted they
                are replaced with "?".
                The encoding names are whatever the iconv() library function
                can accept, see ":!man 3 iconv".
                Most conversions require Vim to be compiled with the |+iconv|
                feature.  Otherwise only UTF-8 to latin1 conversion and back
                can be done.
                This can be used to display messages with special characters,
                no matter what 'encoding' is set to.  Write the message in
                UTF-8 and use: >
                        echo iconv(utf8_str, "utf-8", &enc)
<               Note that Vim uses UTF-8 for all Unicode encodings, conversion
```

```
                from/to UCS-2 is automatically changed to use UTF-8.  You
                cannot use UCS-2 in a string anyway, because of the NUL bytes.
                {only available when compiled with the |+multi_byte| feature}

                                                        *indent()*
indent({lnum})  The result is a Number, which is indent of line {lnum} in the
                current buffer.  The indent is counted in spaces, the value
                of 'tabstop' is relevant.  {lnum} is used just like in
                |getline()|.
                When {lnum} is invalid -1 is returned.


index({list}, {expr} [, {start} [, {ic}]])                     *index()*
                Return the lowest index in |List| {list} where the item has a
                value equal to {expr}.  There is no automatic conversion, so
                the String "4" is different from the Number 4.  And the number
                4 is different from the Float 4.0.  The value of 'ignorecase'
                is not used here, case always matters.
                If {start} is given then start looking at the item with index
                {start} (may be negative for an item relative to the end).
                When {ic} is given and it is |TRUE|, ignore case.  Otherwise
                case must match.
                -1 is returned when {expr} is not found in {list}.
                Example: >
                        :let idx = index(words, "the")
                        :if index(numbers, 123) >= 0


input({prompt} [, {text} [, {completion}]])             *input()*
                The result is a String, which is whatever the user typed on
                the command-line.  The {prompt} argument is either a prompt
                string, or a blank string (for no prompt).  A '\n' can be used
                in the prompt to start a new line.
                The highlighting set with |:echohl| is used for the prompt.
                The input is entered just like a command-line, with the same
                editing commands and mappings.  There is a separate history
                for lines typed for input().
                Example: >
                        :if input("Coffee or beer? ") == "beer"
                        :  echo "Cheers!"
                        :endif
<
                If the optional {text} argument is present and not empty, this
                is used for the default reply, as if the user typed this.
                Example: >
                        :let color = input("Color? ", "white")

<               The optional {completion} argument specifies the type of
                completion supported for the input.  Without it completion is
                not performed.  The supported completion types are the same as
                that can be supplied to a user-defined command using the
                "-complete=" argument.  Refer to |:command-completion| for
                more information.  Example: >
                        let fname = input("File: ", "", "file")
<
                NOTE: This function must not be used in a startup file, for
                the versions that only run in GUI mode (e.g., the Win32 GUI).
                Note: When input() is called from within a mapping it will
                consume remaining characters from that mapping, because a
                mapping is handled like the characters were typed.
                Use |inputsave()| before input() and |inputrestore()|
                after input() to avoid that.  Another solution is to avoid
```

```
                that further characters follow in the mapping, e.g., by using
                |:execute| or |:normal|.

                Example with a mapping: >
                        :nmap \x :call GetFoo()<CR>:exe "/" . Foo<CR>
                        :function GetFoo()
                        :  call inputsave()
                        :  let g:Foo = input("enter search pattern: ")
                        :  call inputrestore()
                        :endfunction

inputdialog({prompt} [, {text} [, {cancelreturn}]])             *inputdialog()*
                Like |input()|, but when the GUI is running and text dialogs
                are supported, a dialog window pops up to input the text.
                Example: >
                   :let n = inputdialog("value for shiftwidth", shiftwidth())
                   :if n != ""
                   :  let &sw = n
                   :endif
<               When the dialog is cancelled {cancelreturn} is returned.  When
                omitted an empty string is returned.
                Hitting <Enter> works like pressing the OK button.  Hitting
                <Esc> works like pressing the Cancel button.
                NOTE: Command-line completion is not supported.

inputlist({textlist})                                       *inputlist()*
                {textlist} must be a |List| of strings.  This |List| is
                displayed, one string per line.  The user will be prompted to
                enter a number, which is returned.
                The user can also select an item by clicking on it with the
                mouse.  For the first string 0 is returned.  When clicking
                above the first item a negative number is returned.  When
                clicking on the prompt one more than the length of {textlist}
                is returned.
                Make sure {textlist} has less than 'lines' entries, otherwise
                it won't work.  It's a good idea to put the entry number at
                the start of the string.  And put a prompt in the first item.
                Example: >
                        let color = inputlist(['Select color:', '1. red',
                                \ '2. green', '3. blue'])

inputrestore()                                              *inputrestore()*
                Restore typeahead that was saved with a previous |inputsave()|.
                Should be called the same number of times inputsave() is
                called.  Calling it more often is harmless though.
                Returns 1 when there is nothing to restore, 0 otherwise.

inputsave()                                                 *inputsave()*
                Preserve typeahead (also from mappings) and clear it, so that
                a following prompt gets input from the user.  Should be
                followed by a matching inputrestore() after the prompt.  Can
                be used several times, in which case there must be just as
                many inputrestore() calls.
                Returns 1 when out of memory, 0 otherwise.

inputsecret({prompt} [, {text}])                            *inputsecret()*
                This function acts much like the |input()| function with but
                two exceptions:
                a) the user's response will be displayed as a sequence of
                asterisks ("*") thereby keeping the entry secret, and
                b) the user's response will not be recorded on the input
                |history| stack.
```

```
                The result is a String, which is whatever the user actually
                typed on the command-line in response to the issued prompt.
                NOTE: Command-line completion is not supported.

insert({list}, {item} [, {idx}])                        *insert()*
                Insert {item} at the start of |List| {list}.
                If {idx} is specified insert {item} before the item with index
                {idx}.  If {idx} is zero it goes before the first item, just
                like omitting {idx}.  A negative {idx} is also possible, see
                |list-index|.  -1 inserts just before the last item.
                Returns the resulting |List|.  Examples: >
                        :let mylist = insert([2, 3, 5], 1)
                        :call insert(mylist, 4, -1)
                        :call insert(mylist, 6, len(mylist))
<               The last example can be done simpler with |add()|.
                Note that when {item} is a |List| it is inserted as a single
                item.  Use |extend()| to concatenate |Lists|.

invert({expr})                                          *invert()*
                Bitwise invert.  The argument is converted to a number.  A
                List, Dict or Float argument causes an error.  Example: >
                        :let bits = invert(bits)

isdirectory({directory})                                *isdirectory()*
                The result is a Number, which is |TRUE| when a directory
                with the name {directory} exists.  If {directory} doesn't
                exist, or isn't a directory, the result is |FALSE|.  {directory}
                is any expression, which is used as a String.

islocked({expr})                                        *islocked()* *E786*
                The result is a Number, which is |TRUE| when {expr} is the
                name of a locked variable.
                {expr} must be the name of a variable, |List| item or
                |Dictionary| entry, not the variable itself!  Example: >
                        :let alist = [0, ['a', 'b'], 2, 3]
                        :lockvar 1 alist
                        :echo islocked('alist')         " 1
                        :echo islocked('alist[1]')      " 0

<               When {expr} is a variable that does not exist you get an error
                message.  Use |exists()| to check for existence.

isnan({expr})                                           *isnan()*
                Return |TRUE| if {expr} is a float with value NaN. >
                        echo isnan(0.0 / 0.0)
<                       1 ~

                {only available when compiled with the |+float| feature}

items({dict})                                           *items()*
                Return a |List| with all the key-value pairs of {dict}.  Each
                |List| item is a list with two items: the key of a {dict}
                entry and the value of this entry.  The |List| is in arbitrary
                order.

job_getchannel({job})                                    *job_getchannel()*
                Get the channel handle that {job} is using.
                To check if the job has no channel: >
                        if string(job_getchannel()) == 'channel fail'
<
                {only available when compiled with the |+job| feature}
```

```
job_info({job})                                          *job_info()*
                Returns a Dictionary with information about {job}:
                   "status"     what |job_status()| returns
                   "channel"    what |job_getchannel()| returns
                   "process"    process ID
                   "tty_in"     terminal input name, empty when none
                   "tty_out"    terminal output name, empty when none
                   "exitval"    only valid when "status" is "dead"
                   "exit_cb"    function to be called on exit
                   "stoponexit" |job-stoponexit|

job_setoptions({job}, {options})                         *job_setoptions()*
                Change options for {job}.  Supported are:
                   "stoponexit" |job-stoponexit|
                   "exit_cb"    |job-exit_cb|

job_start({command} [, {options}])                       *job_start()*
                Start a job and return a Job object.  Unlike |system()| and
                |:!cmd| this does not wait for the job to finish.
                To start a job in a terminal window see |term_start()|.

                {command} can be a String.  This works best on MS-Windows.  On
                Unix it is split up in white-separated parts to be passed to
                execvp().  Arguments in double quotes can contain white space.

                {command} can be a List, where the first item is the executable
                and further items are the arguments.  All items are converted
                to String.  This works best on Unix.

                On MS-Windows, job_start() makes a GUI application hidden. If
                want to show it, Use |:!start| instead.

                The command is executed directly, not through a shell, the
                'shell' option is not used.  To use the shell: >
        let job = job_start(["/bin/sh", "-c", "echo hello"])
<               Or: >
        let job = job_start('/bin/sh -c "echo hello"')
<               Note that this will start two processes, the shell and the
                command it executes.  If you don't want this use the "exec"
                shell command.

                On Unix $PATH is used to search for the executable only when
                the command does not contain a slash.

                The job will use the same terminal as Vim.  If it reads from
                stdin the job and Vim will be fighting over input, that
                doesn't work.  Redirect stdin and stdout to avoid problems: >
        let job = job_start(['sh', '-c', "myserver </dev/null >/dev/null"])
<
                The returned Job object can be used to get the status with
                |job_status()| and stop the job with |job_stop()|.

                {options} must be a Dictionary.  It can contain many optional
                items, see |job-options|.

                {only available when compiled with the |+job| feature}

job_status({job})                                        *job_status()* *E916*
                Returns a String with the status of {job}:
                        "run"   job is running
                        "fail"  job failed to start
                        "dead"  job died or was stopped after running
```

On Unix a non-existing command results in "dead" instead of
"fail", because a fork happens before the failure can be
detected.

If an exit callback was set with the "exit_cb" option and the
job is now detected to be "dead" the callback will be invoked.

For more information see |job_info()|.

{only available when compiled with the |+job| feature}

job_stop({job} [, {how}]) *job_stop()*
Stop the {job}. This can also be used to signal the job.

When {how} is omitted or is "term" the job will be terminated.
For Unix SIGTERM is sent. On MS-Windows the job will be
terminated forcedly (there is no "gentle" way).
This goes to the process group, thus children may also be
affected.

Effect for Unix:
"term" SIGTERM (default)
"hup" SIGHUP
"quit" SIGQUIT
"int" SIGINT
"kill" SIGKILL (strongest way to stop)
number signal with that number

Effect for MS-Windows:
"term" terminate process forcedly (default)
"hup" CTRL_BREAK
"quit" CTRL_BREAK
"int" CTRL_C
"kill" terminate process forcedly
Others CTRL_BREAK

On Unix the signal is sent to the process group. This means
that when the job is "sh -c command" it affects both the shell
and the command.

The result is a Number: 1 if the operation could be executed,
0 if "how" is not supported on the system.
Note that even when the operation was executed, whether the
job was actually stopped needs to be checked with
|job_status()|.

If the status of the job is "dead", the signal will not be
sent. This is to avoid to stop the wrong job (esp. on Unix,
where process numbers are recycled).

When using "kill" Vim will assume the job will die and close
the channel.

{only available when compiled with the |+job| feature}

join({list} [, {sep}]) *join()*
Join the items in {list} together into one String.
When {sep} is specified it is put in between the items. If
{sep} is omitted a single space is used.
Note that {sep} is not added at the end. You might want to
add it there too: >

```
                        let lines = join(mylist, "\n") . "\n"
<               String items are used as-is.  |Lists| and |Dictionaries| are
                converted into a string like with |string()|.
                The opposite function is |split()|.

js_decode({string})                                     *js_decode()*
                This is similar to |json_decode()| with these differences:
                - Object key names do not have to be in quotes.
                - Strings can be in single quotes.
                - Empty items in an array (between two commas) are allowed and
                  result in v:none items.

js_encode({expr})                                       *js_encode()*
                This is similar to |json_encode()| with these differences:
                - Object key names are not in quotes.
                - v:none items in an array result in an empty item between
                  commas.
                For example, the Vim object:
                        [1,v:none,{"one":1},v:none] ~
                Will be encoded as:
                        [1,,{one:1},,] ~
                While json_encode() would produce:
                        [1,null,{"one":1},null] ~
                This encoding is valid for JavaScript. It is more efficient
                than JSON, especially when using an array with optional items.


json_decode({string})                                   *json_decode()*
                This parses a JSON formatted string and returns the equivalent
                in Vim values.  See |json_encode()| for the relation between
                JSON and Vim values.
                The decoding is permissive:
                - A trailing comma in an array and object is ignored, e.g.
                  "[1, 2, ]" is the same as "[1, 2]".
                - More floating point numbers are recognized, e.g. "1." for
                  "1.0", or "001.2" for "1.2". Special floating point values
                  "Infinity" and "NaN" (capitalization ignored) are accepted.
                - Leading zeroes in integer numbers are ignored, e.g. "012"
                  for "12" or "-012" for "-12".
                - Capitalization is ignored in literal names null, true or
                  false, e.g. "NULL" for "null", "True" for "true".
                - Control characters U+0000 through U+001F which are not
                  escaped in strings are accepted, e.g. "       " (tab
                  character in string) for "\t".
                - Backslash in an invalid 2-character sequence escape is
                  ignored, e.g. "\a" is decoded as "a".
                - A correct surrogate pair in JSON strings should normally be
                  a 12 character sequence such as "\uD834\uDD1E", but
                  json_decode() silently accepts truncated surrogate pairs
                  such as "\uD834" or "\uD834\u"
                                                        *E938*
                A duplicate key in an object, valid in rfc7159, is not
                accepted by json_decode() as the result must be a valid Vim
                type, e.g. this fails: {"a":"b", "a":"c"}


json_encode({expr})                                     *json_encode()*
                Encode {expr} as JSON and return this as a string.
                The encoding is specified in:
                https://tools.ietf.org/html/rfc7159.html
                Vim values are converted as follows:
                   Number               decimal number
```

```
                   Float               floating point number
                   Float nan           "NaN"
                   Float inf           "Infinity"
                   String              in double quotes (possibly null)
                   Funcref             not possible, error
                   List                as an array (possibly null); when
                                       used recursively: []
                   Dict                as an object (possibly null); when
                                       used recursively: {}
                   v:false             "false"
                   v:true              "true"
                   v:none              "null"
                   v:null              "null"
               Note that NaN and Infinity are passed on as values.  This is
               missing in the JSON standard, but several implementations do
               allow it.  If not then you will get an error.

keys({dict})                                                    *keys()*
               Return a |List| with all the keys of {dict}.  The |List| is in
               arbitrary order.

                                                                *len()* *E701*
len({expr})    The result is a Number, which is the length of the argument.
               When {expr} is a String or a Number the length in bytes is
               used, as with |strlen()|.
               When {expr} is a |List| the number of items in the |List| is
               returned.
               When {expr} is a |Dictionary| the number of entries in the
               |Dictionary| is returned.
               Otherwise an error is given.

                                                *libcall()* *E364* *E368*
libcall({libname}, {funcname}, {argument})
               Call function {funcname} in the run-time library {libname}
               with single argument {argument}.
               This is useful to call functions in a library that you
               especially made to be used with Vim.  Since only one argument
               is possible, calling standard library functions is rather
               limited.
               The result is the String returned by the function.  If the
               function returns NULL, this will appear as an empty string ""
               to Vim.
               If the function returns a number, use libcallnr()!
               If {argument} is a number, it is passed to the function as an
               int; if {argument} is a string, it is passed as a
               null-terminated string.
               This function will fail in |restricted-mode|.

               libcall() allows you to write your own 'plug-in' extensions to
               Vim without having to recompile the program.  It is NOT a
               means to call system functions!  If you try to do so Vim will
               very probably crash.

               For Win32, the functions you write must be placed in a DLL
               and use the normal C calling convention (NOT Pascal which is
               used in Windows System DLLs).  The function must take exactly
               one parameter, either a character pointer or a long integer,
               and must return a character pointer or NULL.  The character
               pointer returned must point to memory that will remain valid
               after the function has returned (e.g. in static data in the
               DLL).  If it points to allocated memory, that memory will
               leak away.  Using a static buffer in the function should work,
```

```
                it's then freed when the DLL is unloaded.

                WARNING: If the function returns a non-valid pointer, Vim may
                crash!  This also happens if the function returns a number,
                because Vim thinks it's a pointer.
                For Win32 systems, {libname} should be the filename of the DLL
                without the ".DLL" suffix.  A full path is only required if
                the DLL is not in the usual places.
                For Unix: When compiling your own plugins, remember that the
                object code must be compiled as position-independent ('PIC').
                {only in Win32 and some Unix versions, when the |+libcall|
                feature is present}
                Examples: >
                        :echo libcall("libc.so", "getenv", "HOME")
<
                                                        *libcallnr()*
libcallnr({libname}, {funcname}, {argument})
                Just like |libcall()|, but used for a function that returns an
                int instead of a string.
                {only in Win32 on some Unix versions, when the |+libcall|
                feature is present}
                Examples: >
                        :echo libcallnr("/usr/lib/libc.so", "getpid", "")
                        :call libcallnr("libc.so", "printf", "Hello World!\n")
                        :call libcallnr("libc.so", "sleep", 10)
<
                                                        *line()*
line({expr})    The result is a Number, which is the line number of the file
                position given with {expr}.  The accepted positions are:
                    .       the cursor position
                    $       the last line in the current buffer
                    'x      position of mark x (if the mark is not set, 0 is
                            returned)
                    w0      first line visible in current window (one if the
                            display isn't updated, e.g. in silent Ex mode)
                    w$      last line visible in current window (this is one
                            less than "w0" if no lines are visible)
                    v       In Visual mode: the start of the Visual area (the
                            cursor is the end).  When not in Visual mode
                            returns the cursor position.  Differs from |'<| in
                            that it's updated right away.
                Note that a mark in another file can be used.  The line number
                then applies to another buffer.
                To get the column number use |col()|.  To get both use
                |getpos()|.
                Examples: >
                        line(".")               line number of the cursor
                        line("'t")              line number of mark t
                        line("'" . marker)      line number of mark marker
<                                                       *last-position-jump*
                This autocommand jumps to the last known position in a file
                just after opening it, if the '" mark is set: >
    :au BufReadPost *
        \ if line("'\"") > 1 && line("'\"") <= line("$") && &ft !~# 'commit'
        \ |   exe "normal! g`\""
        \ | endif

line2byte({lnum})                                       *line2byte()*
                Return the byte count from the start of the buffer for line
                {lnum}.  This includes the end-of-line character, depending on
                the 'fileformat' option for the current buffer.  The first
                line returns 1. 'encoding' matters, 'fileencoding' is ignored.
```

```
                This can also be used to get the byte count for the line just
                below the last line: >
                        line2byte(line("$") + 1)
<               This is the buffer size plus one.  If 'fileencoding' is empty
                it is the file size plus one.
                When {lnum} is invalid, or the |+byte_offset| feature has been
                disabled at compile time, -1 is returned.
                Also see |byte2line()|, |go| and |:goto|.

lispindent({lnum})                                          *lispindent()*
                Get the amount of indent for line {lnum} according the lisp
                indenting rules, as with 'lisp'.
                The indent is counted in spaces, the value of 'tabstop' is
                relevant.  {lnum} is used just like in |getline()|.
                When {lnum} is invalid or Vim was not compiled the
                |+lispindent| feature, -1 is returned.

localtime()                                                 *localtime()*
                Return the current time, measured as seconds since 1st Jan
                1970.  See also |strftime()| and |getftime()|.


log({expr})                                                 *log()*
                Return the natural logarithm (base e) of {expr} as a |Float|.
                {expr} must evaluate to a |Float| or a |Number| in the range
                (0, inf].
                Examples: >
                        :echo log(10)
<                       2.302585 >
                        :echo log(exp(5))
<                       5.0
                {only available when compiled with the |+float| feature}


log10({expr})                                               *log10()*
                Return the logarithm of Float {expr} to base 10 as a |Float|.
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo log10(1000)
<                       3.0 >
                        :echo log10(0.01)
<                       -2.0
                {only available when compiled with the |+float| feature}

luaeval({expr}[, {expr}])                                       *luaeval()*
                Evaluate Lua expression {expr} and return its result converted
                to Vim data structures. Second {expr} may hold additional
                argument accessible as _A inside first {expr}.
                Strings are returned as they are.
                Boolean objects are converted to numbers.
                Numbers are converted to |Float| values if vim was compiled
                with |+float| and to numbers otherwise.
                Dictionaries and lists obtained by vim.eval() are returned
                as-is.
                Other objects are returned as zero without any errors.
                See |lua-luaeval| for more details.
                {only available when compiled with the |+lua| feature}

map({expr1}, {expr2})                                       *map()*
                {expr1} must be a |List| or a |Dictionary|.
                Replace each item in {expr1} with the result of evaluating
                {expr2}.  {expr2} must be a |string| or |Funcref|.
```

```
                If {expr2} is a |string|, inside {expr2} |v:val| has the value
                of the current item.  For a |Dictionary| |v:key| has the key
                of the current item and for a |List| |v:key| has the index of
                the current item.
                Example: >
                        :call map(mylist, '"> " . v:val . " <"')
<               This puts "> " before and " <" after each item in "mylist".

                Note that {expr2} is the result of an expression and is then
                used as an expression again.  Often it is good to use a
                |literal-string| to avoid having to double backslashes.  You
                still have to double ' quotes

                If {expr2} is a |Funcref| it is called with two arguments:
                        1. The key or the index of the current item.
                        2. the value of the current item.
                The function must return the new value of the item. Example
                that changes each value by "key-value": >
                        func KeyValue(key, val)
                          return a:key . '-' . a:val
                        endfunc
                        call map(myDict, function('KeyValue'))
<               It is shorter when using a |lambda|: >
                        call map(myDict, {key, val -> key . '-' . val})
<               If you do not use "val" you can leave it out: >
                        call map(myDict, {key -> 'item: ' . key})
<
                The operation is done in-place.  If you want a |List| or
                |Dictionary| to remain unmodified make a copy first: >
                        :let tlist = map(copy(mylist), ' v:val . "\t"')

<               Returns {expr1}, the |List| or |Dictionary| that was filtered.
                When an error is encountered while evaluating {expr2} no
                further items in {expr1} are processed.  When {expr2} is a
                Funcref errors inside a function are ignored, unless it was
                defined with the "abort" flag.


maparg({name}[, {mode} [, {abbr} [, {dict}]]])                  *maparg()*
                When {dict} is omitted or zero: Return the rhs of mapping
                {name} in mode {mode}.  The returned String has special
                characters translated like in the output of the ":map" command
                listing.

                When there is no mapping for {name}, an empty String is
                returned.

                The {name} can have special key names, like in the ":map"
                command.

                {mode} can be one of these strings:
                        "n"     Normal
                        "v"     Visual (including Select)
                        "o"     Operator-pending
                        "i"     Insert
                        "c"     Cmd-line
                        "s"     Select
                        "x"     Visual
                        "l"     langmap |language-mapping|
                        "t"     Terminal-Job
                        ""      Normal, Visual and Operator-pending
```

```
                When {mode} is omitted, the modes for "" are used.

                When {abbr} is there and it is |TRUE| use abbreviations
                instead of mappings.

                When {dict} is there and it is |TRUE| return a dictionary
                containing all the information of the mapping with the
                following items:
                  "lhs"      The {lhs} of the mapping.
                  "rhs"      The {rhs} of the mapping as typed.
                  "silent"   1 for a |:map-silent| mapping, else 0.
                  "noremap"  1 if the {rhs} of the mapping is not remappable.
                  "expr"     1 for an expression mapping (|:map-<expr>|).
                  "buffer"   1 for a buffer local mapping (|:map-local|).
                  "mode"     Modes for which the mapping is defined. In
                             addition to the modes mentioned above, these
                             characters will be used:
                             " "     Normal, Visual and Operator-pending
                             "!"     Insert and Commandline mode
                                     (|mapmode-ic|)
                  "sid"      The script local ID, used for <sid> mappings
                             (|<SID>|).
                  "nowait"   Do not wait for other, longer mappings.
                             (|:map-<nowait>|).

                The mappings local to the current buffer are checked first,
                then the global mappings.
                This function can be used to map a key even when it's already
                mapped, and have it do the original mapping too.  Sketch: >
                        exe 'nnoremap <Tab> ==' . maparg('<Tab>', 'n')


mapcheck({name}[, {mode} [, {abbr}]])                   *mapcheck()*
                Check if there is a mapping that matches with {name} in mode
                {mode}.  See |maparg()| for {mode} and special names in
                {name}.
                When {abbr} is there and it is |TRUE| use abbreviations
                instead of mappings.
                A match happens with a mapping that starts with {name} and
                with a mapping which is equal to the start of {name}.

                        matches mapping "a"     "ab"    "abc" ~
                   mapcheck("a")        yes     yes      yes
                   mapcheck("abc")      yes     yes      yes
                   mapcheck("ax")       yes     no       no
                   mapcheck("b")        no      no       no

                The difference with maparg() is that mapcheck() finds a
                mapping that matches with {name}, while maparg() only finds a
                mapping for {name} exactly.
                When there is no mapping that starts with {name}, an empty
                String is returned.  If there is one, the rhs of that mapping
                is returned.  If there are several mappings that start with
                {name}, the rhs of one of them is returned.
                The mappings local to the current buffer are checked first,
                then the global mappings.
                This function can be used to check if a mapping can be added
                without being ambiguous.  Example: >
        :if mapcheck("_vv") == ""
        :   map _vv :set guifont=7x13<CR>
        :endif
<               This avoids adding the "_vv" mapping when there already is a
```

```
                mapping for "_v" or for "_vvv".

match({expr}, {pat}[, {start}[, {count}]])                      *match()*
                When {expr} is a |List| then this returns the index of the
                first item where {pat} matches.  Each item is used as a
                String, |Lists| and |Dictionaries| are used as echoed.
                Otherwise, {expr} is used as a String.  The result is a
                Number, which gives the index (byte offset) in {expr} where
                {pat} matches.
                A match at the first character or |List| item returns zero.
                If there is no match -1 is returned.
                For getting submatches see |matchlist()|.
                Example: >
                        :echo match("testing", "ing")   " results in 4
                        :echo match([1, 'x'], '\a')     " results in 1
<               See |string-match| for how {pat} is used.
                                                                *strpbrk()*
                Vim doesn't have a strpbrk() function.  But you can do: >
                        :let sepidx = match(line, '[.,;: \t]')
<                                                               *strcasestr()*
                Vim doesn't have a strcasestr() function.  But you can add
                "\c" to the pattern to ignore case: >
                        :let idx = match(haystack, '\cneedle')
<
                If {start} is given, the search starts from byte index
                {start} in a String or item {start} in a |List|.
                The result, however, is still the index counted from the
                first character/item.  Example: >
                        :echo match("testing", "ing", 2)
<               result is again "4". >
                        :echo match("testing", "ing", 4)
<               result is again "4". >
                        :echo match("testing", "t", 2)
<               result is "3".
                For a String, if {start} > 0 then it is like the string starts
                {start} bytes later, thus "^" will match at {start}.  Except
                when {count} is given, then it's like matches before the
                {start} byte are ignored (this is a bit complicated to keep it
                backwards compatible).
                For a String, if {start} < 0, it will be set to 0.  For a list
                the index is counted from the end.
                If {start} is out of range ({start} > strlen({expr}) for a
                String or {start} > len({expr}) for a |List|) -1 is returned.

                When {count} is given use the {count}'th match.  When a match
                is found in a String the search for the next one starts one
                character further.  Thus this example results in 1: >
                        echo match("testing", "..", 0, 2)
<               In a |List| the search continues in the next item.
                Note that when {count} is added the way {start} works changes,
                see above.

                See |pattern| for the patterns that are accepted.
                The 'ignorecase' option is used to set the ignore-caseness of
                the pattern.  'smartcase' is NOT used.  The matching is always
                done like 'magic' is set and 'cpoptions' is empty.

                                        *matchadd()* *E798* *E799* *E801*
matchadd({group}, {pattern}[, {priority}[, {id}[, {dict}]]])
                Defines a pattern to be highlighted in the current window (a
                "match").  It will be highlighted with {group}.  Returns an
                identification number (ID), which can be used to delete the
```

```
                match using |matchdelete()|.
                Matching is case sensitive and magic, unless case sensitivity
                or magicness are explicitly overridden in {pattern}.  The
                'magic', 'smartcase' and 'ignorecase' options are not used.
                The "Conceal" value is special, it causes the match to be
                concealed.

                The optional {priority} argument assigns a priority to the
                match.  A match with a high priority will have its
                highlighting overrule that of a match with a lower priority.
                A priority is specified as an integer (negative numbers are no
                exception).  If the {priority} argument is not specified, the
                default priority is 10.  The priority of 'hlsearch' is zero,
                hence all matches with a priority greater than zero will
                overrule it.  Syntax highlighting (see 'syntax') is a separate
                mechanism, and regardless of the chosen priority a match will
                always overrule syntax highlighting.

                The optional {id} argument allows the request for a specific
                match ID.  If a specified ID is already taken, an error
                message will appear and the match will not be added.  An ID
                is specified as a positive integer (zero excluded).  IDs 1, 2
                and 3 are reserved for |:match|, |:2match| and |:3match|,
                respectively.  If the {id} argument is not specified or -1,
                |matchadd()| automatically chooses a free ID.

                The optional {dict} argument allows for further custom
                values. Currently this is used to specify a match specific
                conceal character that will be shown for |hl-Conceal|
                highlighted matches. The dict can have the following members:

                        conceal     Special character to show instead of the
                                    match (only for |hl-Conceal| highlighted
                                    matches, see |:syn-cchar|)

                The number of matches is not limited, as it is the case with
                the |:match| commands.

                Example: >
                        :highlight MyGroup ctermbg=green guibg=green
                        :let m = matchadd("MyGroup", "TODO")
<               Deletion of the pattern: >
                        :call matchdelete(m)

<               A list of matches defined by |matchadd()| and |:match| are
                available from |getmatches()|.  All matches can be deleted in
                one operation by |clearmatches()|.

                                                        *matchaddpos()*
matchaddpos({group}, {pos}[, {priority}[, {id}[, {dict}]]])
                Same as |matchadd()|, but requires a list of positions {pos}
                instead of a pattern. This command is faster than |matchadd()|
                because it does not require to handle regular expressions and
                sets buffer line boundaries to redraw screen. It is supposed
                to be used when fast match additions and deletions are
                required, for example to highlight matching parentheses.

                The list {pos} can contain one of these items:
                - A number.  This whole line will be highlighted.  The first
                  line has number 1.
                - A list with one number, e.g., [23]. The whole line with this
                  number will be highlighted.
```

```
                - A list with two numbers, e.g., [23, 11]. The first number is
                  the line number, the second one is the column number (first
                  column is 1, the value must correspond to the byte index as
                  |col()| would return).  The character at this position will
                  be highlighted.
                - A list with three numbers, e.g., [23, 11, 3]. As above, but
                  the third number gives the length of the highlight in bytes.

                The maximum number of positions is 8.

                Example: >
                        :highlight MyGroup ctermbg=green guibg=green
                        :let m = matchaddpos("MyGroup", [[23, 24], 34])
<               Deletion of the pattern: >
                        :call matchdelete(m)

<               Matches added by |matchaddpos()| are returned by
                |getmatches()| with an entry "pos1", "pos2", etc., with the
                value a list like the {pos} item.
                These matches cannot be set via |setmatches()|, however they
                can still be deleted by |clearmatches()|.

matcharg({nr})                                                  *matcharg()*
                Selects the {nr} match item, as set with a |:match|,
                |:2match| or |:3match| command.
                Return a |List| with two elements:
                        The name of the highlight group used
                        The pattern used.
                When {nr} is not 1, 2 or 3 returns an empty |List|.
                When there is no match item set returns ['', ''].
                This is useful to save and restore a |:match|.
                Highlighting matches using the |:match| commands are limited
                to three matches. |matchadd()| does not have this limitation.

matchdelete({id})                               *matchdelete()* *E802* *E803*
                Deletes a match with ID {id} previously defined by |matchadd()|
                or one of the |:match| commands.  Returns 0 if successful,
                otherwise -1.  See example for |matchadd()|.  All matches can
                be deleted in one operation by |clearmatches()|.

matchend({expr}, {pat}[, {start}[, {count}]])                   *matchend()*
                Same as |match()|, but return the index of first character
                after the match.  Example: >
                        :echo matchend("testing", "ing")
<               results in "7".
                                                        *strspn()* *strcspn()*
                Vim doesn't have a strspn() or strcspn() function, but you can
                do it with matchend(): >
                        :let span = matchend(line, '[a-zA-Z]')
                        :let span = matchend(line, '[^a-zA-Z]')
<               Except that -1 is returned when there are no matches.

                The {start}, if given, has the same meaning as for |match()|. >
                        :echo matchend("testing", "ing", 2)
<               results in "7". >
                        :echo matchend("testing", "ing", 5)
<               result is "-1".
                When {expr} is a |List| the result is equal to |match()|.

matchlist({expr}, {pat}[, {start}[, {count}]])                  *matchlist()*
                Same as |match()|, but return a |List|.  The first item in the
                list is the matched string, same as what matchstr() would
```

```
                return.  Following items are submatches, like "\1", "\2", etc.
                in |:substitute|.  When an optional submatch didn't match an
                empty string is used.  Example: >
                        echo matchlist('acd', '\(a\)\?\(b\)\?\(c\)\?\(.*\)')
<               Results in: ['acd', 'a', '', 'c', 'd', '', '', '', '', '']
                When there is no match an empty list is returned.

matchstr({expr}, {pat}[, {start}[, {count}]])                   *matchstr()*
                Same as |match()|, but return the matched string.  Example: >
                        :echo matchstr("testing", "ing")
<               results in "ing".
                When there is no match "" is returned.
                The {start}, if given, has the same meaning as for |match()|. >
                        :echo matchstr("testing", "ing", 2)
<               results in "ing". >
                        :echo matchstr("testing", "ing", 5)
<               result is "".
                When {expr} is a |List| then the matching item is returned.
                The type isn't changed, it's not necessarily a String.

matchstrpos({expr}, {pat}[, {start}[, {count}]])                *matchstrpos()*
                Same as |matchstr()|, but return the matched string, the start
                position and the end position of the match.  Example: >
                        :echo matchstrpos("testing", "ing")
<               results in ["ing", 4, 7].
                When there is no match ["", -1, -1] is returned.
                The {start}, if given, has the same meaning as for |match()|. >
                        :echo matchstrpos("testing", "ing", 2)
<               results in ["ing", 4, 7]. >
                        :echo matchstrpos("testing", "ing", 5)
<               result is ["", -1, -1].
                When {expr} is a |List| then the matching item, the index
                of first item where {pat} matches, the start position and the
                end position of the match are returned. >
                        :echo matchstrpos([1, '__x'], '\a')
<               result is ["x", 1, 2, 3].
                The type isn't changed, it's not necessarily a String.

                                                        *max()*
max({expr})     Return the maximum value of all items in {expr}.
                {expr} can be a list or a dictionary.  For a dictionary,
                it returns the maximum of all values in the dictionary.
                If {expr} is neither a list nor a dictionary, or one of the
                items in {expr} cannot be used as a Number this results in
                an error.  An empty |List| or |Dictionary| results in zero.

                                                        *min()*
min({expr})     Return the minimum value of all items in {expr}.
                {expr} can be a list or a dictionary.  For a dictionary,
                it returns the minimum of all values in the dictionary.
                If {expr} is neither a list nor a dictionary, or one of the
                items in {expr} cannot be used as a Number this results in
                an error.  An empty |List| or |Dictionary| results in zero.

                                                        *mkdir()* *E739*
mkdir({name} [, {path} [, {prot}]])
                Create directory {name}.
                If {path} is "p" then intermediate directories are created as
                necessary.  Otherwise it must be "".
                If {prot} is given it is used to set the protection bits of
                the new directory.  The default is 0755 (rwxr-xr-x: r/w for
                the user readable for others).  Use 0700 to make it unreadable
```

```
                for others.  This is only used for the last part of {name}.
                Thus if you create /tmp/foo/bar then /tmp/foo will be created
                with 0755.
                Example: >
                        :call mkdir($HOME . "/tmp/foo/bar", "p", 0700)
<               This function is not available in the |sandbox|.
                Not available on all systems.  To check use: >
                        :if exists("*mkdir")
<
                                                        *mode()*
mode([expr])    Return a string that indicates the current mode.
                If [expr] is supplied and it evaluates to a non-zero Number or
                a non-empty String (|non-zero-arg|), then the full mode is
                returned, otherwise only the first letter is returned.

                        n       Normal, Terminal-Normal
                        no      Operator-pending
                        v       Visual by character
                        V       Visual by line
                        CTRL-V  Visual blockwise
                        s       Select by character
                        S       Select by line
                        CTRL-S  Select blockwise
                        i       Insert
                        ic      Insert mode completion |compl-generic|
                        ix      Insert mode |i_CTRL-X| completion
                        R       Replace |R|
                        Rc      Replace mode completion |compl-generic|
                        Rv      Virtual Replace |gR|
                        Rx      Replace mode |i_CTRL-X| completion
                        c       Command-line editing
                        cv      Vim Ex mode |gQ|
                        ce      Normal Ex mode |Q|
                        r       Hit-enter prompt
                        rm      The -- more -- prompt
                        r?      A |:confirm| query of some sort
                        !       Shell or external command is executing
                        t       Terminal-Job mode: keys go to the job
                This is useful in the 'statusline' option or when used
                with |remote_expr()| In most other places it always returns
                "c" or "n".
                Also see |visualmode()|.

mzeval({expr})                                                  *mzeval()*
                Evaluate MzScheme expression {expr} and return its result
                converted to Vim data structures.
                Numbers and strings are returned as they are.
                Pairs (including lists and improper lists) and vectors are
                returned as Vim |Lists|.
                Hash tables are represented as Vim |Dictionary| type with keys
                converted to strings.
                All other types are converted to string with display function.
                Examples: >
                    :mz (define l (list 1 2 3))
                    :mz (define h (make-hash)) (hash-set! h "list" l)
                    :echo mzeval("l")
                    :echo mzeval("h")
<
                {only available when compiled with the |+mzscheme| feature}

nextnonblank({lnum})                                    *nextnonblank()*
                Return the line number of the first line at or below {lnum}
```

```
                that is not blank.  Example: >
                        if getline(nextnonblank(1)) =~ "Java"
<               When {lnum} is invalid or there is no non-blank line at or
                below it, zero is returned.
                See also |prevnonblank()|.

nr2char({expr}[, {utf8}])                               *nr2char()*
                Return a string with a single character, which has the number
                value {expr}.  Examples: >
                        nr2char(64)             returns "@"
                        nr2char(32)             returns " "
<               When {utf8} is omitted or zero, the current 'encoding' is used.
                Example for "utf-8": >
                        nr2char(300)            returns I with bow character
<               With {utf8} set to 1, always return utf-8 characters.
                Note that a NUL character in the file is specified with
                nr2char(10), because NULs are represented with newline
                characters.  nr2char(0) is a real NUL and terminates the
                string, thus results in an empty string.

or({expr}, {expr})                                      *or()*
                Bitwise OR on the two arguments.  The arguments are converted
                to a number.  A List, Dict or Float argument causes an error.
                Example: >
                        :let bits = or(bits, 0x80)


pathshorten({expr})                                     *pathshorten()*
                Shorten directory names in the path {expr} and return the
                result.  The tail, the file name, is kept as-is.  The other
                components in the path are reduced to single letters.  Leading
                '~' and '.' characters are kept.  Example: >
                        :echo pathshorten('~/.vim/autoload/myfile.vim')
<                       ~/.v/a/myfile.vim ~
                It doesn't matter if the path exists or not.

perleval({expr})                                        *perleval()*
                Evaluate Perl expression {expr} in scalar context and return
                its result converted to Vim data structures. If value can't be
                converted, it is returned as a string Perl representation.
                Note: If you want an array or hash, {expr} must return a
                reference to it.
                Example: >
                        :echo perleval('[1 .. 4]')
<                       [1, 2, 3, 4]
                {only available when compiled with the |+perl| feature}

pow({x}, {y})                                           *pow()*
                Return the power of {x} to the exponent {y} as a |Float|.
                {x} and {y} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo pow(3, 3)
<                       27.0 >
                        :echo pow(2, 16)
<                       65536.0 >
                        :echo pow(32, 0.20)
<                       2.0
                {only available when compiled with the |+float| feature}

prevnonblank({lnum})                                    *prevnonblank()*
                Return the line number of the first line at or above {lnum}
                that is not blank.  Example: >
```

```
                        let ind = indent(prevnonblank(v:lnum - 1))
<               When {lnum} is invalid or there is no non-blank line at or
                above it, zero is returned.
                Also see |nextnonblank()|.


printf({fmt}, {expr1} ...)                                  *printf()*
                Return a String with {fmt}, where "%" items are replaced by
                the formatted form of their respective arguments.  Example: >
                        printf("%4d: E%d %.30s", lnum, errno, msg)
<               May result in:
                        "  99: E42 asdfasdfasdfasdfasdfasdfasdfas" ~

                Often used items are:
                  %s    string
                  %6S   string right-aligned in 6 display cells
                  %6s   string right-aligned in 6 bytes
                  %.9s  string truncated to 9 bytes
                  %c    single byte
                  %d    decimal number
                  %5d   decimal number padded with spaces to 5 characters
                  %x    hex number
                  %04x  hex number padded with zeros to at least 4 characters
                  %X    hex number using upper case letters
                  %o    octal number
                  %08b  binary number padded with zeros to at least 8 chars
                  %f    floating point number as 12.23, inf, -inf or nan
                  %F    floating point number as 12.23, INF, -INF or NAN
                  %e    floating point number as 1.23e3, inf, -inf or nan
                  %E    floating point number as 1.23E3, INF, -INF or NAN
                  %g    floating point number, as %f or %e depending on value
                  %G    floating point number, as %F or %E depending on value
                  %%    the % character itself

                Conversion specifications start with '%' and end with the
                conversion type.  All other characters are copied unchanged to
                the result.

                The "%" starts a conversion specification.  The following
                arguments appear in sequence:

                        %  [flags]  [field-width]  [.precision]  type

                flags
                        Zero or more of the following flags:

                    #         The value should be converted to an "alternate
                              form".  For c, d, and s conversions, this option
                              has no effect.  For o conversions, the precision
                              of the number is increased to force the first
                              character of the output string to a zero (except
                              if a zero value is printed with an explicit
                              precision of zero).
                              For b and B conversions, a non-zero result has
                              the string "0b" (or "0B" for B conversions)
                              prepended to it.
                              For x and X conversions, a non-zero result has
                              the string "0x" (or "0X" for X conversions)
                              prepended to it.

                    0 (zero)  Zero padding.  For all conversions the converted
                              value is padded on the left with zeros rather
```

```
                    than blanks.  If a precision is given with a
                    numeric conversion (d, b, B, o, x, and X), the 0
                    flag is ignored.

                -   A negative field width flag; the converted value
                    is to be left adjusted on the field boundary.
                    The converted value is padded on the right with
                    blanks, rather than on the left with blanks or
                    zeros.  A - overrides a 0 if both are given.

                ' ' (space)  A blank should be left before a positive
                    number produced by a signed conversion (d).

                +   A sign must always be placed before a number
                    produced by a signed conversion.  A + overrides
                    a space if both are used.

            field-width
                    An optional decimal digit string specifying a minimum
                    field width.  If the converted value has fewer bytes
                    than the field width, it will be padded with spaces on
                    the left (or right, if the left-adjustment flag has
                    been given) to fill out the field width.

            .precision
                    An optional precision, in the form of a period '.'
                    followed by an optional digit string.  If the digit
                    string is omitted, the precision is taken as zero.
                    This gives the minimum number of digits to appear for
                    d, o, x, and X conversions, or the maximum number of
                    bytes to be printed from a string for s conversions.
                    For floating point it is the number of digits after
                    the decimal point.

            type
                    A character that specifies the type of conversion to
                    be applied, see below.

            A field width or precision, or both, may be indicated by an
            asterisk '*' instead of a digit string.  In this case, a
            Number argument supplies the field width or precision.  A
            negative field width is treated as a left adjustment flag
            followed by a positive field width; a negative precision is
            treated as though it were missing.  Example: >
                    :echo printf("%d: %.*s", nr, width, line)
<           This limits the length of the text used from "line" to
            "width" bytes.

            The conversion specifiers and their meanings are:

                            *printf-d* *printf-b* *printf-B* *printf-o*
                            *printf-x* *printf-X*
            dbBoxX  The Number argument is converted to signed decimal
                    (d), unsigned binary (b and B), unsigned octal (o), or
                    unsigned hexadecimal (x and X) notation.  The letters
                    "abcdef" are used for x conversions; the letters
                    "ABCDEF" are used for X conversions.
                    The precision, if any, gives the minimum number of
                    digits that must appear; if the converted value
                    requires fewer digits, it is padded on the left with
                    zeros.
                    In no case does a non-existent or small field width
```

cause truncation of a numeric field; if the result of
a conversion is wider than the field width, the field
is expanded to contain the conversion result.
The 'h' modifier indicates the argument is 16 bits.
The 'l' modifier indicates the argument is 32 bits.
The 'L' modifier indicates the argument is 64 bits.
Generally, these modifiers are not useful. They are
ignored when type is known from the argument.

i alias for d
D alias for ld
U alias for lu
O alias for lo

printf-c
c The Number argument is converted to a byte, and the
resulting character is written.

printf-s
s The text of the String argument is used. If a
precision is specified, no more bytes than the number
specified are used.
If the argument is not a String type, it is
automatically converted to text with the same format
as ":echo".
printf-S
S The text of the String argument is used. If a
precision is specified, no more display cells than the
number specified are used. Without the |+multi_byte|
feature works just like 's'.

printf-f *E807*
f F The Float argument is converted into a string of the
form 123.456. The precision specifies the number of
digits after the decimal point. When the precision is
zero the decimal point is omitted. When the precision
is not specified 6 is used. A really big number
(out of range or dividing by zero) results in "inf"
or "-inf" with %f (INF or -INF with %F).
"0.0 / 0.0" results in "nan" with %f (NAN with %F).
Example: >
echo printf("%.2f", 12.115)
< 12.12
Note that roundoff depends on the system libraries.
Use |round()| when in doubt.

printf-e *printf-E*
e E The Float argument is converted into a string of the
form 1.234e+03 or 1.234E+03 when using 'E'. The
precision specifies the number of digits after the
decimal point, like with 'f'.

printf-g *printf-G*
g G The Float argument is converted like with 'f' if the
value is between 0.001 (inclusive) and 10000000.0
(exclusive). Otherwise 'e' is used for 'g' and 'E'
for 'G'. When no precision is specified superfluous
zeroes and '+' signs are removed, except for the zero
immediately after the decimal point. Thus 10000000.0
results in 1.0e7.

printf-%

```
                %       A '%' is written.  No argument is converted.  The
                        complete conversion specification is "%%".

              When a Number argument is expected a String argument is also
              accepted and automatically converted.
              When a Float or String argument is expected a Number argument
              is also accepted and automatically converted.
              Any other argument type results in an error message.

                                                        *E766* *E767*
              The number of {exprN} arguments must exactly match the number
              of "%" items.  If there are not sufficient or too many
              arguments an error is given.  Up to 18 arguments can be used.


pumvisible()                                            *pumvisible()*
              Returns non-zero when the popup menu is visible, zero
              otherwise.  See |ins-completion-menu|.
              This can be used to avoid some things that would remove the
              popup menu.

py3eval({expr})                                         *py3eval()*
              Evaluate Python expression {expr} and return its result
              converted to Vim data structures.
              Numbers and strings are returned as they are (strings are
              copied though, Unicode strings are additionally converted to
              'encoding').
              Lists are represented as Vim |List| type.
              Dictionaries are represented as Vim |Dictionary| type with
              keys converted to strings.
              {only available when compiled with the |+python3| feature}

                                                        *E858* *E859*
pyeval({expr})                                          *pyeval()*
              Evaluate Python expression {expr} and return its result
              converted to Vim data structures.
              Numbers and strings are returned as they are (strings are
              copied though).
              Lists are represented as Vim |List| type.
              Dictionaries are represented as Vim |Dictionary| type,
              non-string keys result in error.
              {only available when compiled with the |+python| feature}

pyxeval({expr})                                         *pyxeval()*
              Evaluate Python expression {expr} and return its result
              converted to Vim data structures.
              Uses Python 2 or 3, see |python_x| and 'pyxversion'.
              See also: |pyeval()|, |py3eval()|
              {only available when compiled with the |+python| or the
              |+python3| feature}

                                                        *E726* *E727*
range({expr} [, {max} [, {stride}]])                            *range()*
              Returns a |List| with Numbers:
              - If only {expr} is specified: [0, 1, ..., {expr} - 1]
              - If {max} is specified: [{expr}, {expr} + 1, ..., {max}]
              - If {stride} is specified: [{expr}, {expr} + {stride}, ...,
                {max}] (increasing {expr} with {stride} each time, not
                producing a value past {max}).
              When the maximum is one before the start the result is an
              empty list.  When the maximum is more than one before the
              start this is an error.
```

```
                Examples: >
                        range(4)                " [0, 1, 2, 3]
                        range(2, 4)             " [2, 3, 4]
                        range(2, 9, 3)          " [2, 5, 8]
                        range(2, -2, -1)        " [2, 1, 0, -1, -2]
                        range(0)                " []
                        range(2, 0)             " error!
<
                                                        *readfile()*
readfile({fname} [, {binary} [, {max}]])
                Read file {fname} and return a |List|, each line of the file
                as an item.  Lines are broken at NL characters.  Macintosh
                files separated with CR will result in a single long line
                (unless a NL appears somewhere).
                All NUL characters are replaced with a NL character.
                When {binary} contains "b" binary mode is used:
                - When the last line ends in a NL an extra empty list item is
                  added.
                - No CR characters are removed.
                Otherwise:
                - CR characters that appear before a NL are removed.
                - Whether the last line ends in a NL or not does not matter.
                - When 'encoding' is Unicode any UTF-8 byte order mark is
                  removed from the text.
                When {max} is given this specifies the maximum number of lines
                to be read.  Useful if you only want to check the first ten
                lines of a file: >
                        :for line in readfile(fname, '', 10)
                        :  if line =~ 'Date' | echo line | endif
                        :endfor
<               When {max} is negative -{max} lines from the end of the file
                are returned, or as many as there are.
                When {max} is zero the result is an empty list.
                Note that without {max} the whole file is read into memory.
                Also note that there is no recognition of encoding.  Read a
                file into a buffer if you need to.
                When the file can't be opened an error message is given and
                the result is an empty list.
                Also see |writefile()|.

reltime([{start} [, {end}]])                            *reltime()*
                Return an item that represents a time value.  The format of
                the item depends on the system.  It can be passed to
                |reltimestr()| to convert it to a string  or |reltimefloat()|
                to convert to a Float.
                Without an argument it returns the current time.
                With one argument is returns the time passed since the time
                specified in the argument.
                With two arguments it returns the time passed between {start}
                and {end}.
                The {start} and {end} arguments must be values returned by
                reltime().
                {only available when compiled with the |+reltime| feature}

reltimefloat({time})                            *reltimefloat()*
                Return a Float that represents the time value of {time}.
                Example: >
                        let start = reltime()
                        call MyFunction()
                        let seconds = reltimefloat(reltime(start))
<               See the note of reltimestr() about overhead.
                Also see |profiling|.
```

```
                {only available when compiled with the |+reltime| feature}

reltimestr({time})                                  *reltimestr()*
                Return a String that represents the time value of {time}.
                This is the number of seconds, a dot and the number of
                microseconds.  Example: >
                        let start = reltime()
                        call MyFunction()
                        echo reltimestr(reltime(start))
<               Note that overhead for the commands will be added to the time.
                The accuracy depends on the system.
                Leading spaces are used to make the string align nicely.  You
                can use split() to remove it. >
                        echo split(reltimestr(reltime(start)))[0]
<               Also see |profiling|.
                {only available when compiled with the |+reltime| feature}

                                                    *remote_expr()* *E449*
remote_expr({server}, {string} [, {idvar} [, {timeout}]])
                Send the {string} to {server}.  The string is sent as an
                expression and the result is returned after evaluation.
                The result must be a String or a |List|.  A |List| is turned
                into a String by joining the items with a line break in
                between (not at the end), like with join(expr, "\n").
                If {idvar} is present and not empty, it is taken as the name
                of a variable and a {serverid} for later use with
                remote_read() is stored there.
                If {timeout} is given the read times out after this many
                seconds.  Otherwise a timeout of 600 seconds is used.
                See also |clientserver| |RemoteReply|.
                This function is not available in the |sandbox|.
                {only available when compiled with the |+clientserver| feature}
                Note: Any errors will cause a local error message to be issued
                and the result will be the empty string.
                Examples: >
                        :echo remote_expr("gvim", "2+2")
                        :echo remote_expr("gvim1", "b:current_syntax")
<

remote_foreground({server})                         *remote_foreground()*
                Move the Vim server with the name {server} to the foreground.
                This works like: >
                        remote_expr({server}, "foreground()")
<               Except that on Win32 systems the client does the work, to work
                around the problem that the OS doesn't always allow the server
                to bring itself to the foreground.
                Note: This does not restore the window if it was minimized,
                like foreground() does.
                This function is not available in the |sandbox|.
                {only in the Win32, Athena, Motif and GTK GUI versions and the
                Win32 console version}


remote_peek({serverid} [, {retvar}])            *remote_peek()*
                Returns a positive number if there are available strings
                from {serverid}.  Copies any reply string into the variable
                {retvar} if specified.  {retvar} must be a string with the
                name of a variable.
                Returns zero if none are available.
                Returns -1 if something is wrong.
                See also |clientserver|.
                This function is not available in the |sandbox|.
```

```
                {only available when compiled with the |+clientserver| feature}
                Examples: >
                        :let repl = ""
                        :echo "PEEK: ".remote_peek(id, "repl").": ".repl

remote_read({serverid}, [{timeout}])                    *remote_read()*
                Return the oldest available reply from {serverid} and consume
                it.  Unless a {timeout} in seconds is given, it blocks until a
                reply is available.
                See also |clientserver|.
                This function is not available in the |sandbox|.
                {only available when compiled with the |+clientserver| feature}
                Example: >
                        :echo remote_read(id)
<
                                                        *remote_send()* *E241*
remote_send({server}, {string} [, {idvar}])
                Send the {string} to {server}.  The string is sent as input
                keys and the function returns immediately.  At the Vim server
                the keys are not mapped |:map|.
                If {idvar} is present, it is taken as the name of a variable
                and a {serverid} for later use with remote_read() is stored
                there.
                See also |clientserver| |RemoteReply|.
                This function is not available in the |sandbox|.
                {only available when compiled with the |+clientserver| feature}

                Note: Any errors will be reported in the server and may mess
                up the display.
                Examples: >
                :echo remote_send("gvim", ":DropAndReply ".file, "serverid").
                 \ remote_read(serverid)

                :autocmd NONE RemoteReply *
                 \ echo remote_read(expand("<amatch>"))
                :echo remote_send("gvim", ":sleep 10 | echo ".
                 \ 'server2client(expand("<client>"), "HELLO")<CR>')
<
                                        *remote_startserver()* *E941* *E942*
remote_startserver({name})
                Become the server {name}.  This fails if already running as a
                server, when |v:servername| is not empty.
                {only available when compiled with the |+clientserver| feature}

remove({list}, {idx} [, {end}])                         *remove()*
                Without {end}: Remove the item at {idx} from |List| {list} and
                return the item.
                With {end}: Remove items from {idx} to {end} (inclusive) and
                return a List with these items.  When {idx} points to the same
                item as {end} a list with one item is returned.  When {end}
                points to an item before {idx} this is an error.
                See |list-index| for possible values of {idx} and {end}.
                Example: >
                        :echo "last item: " . remove(mylist, -1)
                        :call remove(mylist, 0, 9)
remove({dict}, {key})
                Remove the entry from {dict} with key {key}.  Example: >
                        :echo "removed " . remove(dict, "one")
<               If there is no {key} in {dict} this is an error.

                Use |delete()| to remove a file.
```

```
rename({from}, {to})                                    *rename()*
               Rename the file by the name {from} to the name {to}.  This
               should also work to move files across file systems.  The
               result is a Number, which is 0 if the file was renamed
               successfully, and non-zero when the renaming failed.
               NOTE: If {to} exists it is overwritten without warning.
               This function is not available in the |sandbox|.

repeat({expr}, {count})                                 *repeat()*
               Repeat {expr} {count} times and return the concatenated
               result.  Example: >
                       :let separator = repeat('-', 80)
<              When {count} is zero or negative the result is empty.
               When {expr} is a |List| the result is {expr} concatenated
               {count} times.  Example: >
                       :let longlist = repeat(['a', 'b'], 3)
<              Results in ['a', 'b', 'a', 'b', 'a', 'b'].


resolve({filename})                                     *resolve()* *E655*
               On MS-Windows, when {filename} is a shortcut (a .lnk file),
               returns the path the shortcut points to in a simplified form.
               On Unix, repeat resolving symbolic links in all path
               components of {filename} and return the simplified result.
               To cope with link cycles, resolving of symbolic links is
               stopped after 100 iterations.
               On other systems, return the simplified {filename}.
               The simplification step is done as by |simplify()|.
               resolve() keeps a leading path component specifying the
               current directory (provided the result is still a relative
               path name) and also keeps a trailing path separator.

                                                        *reverse()*
reverse({list}) Reverse the order of items in {list} in-place.  Returns
               {list}.
               If you want a list to remain unmodified make a copy first: >
                       :let revlist = reverse(copy(mylist))

round({expr})                                                   *round()*
               Round off {expr} to the nearest integral value and return it
               as a |Float|.  If {expr} lies halfway between two integral
               values, then use the larger one (away from zero).
               {expr} must evaluate to a |Float| or a |Number|.
               Examples: >
                       echo round(0.456)
<                      0.0  >
                       echo round(4.5)
<                      5.0 >
                       echo round(-4.5)
<                      -5.0
               {only available when compiled with the |+float| feature}

screenattr(row, col)                                    *screenattr()*
               Like |screenchar()|, but return the attribute.  This is a rather
               arbitrary number that can only be used to compare to the
               attribute at other positions.

screenchar(row, col)                                    *screenchar()*
               The result is a Number, which is the character at position
               [row, col] on the screen.  This works for every possible
               screen position, also status lines, window separators and the
               command line.  The top left position is row one, column one
```

```
                The character excludes composing characters.  For double-byte
                encodings it may only be the first byte.
                This is mainly to be used for testing.
                Returns -1 when row or col is out of range.

screencol()                                                     *screencol()*
                The result is a Number, which is the current screen column of
                the cursor. The leftmost column has number 1.
                This function is mainly used for testing.

                Note: Always returns the current screen column, thus if used
                in a command (e.g. ":echo screencol()") it will return the
                column inside the command line, which is 1 when the command is
                executed. To get the cursor position in the file use one of
                the following mappings: >
                        nnoremap <expr> GG ":echom ".screencol()."\n"
                        nnoremap <silent> GG :echom screencol()<CR>
<
screenrow()                                                     *screenrow()*
                The result is a Number, which is the current screen row of the
                cursor.  The top line has number one.
                This function is mainly used for testing.
                Alternatively you can use |winline()|.

                Note: Same restrictions as with |screencol()|.

search({pattern} [, {flags} [, {stopline} [, {timeout}]]])      *search()*
                Search for regexp pattern {pattern}.  The search starts at the
                cursor position (you can use |cursor()| to set it).

                When a match has been found its line number is returned.
                If there is no match a 0 is returned and the cursor doesn't
                move.  No error message is given.

                {flags} is a String, which can contain these character flags:
                'b'     search Backward instead of forward
                'c'     accept a match at the Cursor position
                'e'     move to the End of the match
                'n'     do Not move the cursor
                'p'     return number of matching sub-Pattern (see below)
                's'     Set the ' mark at the previous location of the cursor
                'w'     Wrap around the end of the file
                'W'     don't Wrap around the end of the file
                'z'     start searching at the cursor column instead of zero
                If neither 'w' or 'W' is given, the 'wrapscan' option applies.

                If the 's' flag is supplied, the ' mark is set, only if the
                cursor is moved. The 's' flag cannot be combined with the 'n'
                flag.

                'ignorecase', 'smartcase' and 'magic' are used.

                When the 'z' flag is not given, searching always starts in
                column zero and then matches before the cursor are skipped.
                When the 'c' flag is present in 'cpo' the next search starts
                after the match.  Without the 'c' flag the next search starts
                one column further.

                When the {stopline} argument is given then the search stops
                after searching this line.  This is useful to restrict the
                search to a range of lines.  Examples: >
                        let match = search('(', 'b', line("w0"))
```

```
                        let end = search('END', '', line("w$"))
<               When {stopline} is used and it is not zero this also implies
                that the search does not wrap around the end of the file.
                A zero value is equal to not giving the argument.

                When the {timeout} argument is given the search stops when
                more than this many milliseconds have passed.  Thus when
                {timeout} is 500 the search stops after half a second.
                The value must not be negative.  A zero value is like not
                giving the argument.
                {only available when compiled with the |+reltime| feature}

                                                        *search()-sub-match*
                With the 'p' flag the returned value is one more than the
                first sub-match in \(\).  One if none of them matched but the
                whole pattern did match.
                To get the column number too use |searchpos()|.

                The cursor will be positioned at the match, unless the 'n'
                flag is used.

                Example (goes over all files in the argument list): >
                    :let n = 1
                    :while n <= argc()      " loop over all files in arglist
                    :  exe "argument " . n
                    :  " start at the last char in the file and wrap for the
                    :  " first search to find match at start of file
                    :  normal G$
                    :  let flags = "w"
                    :  while search("foo", flags) > 0
                    :    s/foo/bar/g
                    :    let flags = "W"
                    :  endwhile
                    :  update               " write the file if modified
                    :  let n = n + 1
                    :endwhile
<
                Example for using some flags: >
                    :echo search('\<if\|\(else\)\|\(endif\)', 'ncpe')
<               This will search for the keywords "if", "else", and "endif"
                under or after the cursor.  Because of the 'p' flag, it
                returns 1, 2, or 3 depending on which keyword is found, or 0
                if the search fails.  With the cursor on the first word of the
                line:
                    if (foo == 0) | let foo = foo + 1 | endif ~
                the function returns 1.  Without the 'c' flag, the function
                finds the "endif" and returns 3.  The same thing happens
                without the 'e' flag if the cursor is on the "f" of "if".
                The 'n' flag tells the function not to move the cursor.


searchdecl({name} [, {global} [, {thisblock}]])                 *searchdecl()*
                Search for the declaration of {name}.

                With a non-zero {global} argument it works like |gD|, find
                first match in the file.  Otherwise it works like |gd|, find
                first match in the function.

                With a non-zero {thisblock} argument matches in a {} block
                that ends before the cursor position are ignored.  Avoids
                finding variable declarations only valid in another scope.
```

```
                Moves the cursor to the found match.
                Returns zero for success, non-zero for failure.
                Example: >
                        if searchdecl('myvar') == 0
                           echo getline('.')
                        endif
<
                                                        *searchpair()*
searchpair({start}, {middle}, {end} [, {flags} [, {skip}
                                [, {stopline} [, {timeout}]]]])
                Search for the match of a nested start-end pair.  This can be
                used to find the "endif" that matches an "if", while other
                if/endif pairs in between are ignored.
                The search starts at the cursor.  The default is to search
                forward, include 'b' in {flags} to search backward.
                If a match is found, the cursor is positioned at it and the
                line number is returned.  If no match is found 0 or -1 is
                returned and the cursor doesn't move.  No error message is
                given.

                {start}, {middle} and {end} are patterns, see |pattern|.  They
                must not contain \( \) pairs.  Use of \%( \) is allowed.  When
                {middle} is not empty, it is found when searching from either
                direction, but only when not in a nested start-end pair.  A
                typical use is: >
                        searchpair('\<if\>', '\<else\>', '\<endif\>')
<               By leaving {middle} empty the "else" is skipped.

                {flags} 'b', 'c', 'n', 's', 'w' and 'W' are used like with
                |search()|.  Additionally:
                'r'     Repeat until no more matches found; will find the
                        outer pair.  Implies the 'W' flag.
                'm'     Return number of matches instead of line number with
                        the match; will be > 1 when 'r' is used.
                Note: it's nearly always a good idea to use the 'W' flag, to
                avoid wrapping around the end of the file.

                When a match for {start}, {middle} or {end} is found, the
                {skip} expression is evaluated with the cursor positioned on
                the start of the match.  It should return non-zero if this
                match is to be skipped.  E.g., because it is inside a comment
                or a string.
                When {skip} is omitted or empty, every match is accepted.
                When evaluating {skip} causes an error the search is aborted
                and -1 returned.

                For {stopline} and {timeout} see |search()|.

                The value of 'ignorecase' is used.  'magic' is ignored, the
                patterns are used like it's on.

                The search starts exactly at the cursor.  A match with
                {start}, {middle} or {end} at the next character, in the
                direction of searching, is the first one found.  Example: >
                        if 1
                          if 2
                          endif 2
                        endif 1
<               When starting at the "if 2", with the cursor on the "i", and
                searching forwards, the "endif 2" is found.  When starting on
                the character just before the "if 2", the "endif 1" will be
                found.  That's because the "if 2" will be found first, and
```

```
                then this is considered to be a nested if/endif from "if 2" to
                "endif 2".
                When searching backwards and {end} is more than one character,
                it may be useful to put "\zs" at the end of the pattern, so
                that when the cursor is inside a match with the end it finds
                the matching start.

                Example, to find the "endif" command in a Vim script: >

        :echo searchpair('\<if\>', '\<el\%[seif]\>', '\<en\%[dif]\>', 'W',
                        \ 'getline(".") =~ "^\\s*\""')

<               The cursor must be at or after the "if" for which a match is
                to be found.  Note that single-quote strings are used to avoid
                having to double the backslashes.  The skip expression only
                catches comments at the start of a line, not after a command.
                Also, a word "en" or "if" halfway a line is considered a
                match.
                Another example, to search for the matching "{" of a "}": >

        :echo searchpair('{', '', '}', 'bW')

<               This works when the cursor is at or before the "}" for which a
                match is to be found.  To reject matches that syntax
                highlighting recognized as strings: >

        :echo searchpair('{', '', '}', 'bW',
             \ 'synIDattr(synID(line("."), col("."), 0), "name") =~? "string"')
<
                                                        *searchpairpos()*
searchpairpos({start}, {middle}, {end} [, {flags} [, {skip}
                                [, {stopline} [, {timeout}]]]])
                Same as |searchpair()|, but returns a |List| with the line and
                column position of the match. The first element of the |List|
                is the line number and the second element is the byte index of
                the column position of the match.  If no match is found,
                returns [0, 0]. >

                        :let [lnum,col] = searchpairpos('{', '', '}', 'n')
<
                See |match-parens| for a bigger and more useful example.

searchpos({pattern} [, {flags} [, {stopline} [, {timeout}]]])   *searchpos()*
                Same as |search()|, but returns a |List| with the line and
                column position of the match. The first element of the |List|
                is the line number and the second element is the byte index of
                the column position of the match. If no match is found,
                returns [0, 0].
                Example: >
        :let [lnum, col] = searchpos('mypattern', 'n')

<               When the 'p' flag is given then there is an extra item with
                the sub-pattern match number |search()-sub-match|.  Example: >
        :let [lnum, col, submatch] = searchpos('\(\l\)\|\(\u\)', 'np')
<               In this example "submatch" is 2 when a lowercase letter is
                found |/\l|, 3 when an uppercase letter is found |/\u|.

server2client({clientid}, {string})                     *server2client()*
                Send a reply string to {clientid}.  The most recent {clientid}
                that sent a string can be retrieved with expand("<client>").
                {only available when compiled with the |+clientserver| feature}
                Note:
```

```
                This id has to be stored before the next command can be
                received.  I.e. before returning from the received command and
                before calling any commands that waits for input.
                See also |clientserver|.
                Example: >
                        :echo server2client(expand("<client>"), "HELLO")
<
serverlist()                                        *serverlist()*
                Return a list of available server names, one per line.
                When there are no servers or the information is not available
                an empty string is returned.  See also |clientserver|.
                {only available when compiled with the |+clientserver| feature}
                Example: >
                        :echo serverlist()
<
setbufline({expr}, {lnum}, {text})                      *setbufline()*
                Set line {lnum} to {text} in buffer {expr}.  To insert
                lines use |append()|.

                For the use of {expr}, see |bufname()| above.

                {lnum} is used like with |setline()|.
                This works like |setline()| for the specified buffer.
                On success 0 is returned, on failure 1 is returned.

                If {expr} is not a valid buffer or {lnum} is not valid, an
                error message is given.

setbufvar({expr}, {varname}, {val})                     *setbufvar()*
                Set option or local variable {varname} in buffer {expr} to
                {val}.
                This also works for a global or local window option, but it
                doesn't work for a global or local window variable.
                For a local window option the global value is unchanged.
                For the use of {expr}, see |bufname()| above.
                Note that the variable name without "b:" must be used.
                Examples: >
                        :call setbufvar(1, "&mod", 1)
                        :call setbufvar("todo", "myvar", "foobar")
<               This function is not available in the |sandbox|.

setcharsearch({dict})                                   *setcharsearch()*
                Set the current character search information to {dict},
                which contains one or more of the following entries:

                    char        character which will be used for a subsequent
                                |,| or |;| command; an empty string clears the
                                character search
                    forward     direction of character search; 1 for forward,
                                0 for backward
                    until       type of character search; 1 for a |t| or |T|
                                character search, 0 for an |f| or |F|
                                character search

                This can be useful to save/restore a user's character search
                from a script: >
                        :let prevsearch = getcharsearch()
                        :" Perform a command which clobbers user's search
                        :call setcharsearch(prevsearch)
<               Also see |getcharsearch()|.

setcmdpos({pos})                                        *setcmdpos()*
```

```
                Set the cursor position in the command line to byte position
                {pos}.  The first position is 1.
                Use |getcmdpos()| to obtain the current position.
                Only works while editing the command line, thus you must use
                |c_CTRL-\_e|, |c_CTRL-R_=| or |c_CTRL-R_CTRL-R| with '='.  For
                |c_CTRL-\_e| and |c_CTRL-R_CTRL-R| with '=' the position is
                set after the command line is set to the expression.  For
                |c_CTRL-R_=| it is set after evaluating the expression but
                before inserting the resulting text.
                When the number is too big the cursor is put at the end of the
                line.  A number smaller than one has undefined results.
                Returns 0 when successful, 1 when not editing the command
                line.

setfperm({fname}, {mode})                                *setfperm()* *chmod*
                Set the file permissions for {fname} to {mode}.
                {mode} must be a string with 9 characters.  It is of the form
                "rwxrwxrwx", where each group of "rwx" flags represent, in
                turn, the permissions of the owner of the file, the group the
                file belongs to, and other users.  A '-' character means the
                permission is off, any other character means on.  Multi-byte
                characters are not supported.

                For example "rw-r-----" means read-write for the user,
                readable by the group, not accessible by others.  "xx-x-----"
                would do the same thing.

                Returns non-zero for success, zero for failure.

                To read permissions see |getfperm()|.


setline({lnum}, {text})                                  *setline()*
                Set line {lnum} of the current buffer to {text}.  To insert
                lines use |append()|. To set lines in another buffer use
                |setbufline()|.

                {lnum} is used like with |getline()|.
                When {lnum} is just below the last line the {text} will be
                added as a new line.

                If this succeeds, 0 is returned.  If this fails (most likely
                because {lnum} is invalid) 1 is returned.

                Example: >
                        :call setline(5, strftime("%c"))

<               When {text} is a |List| then line {lnum} and following lines
                will be set to the items in the list.  Example: >
                        :call setline(5, ['aaa', 'bbb', 'ccc'])
<               This is equivalent to: >
                        :for [n, l] in [[5, 'aaa'], [6, 'bbb'], [7, 'ccc']]
                        :  call setline(n, l)
                        :endfor

<               Note: The '[ and '] marks are not set.

setloclist({nr}, {list}[, {action}[, {what}]])           *setloclist()*
                Create or replace or add to the location list for window {nr}.
                {nr} can be the window number or the |window-ID|.
                When {nr} is zero the current window is used.
```

```
                For a location list window, the displayed location list is
                modified.  For an invalid window number {nr}, -1 is returned.
                Otherwise, same as |setqflist()|.
                Also see |location-list|.

                If the optional {what} dictionary argument is supplied, then
                only the items listed in {what} are set. Refer to |setqflist()|
                for the list of supported keys in {what}.

setmatches({list})                                      *setmatches()*
                Restores a list of matches saved by |getmatches()|.  Returns 0
                if successful, otherwise -1.  All current matches are cleared
                before the list is restored.  See example for |getmatches()|.

                                                        *setpos()*
setpos({expr}, {list})
                Set the position for {expr}.  Possible values:
                        .       the cursor
                        'x      mark x

                {list} must be a |List| with four or five numbers:
                    [bufnum, lnum, col, off]
                    [bufnum, lnum, col, off, curswant]

                "bufnum" is the buffer number.  Zero can be used for the
                current buffer.  When setting an uppercase mark "bufnum" is
                used for the mark position.  For other marks it specifies the
                buffer to set the mark in.  You can use the |bufnr()| function
                to turn a file name into a buffer number.
                For setting the cursor and the ' mark "bufnum" is ignored,
                since these are associated with a window, not a buffer.
                Does not change the jumplist.

                "lnum" and "col" are the position in the buffer.  The first
                column is 1.  Use a zero "lnum" to delete a mark.  If "col" is
                smaller than 1 then 1 is used.

                The "off" number is only used when 'virtualedit' is set. Then
                it is the offset in screen columns from the start of the
                character.  E.g., a position within a <Tab> or after the last
                character.

                The "curswant" number is only used when setting the cursor
                position.  It sets the preferred column for when moving the
                cursor vertically.  When the "curswant" number is missing the
                preferred column is not set.  When it is present and setting a
                mark position it is not used.

                Note that for '< and '> changing the line number may result in
                the marks to be effectively be swapped, so that '< is always
                before '>.

                Returns 0 when the position could be set, -1 otherwise.
                An error message is given if {expr} is invalid.

                Also see |getpos()| and |getcurpos()|.

                This does not restore the preferred column for moving
                vertically; if you set the cursor position with this, |j| and
                |k| motions will jump to previous columns!  Use |cursor()| to
                also set the preferred column.  Also see the "curswant" key in
                |winrestview()|.
```

```
setqflist({list} [, {action}[, {what}]])                *setqflist()*
                Create or replace or add to the quickfix list.

                When {what} is not present, use the items in {list}.  Each
                item must be a dictionary.  Non-dictionary items in {list} are
                ignored.  Each dictionary item can contain the following
                entries:

                    bufnr       buffer number; must be the number of a valid
                                buffer
                    filename    name of a file; only used when "bufnr" is not
                                present or it is invalid.
                    lnum        line number in the file
                    pattern     search pattern used to locate the error
                    col         column number
                    vcol        when non-zero: "col" is visual column
                                when zero: "col" is byte index
                    nr          error number
                    text        description of the error
                    type        single-character error type, 'E', 'W', etc.
                    valid       recognized error message

                The "col", "vcol", "nr", "type" and "text" entries are
                optional.  Either "lnum" or "pattern" entry can be used to
                locate a matching error line.
                If the "filename" and "bufnr" entries are not present or
                neither the "lnum" or "pattern" entries are present, then the
                item will not be handled as an error line.
                If both "pattern" and "lnum" are present then "pattern" will
                be used.
                If the "valid" entry is not supplied, then the valid flag is
                set when "bufnr" is a valid buffer or "filename" exists.
                If you supply an empty {list}, the quickfix list will be
                cleared.
                Note that the list is not exactly the same as what
                |getqflist()| returns.

                {action} values:                                    *E927*
                'a'     The items from {list} are added to the existing
                        quickfix list. If there is no existing list, then a
                        new list is created.

                'r'     The items from the current quickfix list are replaced
                        with the items from {list}.  This can also be used to
                        clear the list: >
                                :call setqflist([], 'r')
<
                'f'     All the quickfix lists in the quickfix stack are
                        freed.

                If {action} is not present or is set to ' ', then a new list
                is created. The new quickfix list is added after the current
                quickfix list in the stack and all the following lists are
                freed. To add a new quickfix list at the end of the stack,
                set "nr" in {what} to "$".

                If the optional {what} dictionary argument is supplied, then
                only the items listed in {what} are set. The first {list}
                argument is ignored.  The following items can be specified in
                {what}:
                    context     any Vim type can be stored as a context
```

```
                    efm         errorformat to use when parsing text from
                                "lines". If this is not present, then the
                                'errorformat' option value is used.
                    id          quickfix list identifier |quickfix-ID|
                    items       list of quickfix entries. Same as the {list}
                                argument.
                    lines       use 'errorformat' to parse a list of lines and
                                add the resulting entries to the quickfix list
                                {nr} or {id}.  Only a |List| value is supported.
                    nr          list number in the quickfix stack; zero
                                means the current quickfix list and "$" means
                                the last quickfix list
                    title       quickfix list title text
                Unsupported keys in {what} are ignored.
                If the "nr" item is not present, then the current quickfix list
                is modified. When creating a new quickfix list, "nr" can be
                set to a value one greater than the quickfix stack size.
                When modifying a quickfix list, to guarantee that the correct
                list is modified, "id" should be used instead of "nr" to
                specify the list.

                Examples: >
                   :call setqflist([], 'r', {'title': 'My search'})
                   :call setqflist([], 'r', {'nr': 2, 'title': 'Errors'})
                   :call setqflist([], 'a', {'id':myid, 'lines':["F1:10:L10"]})
<
                Returns zero for success, -1 for failure.

                This function can be used to create a quickfix list
                independent of the 'errorformat' setting.  Use a command like
                `:cc 1` to jump to the first position.


                                                        *setreg()*
setreg({regname}, {value} [, {options}])
                Set the register {regname} to {value}.
                {value} may be any value returned by |getreg()|, including
                a |List|.
                If {options} contains "a" or {regname} is upper case,
                then the value is appended.
                {options} can also contain a register type specification:
                    "c" or "v"        |characterwise| mode
                    "l" or "V"        |linewise| mode
                    "b" or "<CTRL-V>" |blockwise-visual| mode
                If a number immediately follows "b" or "<CTRL-V>" then this is
                used as the width of the selection - if it is not specified
                then the width of the block is set to the number of characters
                in the longest line (counting a <Tab> as 1 character).

                If {options} contains no register settings, then the default
                is to use character mode unless {value} ends in a <NL> for
                string {value} and linewise mode for list {value}. Blockwise
                mode is never selected automatically.
                Returns zero for success, non-zero for failure.

                                                        *E883*
                Note: you may not use |List| containing more than one item to
                      set search and expression registers. Lists containing no
                      items act like empty strings.

                Examples: >
                        :call setreg(v:register, @*)
```

```
                        :call setreg('*', @%, 'ac')
                        :call setreg('a', "1\n2\n3", 'b5')

<               This example shows using the functions to save and restore a
                register: >
                        :let var_a = getreg('a', 1, 1)
                        :let var_amode = getregtype('a')
                            ....
                        :call setreg('a', var_a, var_amode)
<               Note: you may not reliably restore register value
                without using the third argument to |getreg()| as without it
                newlines are represented as newlines AND Nul bytes are
                represented as newlines as well, see |NL-used-for-Nul|.

                You can also change the type of a register by appending
                nothing: >
                        :call setreg('a', '', 'al')

settabvar({tabnr}, {varname}, {val})                    *settabvar()*
                Set tab-local variable {varname} to {val} in tab page {tabnr}.
                |t:var|
                Note that the variable name without "t:" must be used.
                Tabs are numbered starting with one.
                This function is not available in the |sandbox|.

settabwinvar({tabnr}, {winnr}, {varname}, {val})        *settabwinvar()*
                Set option or local variable {varname} in window {winnr} to
                {val}.
                Tabs are numbered starting with one.  For the current tabpage
                use |setwinvar()|.
                {winnr} can be the window number or the |window-ID|.
                When {winnr} is zero the current window is used.
                This also works for a global or local buffer option, but it
                doesn't work for a global or local buffer variable.
                For a local buffer option the global value is unchanged.
                Note that the variable name without "w:" must be used.
                Examples: >
                        :call settabwinvar(1, 1, "&list", 0)
                        :call settabwinvar(3, 2, "myvar", "foobar")
<               This function is not available in the |sandbox|.

setwinvar({nr}, {varname}, {val})                       *setwinvar()*
                Like |settabwinvar()| for the current tab page.
                Examples: >
                        :call setwinvar(1, "&list", 0)
                        :call setwinvar(2, "myvar", "foobar")

sha256({string})                                                *sha256()*
                Returns a String with 64 hex characters, which is the SHA256
                checksum of {string}.
                {only available when compiled with the |+cryptv| feature}

shellescape({string} [, {special}])                     *shellescape()*
                Escape {string} for use as a shell command argument.
                On MS-Windows and MS-DOS, when 'shellslash' is not set, it
                will enclose {string} in double quotes and double all double
                quotes within {string}.
                Otherwise it will enclose {string} in single quotes and
                replace all "'" with "'\''".

                When the {special} argument is present and it's a non-zero
                Number or a non-empty String (|non-zero-arg|), then special
```

```
                items such as "!", "%", "#" and "<cword>" will be preceded by
                a backslash.  This backslash will be removed again by the |:!|
                command.

                The "!" character will be escaped (again with a |non-zero-arg|
                {special}) when 'shell' contains "csh" in the tail.  That is
                because for csh and tcsh "!" is used for history replacement
                even when inside single quotes.

                With a |non-zero-arg| {special} the <NL> character is also
                escaped.  When 'shell' containing "csh" in the tail it's
                escaped a second time.

                Example of use with a |:!| command: >
                    :exe '!dir ' . shellescape(expand('<cfile>'), 1)
<               This results in a directory listing for the file under the
                cursor.  Example of use with |system()|: >
                    :call system("chmod +w -- " . shellescape(expand("%")))
<               See also |::S|.


shiftwidth()                                            *shiftwidth()*
                Returns the effective value of 'shiftwidth'. This is the
                'shiftwidth' value unless it is zero, in which case it is the
                'tabstop' value.  This function was introduced with patch
                7.3.694 in 2012, everybody should have it by now.


simplify({filename})                                    *simplify()*
                Simplify the file name as much as possible without changing
                the meaning.  Shortcuts (on MS-Windows) or symbolic links (on
                Unix) are not resolved.  If the first path component in
                {filename} designates the current directory, this will be
                valid for the result as well.  A trailing path separator is
                not removed either.
                Example: >
                        simplify("./dir/.././/file/") == "./file/"
<               Note: The combination "dir/.." is only removed if "dir" is
                a searchable directory or does not exist.  On Unix, it is also
                removed when "dir" is a symbolic link within the same
                directory.  In order to resolve all the involved symbolic
                links before simplifying the path name, use |resolve()|.


sin({expr})                                             *sin()*
                Return the sine of {expr}, measured in radians, as a |Float|.
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo sin(100)
<                       -0.506366 >
                        :echo sin(-4.01)
<                       0.763301
                {only available when compiled with the |+float| feature}


sinh({expr})                                            *sinh()*
                Return the hyperbolic sine of {expr} as a |Float| in the range
                [-inf, inf].
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo sinh(0.5)
<                       0.521095 >
```

```
                        :echo sinh(-0.9)
<                       -1.026517
                {only available when compiled with the |+float| feature}


sort({list} [, {func} [, {dict}]])                      *sort()* *E702*
                Sort the items in {list} in-place.  Returns {list}.

                If you want a list to remain unmodified make a copy first: >
                        :let sortedlist = sort(copy(mylist))

<               When {func} is omitted, is empty or zero, then sort() uses the
                string representation of each item to sort on.  Numbers sort
                after Strings, |Lists| after Numbers.  For sorting text in the
                current buffer use |:sort|.

                When {func} is given and it is '1' or 'i' then case is
                ignored.

                When {func} is given and it is 'n' then all items will be
                sorted numerical (Implementation detail: This uses the
                strtod() function to parse numbers, Strings, Lists, Dicts and
                Funcrefs will be considered as being 0).

                When {func} is given and it is 'N' then all items will be
                sorted numerical. This is like 'n' but a string containing
                digits will be used as the number they represent.

                When {func} is given and it is 'f' then all items will be
                sorted numerical. All values must be a Number or a Float.

                When {func} is a |Funcref| or a function name, this function
                is called to compare items.  The function is invoked with two
                items as argument and must return zero if they are equal, 1 or
                bigger if the first one sorts after the second one, -1 or
                smaller if the first one sorts before the second one.

                {dict} is for functions with the "dict" attribute.  It will be
                used to set the local variable "self". |Dictionary-function|

                The sort is stable, items which compare equal (as number or as
                string) will keep their relative position. E.g., when sorting
                on numbers, text strings will sort next to each other, in the
                same order as they were originally.

                Also see |uniq()|.

                Example: >
                        func MyCompare(i1, i2)
                           return a:i1 == a:i2 ? 0 : a:i1 > a:i2 ? 1 : -1
                        endfunc
                        let sortedlist = sort(mylist, "MyCompare")
<               A shorter compare version for this specific simple case, which
                ignores overflow: >
                        func MyCompare(i1, i2)
                           return a:i1 - a:i2
                        endfunc
<
                                                        *soundfold()*
soundfold({word})
                Return the sound-folded equivalent of {word}.  Uses the first
                language in 'spelllang' for the current window that supports
```

soundfolding. 'spell' must be set. When no sound folding is
possible the {word} is returned unmodified.
This can be used for making spelling suggestions. Note that
the method can be quite slow.

spellbadword()
spellbadword([{sentence}])
Without argument: The result is the badly spelled word under
or after the cursor. The cursor is moved to the start of the
bad word. When no bad word is found in the cursor line the
result is an empty string and the cursor doesn't move.

With argument: The result is the first word in {sentence} that
is badly spelled. If there are no spelling mistakes the
result is an empty string.

The return value is a list with two items:
- The badly spelled word or an empty string.
- The type of the spelling error:
"bad" spelling mistake
"rare" rare word
"local" word only valid in another region
"caps" word should start with Capital
Example: >
echo spellbadword("the quik brown fox")
< ['quik', 'bad'] ~

The spelling information for the current window is used. The
'spell' option must be set and the value of 'spelllang' is
used.

spellsuggest()
spellsuggest({word} [, {max} [, {capital}]])
Return a |List| with spelling suggestions to replace {word}.
When {max} is given up to this number of suggestions are
returned. Otherwise up to 25 suggestions are returned.

When the {capital} argument is given and it's non-zero only
suggestions with a leading capital will be given. Use this
after a match with 'spellcapcheck'.

{word} can be a badly spelled word followed by other text.
This allows for joining two words that were split. The
suggestions also include the following text, thus you can
replace a line.

{word} may also be a good word. Similar words will then be
returned. {word} itself is not included in the suggestions,
although it may appear capitalized.

The spelling information for the current window is used. The
'spell' option must be set and the values of 'spelllang' and
'spellsuggest' are used.

split({expr} [, {pattern} [, {keepempty}]]) *split()*
Make a |List| out of {expr}. When {pattern} is omitted or
empty each white-separated sequence of characters becomes an
item.
Otherwise the string is split where {pattern} matches,
removing the matched characters. 'ignorecase' is not used
here, add \c to ignore case. |/\c|

When the first or last item is empty it is omitted, unless the
{keepempty} argument is given and it's non-zero.
Other empty items are kept when {pattern} matches at least one
character or when {keepempty} is non-zero.
Example: >
	:let words = split(getline('.'), '\W\+')
< To split a string in individual characters: >
	:for c in split(mystring, '\zs')
< If you want to keep the separator you can also use '\zs' at
the end of the pattern: >
	:echo split('abc:def:ghi', ':\zs')
< ['abc:', 'def:', 'ghi'] ~
Splitting a table where the first element can be empty: >
	:let items = split(line, ':', 1)
< The opposite function is |join()|.

sqrt({expr}) *sqrt()*
Return the non-negative square root of Float {expr} as a
|Float|.
{expr} must evaluate to a |Float| or a |Number|. When {expr}
is negative the result is NaN (Not a Number).
Examples: >
	:echo sqrt(100)
< 10.0 >
	:echo sqrt(-4.01)
< nan
"nan" may be different, it depends on system libraries.
{only available when compiled with the |+float| feature}

str2float({expr}) *str2float()*
Convert String {expr} to a Float. This mostly works the same
as when using a floating point number in an expression, see
|floating-point-format|. But it's a bit more permissive.
E.g., "1e40" is accepted, while in an expression you need to
write "1.0e40".
Text after the number is silently ignored.
The decimal point is always '.', no matter what the locale is
set to. A comma ends the number: "12,345.67" is converted to
12.0. You can strip out thousands separators with
|substitute()|: >
	let f = str2float(substitute(text, ',', '', 'g'))
< {only available when compiled with the |+float| feature}

str2nr({expr} [, {base}]) *str2nr()*
Convert string {expr} to a number.
{base} is the conversion base, it can be 2, 8, 10 or 16.
When {base} is omitted base 10 is used. This also means that
a leading zero doesn't cause octal conversion to be used, as
with the default String to Number conversion.
When {base} is 16 a leading "0x" or "0X" is ignored. With a
different base the result will be zero. Similarly, when
{base} is 8 a leading "0" is ignored, and when {base} is 2 a
leading "0b" or "0B" is ignored.
Text after the number is silently ignored.

strchars({expr} [, {skipcc}]) *strchars()*
The result is a Number, which is the number of characters
in String {expr}.

```
                When {skipcc} is omitted or zero, composing characters are
                counted separately.
                When {skipcc} set to 1, Composing characters are ignored.
                Also see |strlen()|, |strdisplaywidth()| and |strwidth()|.

                {skipcc} is only available after 7.4.755.  For backward
                compatibility, you can define a wrapper function: >
                    if has("patch-7.4.755")
                      function s:strchars(str, skipcc)
                        return strchars(a:str, a:skipcc)
                      endfunction
                    else
                      function s:strchars(str, skipcc)
                        if a:skipcc
                          return strlen(substitute(a:str, ".", "x", "g"))
                        else
                          return strchars(a:str)
                        endif
                      endfunction
                    endif
<
strcharpart({src}, {start}[, {len}])                    *strcharpart()*
                Like |strpart()| but using character index and length instead
                of byte index and length.
                When a character index is used where a character does not
                exist it is assumed to be one character.  For example: >
                        strcharpart('abc', -1, 2)
<               results in 'a'.

strdisplaywidth({expr}[, {col}])                        *strdisplaywidth()*
                The result is a Number, which is the number of display cells
                String {expr} occupies on the screen when it starts at {col}.
                When {col} is omitted zero is used.  Otherwise it is the
                screen column where to start.  This matters for Tab
                characters.
                The option settings of the current window are used.  This
                matters for anything that's displayed differently, such as
                'tabstop' and 'display'.
                When {expr} contains characters with East Asian Width Class
                Ambiguous, this function's return value depends on 'ambiwidth'.
                Also see |strlen()|, |strwidth()| and |strchars()|.

strftime({format} [, {time}])                           *strftime()*
                The result is a String, which is a formatted date and time, as
                specified by the {format} string.  The given {time} is used,
                or the current time if no time is given.  The accepted
                {format} depends on your system, thus this is not portable!
                See the manual page of the C function strftime() for the
                format.  The maximum length of the result is 80 characters.
                See also |localtime()| and |getftime()|.
                The language can be changed with the |:language| command.
                Examples: >
                  :echo strftime("%c")             Sun Apr 27 11:49:23 1997
                  :echo strftime("%Y %b %d %X")    1997 Apr 27 11:53:25
                  :echo strftime("%y%m%d %T")      970427 11:53:55
                  :echo strftime("%H:%M")          11:55
                  :echo strftime("%c", getftime("file.c"))
                                                   Show mod time of file.c.
<               Not available on all systems.  To check use: >
                        :if exists("*strftime")

strgetchar({str}, {index})                              *strgetchar()*
```

```
                Get character {index} from {str}.  This uses a character
                index, not a byte index.  Composing characters are considered
                separate characters here.
                Also see |strcharpart()| and |strchars()|.

stridx({haystack}, {needle} [, {start}])                *stridx()*
                The result is a Number, which gives the byte index in
                {haystack} of the first occurrence of the String {needle}.
                If {start} is specified, the search starts at index {start}.
                This can be used to find a second match: >
                        :let colon1 = stridx(line, ":")
                        :let colon2 = stridx(line, ":", colon1 + 1)
<               The search is done case-sensitive.
                For pattern searches use |match()|.
                -1 is returned if the {needle} does not occur in {haystack}.
                See also |strridx()|.
                Examples: >
                  :echo stridx("An Example", "Example")      3
                  :echo stridx("Starting point", "Start")    0
                  :echo stridx("Starting point", "start")   -1
<                                              *strstr()* *strchr()*
                stridx() works similar to the C function strstr().  When used
                with a single character it works similar to strchr().

                                                        *string()*
string({expr})  Return {expr} converted to a String.  If {expr} is a Number,
                Float, String or a composition of them, then the result can be
                parsed back with |eval()|.
                        {expr} type     result ~
                        String          'string' (single quotes are doubled)
                        Number          123
                        Float           123.123456 or 1.123456e8
                        Funcref         function('name')
                        List            [item, item]
                        Dictionary      {key: value, key: value}

                When a List or Dictionary has a recursive reference it is
                replaced by "[...]" or "{...}".  Using eval() on the result
                will then fail.

                Also see |strtrans()|.

                                                        *strlen()*
strlen({expr})  The result is a Number, which is the length of the String
                {expr} in bytes.
                If the argument is a Number it is first converted to a String.
                For other types an error is given.
                If you want to count the number of multi-byte characters use
                |strchars()|.
                Also see |len()|, |strdisplaywidth()| and |strwidth()|.

strpart({src}, {start}[, {len}])                        *strpart()*
                The result is a String, which is part of {src}, starting from
                byte {start}, with the byte length {len}.
                To count characters instead of bytes use |strcharpart()|.

                When bytes are selected which do not exist, this doesn't
                result in an error, the bytes are simply omitted.
                If {len} is missing, the copy continues from {start} till the
                end of the {src}. >
                        strpart("abcdefg", 3, 2)    == "de"
                        strpart("abcdefg", -2, 4)   == "ab"
```

```
                        strpart("abcdefg", 5, 4)    == "fg"
                        strpart("abcdefg", 3)       == "defg"

<               Note: To get the first character, {start} must be 0.  For
                example, to get three bytes under and after the cursor: >
                        strpart(getline("."), col(".") - 1, 3)
<
strridx({haystack}, {needle} [, {start}])                       *strridx()*
                The result is a Number, which gives the byte index in
                {haystack} of the last occurrence of the String {needle}.
                When {start} is specified, matches beyond this index are
                ignored.  This can be used to find a match before a previous
                match: >
                        :let lastcomma = strridx(line, ",")
                        :let comma2 = strridx(line, ",", lastcomma - 1)
<               The search is done case-sensitive.
                For pattern searches use |match()|.
                -1 is returned if the {needle} does not occur in {haystack}.
                If the {needle} is empty the length of {haystack} is returned.
                See also |stridx()|.  Examples: >
                  :echo strridx("an angry armadillo", "an")          3
<                                                       *strrchr()*
                When used with a single character it works similar to the C
                function strrchr().

strtrans({expr})                                        *strtrans()*
                The result is a String, which is {expr} with all unprintable
                characters translated into printable characters |'isprint'|.
                Like they are shown in a window.  Example: >
                        echo strtrans(@a)
<               This displays a newline in register a as "^@" instead of
                starting a new line.

strwidth({expr})                                        *strwidth()*
                The result is a Number, which is the number of display cells
                String {expr} occupies.  A Tab character is counted as one
                cell, alternatively use |strdisplaywidth()|.
                When {expr} contains characters with East Asian Width Class
                Ambiguous, this function's return value depends on 'ambiwidth'.
                Also see |strlen()|, |strdisplaywidth()| and |strchars()|.

submatch({nr}[, {list}])                        *submatch()* *E935*
                Only for an expression in a |:substitute| command or
                substitute() function.
                Returns the {nr}'th submatch of the matched text.  When {nr}
                is 0 the whole matched text is returned.
                Note that a NL in the string can stand for a line break of a
                multi-line match or a NUL character in the text.
                Also see |sub-replace-expression|.

                If {list} is present and non-zero then submatch() returns
                a list of strings, similar to |getline()| with two arguments.
                NL characters in the text represent NUL characters in the
                text.
                Only returns more than one item for |:substitute|, inside
                |substitute()| this list will always contain one or zero
                items, since there are no real line breaks.

                When substitute() is used recursively only the submatches in
                the current (deepest) call can be obtained.

                Example: >
```

```
                        :s/\d\+/\=submatch(0) + 1/
<               This finds the first number in the line and adds one to it.
                A line break is included as a newline character.

substitute({expr}, {pat}, {sub}, {flags})               *substitute()*
                The result is a String, which is a copy of {expr}, in which
                the first match of {pat} is replaced with {sub}.
                When {flags} is "g", all matches of {pat} in {expr} are
                replaced.  Otherwise {flags} should be "".

                This works like the ":substitute" command (without any flags).
                But the matching with {pat} is always done like the 'magic'
                option is set and 'cpoptions' is empty (to make scripts
                portable).  'ignorecase' is still relevant, use |/\c| or |/\C|
                if you want to ignore or match case and ignore 'ignorecase'.
                'smartcase' is not used.  See |string-match| for how {pat} is
                used.

                A "~" in {sub} is not replaced with the previous {sub}.
                Note that some codes in {sub} have a special meaning
                |sub-replace-special|.  For example, to replace something with
                "\n" (two characters), use "\\\\n" or '\\n'.

                When {pat} does not match in {expr}, {expr} is returned
                unmodified.

                Example: >
                   :let &path = substitute(&path, ",\\=[^,]*$", "", "")
<               This removes the last component of the 'path' option. >
                   :echo substitute("testing", ".*", "\\U\\0", "")
<               results in "TESTING".

                When {sub} starts with "\=", the remainder is interpreted as
                an expression. See |sub-replace-expression|.  Example: >
                   :echo substitute(s, '%\(\x\x\)',
                           \ '\=nr2char("0x" . submatch(1))', 'g')

<               When {sub} is a Funcref that function is called, with one
                optional argument.  Example: >
                   :echo substitute(s, '%\(\x\x\)', SubNr, 'g')
<               The optional argument is a list which contains the whole
                matched string and up to nine submatches, like what
                |submatch()| returns.  Example: >
                   :echo substitute(s, '%\(\x\x\)', {m -> '0x' . m[1]}, 'g')

synID({lnum}, {col}, {trans})                           *synID()*
                The result is a Number, which is the syntax ID at the position
                {lnum} and {col} in the current window.
                The syntax ID can be used with |synIDattr()| and
                |synIDtrans()| to obtain syntax information about text.

                {col} is 1 for the leftmost column, {lnum} is 1 for the first
                line.  'synmaxcol' applies, in a longer line zero is returned.
                Note that when the position is after the last character,
                that's where the cursor can be in Insert mode, synID() returns
                zero.

                When {trans} is |TRUE|, transparent items are reduced to the
                item that they reveal.  This is useful when wanting to know
                the effective color.  When {trans} is |FALSE|, the transparent
                item is returned.  This is useful when wanting to know which
                syntax item is effective (e.g. inside parens).
```

```
                Warning: This function can be very slow.  Best speed is
                obtained by going through the file in forward direction.

                Example (echoes the name of the syntax item under the cursor): >
                        :echo synIDattr(synID(line("."), col("."), 1), "name")
<

synIDattr({synID}, {what} [, {mode}])                   *synIDattr()*
                The result is a String, which is the {what} attribute of
                syntax ID {synID}.  This can be used to obtain information
                about a syntax item.
                {mode} can be "gui", "cterm" or "term", to get the attributes
                for that mode.  When {mode} is omitted, or an invalid value is
                used, the attributes for the currently active highlighting are
                used (GUI, cterm or term).
                Use synIDtrans() to follow linked highlight groups.
                {what}          result
                "name"          the name of the syntax item
                "fg"            foreground color (GUI: color name used to set
                                the color, cterm: color number as a string,
                                term: empty string)
                "bg"            background color (as with "fg")
                "font"          font name (only available in the GUI)
                                |highlight-font|
                "sp"            special color (as with "fg") |highlight-guisp|
                "fg#"           like "fg", but for the GUI and the GUI is
                                running the name in "#RRGGBB" form
                "bg#"           like "fg#" for "bg"
                "sp#"           like "fg#" for "sp"
                "bold"          "1" if bold
                "italic"        "1" if italic
                "reverse"       "1" if reverse
                "inverse"       "1" if inverse (= reverse)
                "standout"      "1" if standout
                "underline"     "1" if underlined
                "undercurl"     "1" if undercurled
                "strike"        "1" if strikethrough

                Example (echoes the color of the syntax item under the
                cursor): >
        :echo synIDattr(synIDtrans(synID(line("."), col("."), 1)), "fg")
<
synIDtrans({synID})                                     *synIDtrans()*
                The result is a Number, which is the translated syntax ID of
                {synID}.  This is the syntax group ID of what is being used to
                highlight the character.  Highlight links given with
                ":highlight link" are followed.

synconcealed({lnum}, {col})                             *synconcealed()*
                The result is a List with currently three items:
                1. The first item in the list is 0 if the character at the
                   position {lnum} and {col} is not part of a concealable
                   region, 1 if it is.
                2. The second item in the list is a string. If the first item
                   is 1, the second item contains the text which will be
                   displayed in place of the concealed text, depending on the
                   current setting of 'conceallevel' and 'listchars'.
                3. The third and final item in the list is a number
                   representing the specific syntax region matched in the
                   line. When the character is not concealed the value is
                   zero. This allows detection of the beginning of a new
                   concealable region if there are two consecutive regions
```

```
                  with the same replacement character.  For an example, if
                  the text is "123456" and both "23" and "45" are concealed
                  and replace by the character "X", then:
                       call                    returns ~
                       synconcealed(lnum, 1)   [0, '', 0]
                       synconcealed(lnum, 2)   [1, 'X', 1]
                       synconcealed(lnum, 3)   [1, 'X', 1]
                       synconcealed(lnum, 4)   [1, 'X', 2]
                       synconcealed(lnum, 5)   [1, 'X', 2]
                       synconcealed(lnum, 6)   [0, '', 0]


synstack({lnum}, {col})                                 *synstack()*
               Return a |List|, which is the stack of syntax items at the
               position {lnum} and {col} in the current window.  Each item in
               the List is an ID like what |synID()| returns.
               The first item in the List is the outer region, following are
               items contained in that one.  The last one is what |synID()|
               returns, unless not the whole item is highlighted or it is a
               transparent item.
               This function is useful for debugging a syntax file.
               Example that shows the syntax stack under the cursor: >
                       for id in synstack(line("."), col("."))
                          echo synIDattr(id, "name")
                       endfor
<              When the position specified with {lnum} and {col} is invalid
               nothing is returned.  The position just after the last
               character in a line and the first column in an empty line are
               valid positions.

system({expr} [, {input}])                              *system()* *E677*
               Get the output of the shell command {expr} as a string.  See
               |systemlist()| to get the output as a List.

               When {input} is given and is a string this string is written
               to a file and passed as stdin to the command.  The string is
               written as-is, you need to take care of using the correct line
               separators yourself.
               If {input} is given and is a |List| it is written to the file
               in a way |writefile()| does with {binary} set to "b" (i.e.
               with a newline between each list item with newlines inside
               list items converted to NULs).
               When {input} is given and is a number that is a valid id for
               an existing buffer then the content of the buffer is written
               to the file line by line, each line terminated by a NL and
               NULs characters where the text has a NL.

               Pipes are not used, the 'shelltemp' option is not used.

               When prepended by |:silent| the terminal will not be set to
               cooked mode.  This is meant to be used for commands that do
               not need the user to type.  It avoids stray characters showing
               up on the screen which require |CTRL-L| to remove. >
                       :silent let f = system('ls *.vim')
<
               Note: Use |shellescape()| or |::S| with |expand()| or
               |fnamemodify()| to escape special characters in a command
               argument.  Newlines in {expr} may cause the command to fail.
               The characters in 'shellquote' and 'shellxquote' may also
               cause trouble.
               This is not to be used for interactive commands.
```

```
                The result is a String.  Example: >
                    :let files = system("ls " .  shellescape(expand('%:h')))
                    :let files = system('ls ' . expand('%:h:S'))

<               To make the result more system-independent, the shell output
                is filtered to replace <CR> with <NL> for Macintosh, and
                <CR><NL> with <NL> for DOS-like systems.
                To avoid the string being truncated at a NUL, all NUL
                characters are replaced with SOH (0x01).

                The command executed is constructed using several options:
        'shell' 'shellcmdflag' 'shellxquote' {expr} 'shellredir' {tmp} 'shellxquote'
                ({tmp} is an automatically generated file name).
                For Unix and OS/2 braces are put around {expr} to allow for
                concatenated commands.

                The command will be executed in "cooked" mode, so that a
                CTRL-C will interrupt the command (on Unix at least).

                The resulting error code can be found in |v:shell_error|.
                This function will fail in |restricted-mode|.

                Note that any wrong value in the options mentioned above may
                make the function fail.  It has also been reported to fail
                when using a security agent application.
                Unlike ":!cmd" there is no automatic check for changed files.
                Use |:checktime| to force a check.


systemlist({expr} [, {input}])                          *systemlist()*
                Same as |system()|, but returns a |List| with lines (parts of
                output separated by NL) with NULs transformed into NLs. Output
                is the same as |readfile()| will output with {binary} argument
                set to "b".  Note that on MS-Windows you may get trailing CR
                characters.

                Returns an empty string on error.


tabpagebuflist([{arg}])                                 *tabpagebuflist()*
                The result is a |List|, where each item is the number of the
                buffer associated with each window in the current tab page.
                {arg} specifies the number of the tab page to be used. When
                omitted the current tab page is used.
                When {arg} is invalid the number zero is returned.
                To get a list of all buffers in all tabs use this: >
                        let buflist = []
                        for i in range(tabpagenr('$'))
                           call extend(buflist, tabpagebuflist(i + 1))
                        endfor
<               Note that a buffer may appear in more than one window.


tabpagenr([{arg}])                                      *tabpagenr()*
                The result is a Number, which is the number of the current
                tab page.  The first tab page has number 1.
                When the optional argument is "$", the number of the last tab
                page is returned (the tab page count).
                The number can be used with the |:tab| command.


tabpagewinnr({tabarg} [, {arg}])                        *tabpagewinnr()*
```

```
                Like |winnr()| but for tab page {tabarg}.
                {tabarg} specifies the number of tab page to be used.
                {arg} is used like with |winnr()|:
                - When omitted the current window number is returned.  This is
                  the window which will be used when going to this tab page.
                - When "$" the number of windows is returned.
                - When "#" the previous window nr is returned.
                Useful examples: >
                    tabpagewinnr(1)         " current window of tab page 1
                    tabpagewinnr(4, '$')    " number of windows in tab page 4
<               When {tabarg} is invalid zero is returned.

                                                        *tagfiles()*
tagfiles()      Returns a |List| with the file names used to search for tags
                for the current buffer.  This is the 'tags' option expanded.


taglist({expr}[, {filename}])                           *taglist()*
                Returns a list of tags matching the regular expression {expr}.

                If {filename} is passed it is used to prioritize the results
                in the same way that |:tselect| does. See |tag-priority|.
                {filename} should be the full path of the file.

                Each list item is a dictionary with at least the following
                entries:
                        name            Name of the tag.
                        filename        Name of the file where the tag is
                                        defined.  It is either relative to the
                                        current directory or a full path.
                        cmd             Ex command used to locate the tag in
                                        the file.
                        kind            Type of the tag.  The value for this
                                        entry depends on the language specific
                                        kind values.  Only available when
                                        using a tags file generated by
                                        Exuberant ctags or hdrtag.
                        static          A file specific tag.  Refer to
                                        |static-tag| for more information.
                More entries may be present, depending on the content of the
                tags file: access, implementation, inherits and signature.
                Refer to the ctags documentation for information about these
                fields.  For C code the fields "struct", "class" and "enum"
                may appear, they give the name of the entity the tag is
                contained in.

                The ex-command "cmd" can be either an ex search pattern, a
                line number or a line number followed by a byte number.

                If there are no matching tags, then an empty list is returned.

                To get an exact tag match, the anchors '^' and '$' should be
                used in {expr}.  This also make the function work faster.
                Refer to |tag-regexp| for more information about the tag
                search regular expression pattern.

                Refer to |'tags'| for information about how the tags file is
                located by Vim. Refer to |tags-file-format| for the format of
                the tags file generated by the different ctags tools.

tan({expr})                                             *tan()*
                Return the tangent of {expr}, measured in radians, as a |Float|
```

```
                in the range [-inf, inf].
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo tan(10)
<                       0.648361 >
                        :echo tan(-4.01)
<                       -1.181502
                {only available when compiled with the |+float| feature}


tanh({expr})                                            *tanh()*
                Return the hyperbolic tangent of {expr} as a |Float| in the
                range [-1, 1].
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        :echo tanh(0.5)
<                       0.462117 >
                        :echo tanh(-1)
<                       -0.761594
                {only available when compiled with the |+float| feature}


tempname()                                      *tempname()* *temp-file-name*
                The result is a String, which is the name of a file that
                doesn't exist.  It can be used for a temporary file.  The name
                is different for at least 26 consecutive calls.  Example: >
                        :let tmpfile = tempname()
                        :exe "redir > " . tmpfile
<               For Unix, the file will be in a private directory |tempfile|.
                For MS-Windows forward slashes are used when the 'shellslash'
                option is set or when 'shellcmdflag' starts with '-'.

term_getaltscreen({buf})                                *term_getaltscreen()*
                Returns 1 if the terminal of {buf} is using the alternate
                screen.
                {buf} is used as with |term_getsize()|.
                {only available when compiled with the |+terminal| feature}

term_getattr({attr}, {what})                            *term_getattr()*
                Given {attr}, a value returned by term_scrape() in the "attr"
                item, return whether {what} is on.  {what} can be one of:
                        bold
                        italic
                        underline
                        strike
                        reverse
                {only available when compiled with the |+terminal| feature}

term_getcursor({buf})                                   *term_getcursor()*
                Get the cursor position of terminal {buf}. Returns a list with
                two numbers and a dictionary: [row, col, dict].

                "row" and "col" are one based, the first screen cell is row
                1, column 1.  This is the cursor position of the terminal
                itself, not of the Vim window.

                "dict" can have these members:
                   "visible"    one when the cursor is visible, zero when it
                                is hidden.
                   "blink"      one when the cursor is visible, zero when it
                                is hidden.
                   "shape"      1 for a block cursor, 2 for underline and 3
```

```
                                for a vertical bar.

                {buf} must be the buffer number of a terminal window. If the
                buffer does not exist or is not a terminal window, an empty
                list is returned.
                {only available when compiled with the |+terminal| feature}

term_getjob({buf})                                      *term_getjob()*
                Get the Job associated with terminal window {buf}.
                {buf} is used as with |term_getsize()|.
                Returns |v:null| when there is no job.
                {only available when compiled with the |+terminal| feature}

term_getline({buf}, {row})                              *term_getline()*
                Get a line of text from the terminal window of {buf}.
                {buf} is used as with |term_getsize()|.

                The first line has {row} one.  When {row} is "." the cursor
                line is used.  When {row} is invalid an empty string is
                returned.
                {only available when compiled with the |+terminal| feature}

term_getscrolled({buf})                                 *term_getscrolled()*
                Return the number of lines that scrolled to above the top of
                terminal {buf}.  This is the offset between the row number
                used for |term_getline()| and |getline()|, so that: >
                        term_getline(buf, N)
<               is equal to: >
                        `getline(N + term_getscrolled(buf))
<               (if that line exists).

                {buf} is used as with |term_getsize()|.
                {only available when compiled with the |+terminal| feature}

term_getsize({buf})                                     *term_getsize()*
                Get the size of terminal {buf}. Returns a list with two
                numbers: [rows, cols].  This is the size of the terminal, not
                the window containing the terminal.

                {buf} must be the buffer number of a terminal window.  Use an
                empty string for the current buffer.  If the buffer does not
                exist or is not a terminal window, an empty list is returned.
                {only available when compiled with the |+terminal| feature}

term_getstatus({buf})                                   *term_getstatus()*
                Get the status of terminal {buf}. This returns a comma
                separated list of these items:
                        running         job is running
                        finished        job has finished
                        normal          in Terminal-Normal mode
                One of "running" or "finished" is always present.

                {buf} must be the buffer number of a terminal window. If the
                buffer does not exist or is not a terminal window, an empty
                string is returned.
                {only available when compiled with the |+terminal| feature}

term_gettitle({buf})                                    *term_gettitle()*
                Get the title of terminal {buf}. This is the title that the
                job in the terminal has set.

                {buf} must be the buffer number of a terminal window. If the
```

```
                buffer does not exist or is not a terminal window, an empty
                string is returned.
                {only available when compiled with the |+terminal| feature}

term_gettty({buf} [, {input}])                          *term_gettty()*
                Get the name of the controlling terminal associated with
                terminal window {buf}.  {buf} is used as with |term_getsize()|.

                When {input} is omitted or 0, return the name for writing
                (stdout). When {input} is 1 return the name for reading
                (stdin). On UNIX, both return same name.
                {only available when compiled with the |+terminal| feature}

term_list()                                             *term_list()*
                Return a list with the buffer numbers of all buffers for
                terminal windows.
                {only available when compiled with the |+terminal| feature}

term_scrape({buf}, {row})                               *term_scrape()*
                Get the contents of {row} of terminal screen of {buf}.
                For {buf} see |term_getsize()|.

                The first line has {row} one.  When {row} is "." the cursor
                line is used.  When {row} is invalid an empty string is
                returned.

                Return a List containing a Dict for each screen cell:
                    "chars"     character(s) at the cell
                    "fg"        foreground color as #rrggbb
                    "bg"        background color as #rrggbb
                    "attr"      attributes of the cell, use |term_getattr()|
                                to get the individual flags
                    "width"     cell width: 1 or 2
                {only available when compiled with the |+terminal| feature}

term_sendkeys({buf}, {keys})                            *term_sendkeys()*
                Send keystrokes {keys} to terminal {buf}.
                {buf} is used as with |term_getsize()|.

                {keys} are translated as key sequences. For example, "\<c-x>"
                means the character CTRL-X.
                {only available when compiled with the |+terminal| feature}

term_setsize({buf}, {expr})                             *term_setsize()*
                Not implemented yet.
                {only available when compiled with the |+terminal| feature}

term_start({cmd}, {options})                            *term_start()*
                Open a terminal window and run {cmd} in it.

                Returns the buffer number of the terminal window.  If {cmd}
                cannot be executed the window does open and shows an error
                message.
                If opening the window fails zero is returned.

                {options} are similar to what is used for |job_start()|, see
                |job-options|.  However, not all options can be used.  These
                are supported:
                   all timeout options
                   "stoponexit"
                   "callback", "out_cb", "err_cb"
                   "exit_cb", "close_cb"
```

```
                   "in_io", "in_top", "in_bot", "in_name", "in_buf"
                   "out_io", "out_name", "out_buf", "out_modifiable", "out_msg"
                   "err_io", "err_name", "err_buf", "err_modifiable", "err_msg"
                However, at least one of stdin, stdout or stderr must be
                connected to the terminal.  When I/O is connected to the
                terminal then the callback function for that part is not used.

                There are extra options:
                   "term_name"       name to use for the buffer name, instead
                                     of the command name.
                   "term_rows"       vertical size to use for the terminal,
                                     instead of using 'termsize'
                   "term_cols"       horizontal size to use for the terminal,
                                     instead of using 'termsize'
                   "vertical"        split the window vertically
                   "curwin"          use the current window, do not split the
                                     window; fails if the current buffer
                                     cannot be |abandon|ed
                   "hidden"          do not open a window
                   "term_finish"     What to do when the job is finished:
                                        "close": close any windows
                                        "open": open window if needed
                                     Note that "open" can be interruptive.
                                     See |term++close| and |term++open|.
                   "term_opencmd"    command to use for opening the window when
                                     "open" is used for "term_finish"; must
                                     have "%d" where the buffer number goes,
                                     e.g. "10split|buffer %d"; when not
                                     specified "botright sbuf %d" is used
                   "eof_chars"       Text to send after all buffer lines were
                                     written to the terminal.  When not set
                                     CTRL-D is used on MS-Windows. For Python
                                     use CTRL-Z or "exit()". For a shell use
                                     "exit".  A CR is always added.

                {only available when compiled with the |+terminal| feature}

term_wait({buf} [, {time}])                                    *term_wait()*
                Wait for pending updates of {buf} to be handled.
                {buf} is used as with |term_getsize()|.
                {time} is how long to wait for updates to arrive in msec.  If
                not set then 10 msec will be used.
                {only available when compiled with the |+terminal| feature}

test_alloc_fail({id}, {countdown}, {repeat})            *test_alloc_fail()*
                This is for testing: If the memory allocation with {id} is
                called, then decrement {countdown}, and when it reaches zero
                let memory allocation fail {repeat} times.  When {repeat} is
                smaller than one it fails one time.

test_autochdir()                                        *test_autochdir()*
                Set a flag to enable the effect of 'autochdir' before Vim
                startup has finished.

test_feedinput({string})                                *test_feedinput()*
                Characters in {string} are queued for processing as if they
                were typed by the user. This uses a low level input buffer.
                This function works only when with |+unix| or GUI is running.

test_garbagecollect_now()                       *test_garbagecollect_now()*
                Like garbagecollect(), but executed right away.  This must
                only be called directly to avoid any structure to exist
```

```
                internally, and |v:testing| must have been set before calling
                any function.

test_ignore_error({expr})                          *test_ignore_error()*
                Ignore any error containing {expr}.  A normal message is given
                instead.
                This is only meant to be used in tests, where catching the
                error with try/catch cannot be used (because it skips over
                following code).
                {expr} is used literally, not as a pattern.
                There is currently no way to revert this.

test_null_channel()                                        *test_null_channel()*
                Return a Channel that is null. Only useful for testing.
                {only available when compiled with the +channel feature}

test_null_dict()                                           *test_null_dict()*
                Return a Dict that is null. Only useful for testing.

test_null_job()                                            *test_null_job()*
                Return a Job that is null. Only useful for testing.
                {only available when compiled with the +job feature}

test_null_list()                                           *test_null_list()*
                Return a List that is null. Only useful for testing.

test_null_partial()                                        *test_null_partial()*
                Return a Partial that is null. Only useful for testing.

test_null_string()                                         *test_null_string()*
                Return a String that is null. Only useful for testing.

test_override({name}, {val})                               *test_override()*
                Overrides certain parts of Vims internal processing to be able
                to run tests. Only to be used for testing Vim!
                The override is enabled when {val} is non-zero and removed
                when {val} is zero.
                Current supported values for name are:

                name         effect when {val} is non-zero ~
                redraw       disable the redrawing() function
                char_avail   disable the char_avail() function
                starting     reset the "starting" variable, see below
                ALL          clear all overrides ({val} is not used)

                "starting" is to be used when a test should behave like
                startup was done.  Since the tests are run by sourcing a
                script the "starting" variable is non-zero. This is usually a
                good thing (tests run faster), but sometimes changes behavior
                in a way that the test doesn't work properly.
                When using: >
                        call test_override('starting', 1)
<               The value of "starting" is saved.  It is restored by: >
                        call test_override('starting', 0)

test_settime({expr})                                       *test_settime()*
                Set the time Vim uses internally.  Currently only used for
                timestamps in the history, as they are used in viminfo, and
                for undo.
                Using a value of 1 makes Vim not sleep after a warning or
                error message.
                {expr} must evaluate to a number.  When the value is zero the
```

```
                normal behavior is restored.

                                                        *timer_info()*
timer_info([{id}])
                Return a list with information about timers.
                When {id} is given only information about this timer is
                returned.  When timer {id} does not exist an empty list is
                returned.
                When {id} is omitted information about all timers is returned.

                For each timer the information is stored in a Dictionary with
                these items:
                    "id"            the timer ID
                    "time"          time the timer was started with
                    "remaining"     time until the timer fires
                    "repeat"        number of times the timer will still fire;
                                    -1 means forever
                    "callback"      the callback
                    "paused"        1 if the timer is paused, 0 otherwise

                {only available when compiled with the |+timers| feature}

timer_pause({timer}, {paused})                          *timer_pause()*
                Pause or unpause a timer.  A paused timer does not invoke its
                callback when its time expires.  Unpausing a timer may cause
                the callback to be invoked almost immediately if enough time
                has passed.

                Pausing a timer is useful to avoid the callback to be called
                for a short time.

                If {paused} evaluates to a non-zero Number or a non-empty
                String, then the timer is paused, otherwise it is unpaused.
                See |non-zero-arg|.

                {only available when compiled with the |+timers| feature}

                                              *timer_start()* *timer* *timers*
timer_start({time}, {callback} [, {options}])
                Create a timer and return the timer ID.

                {time} is the waiting time in milliseconds. This is the
                minimum time before invoking the callback.  When the system is
                busy or Vim is not waiting for input the time will be longer.

                {callback} is the function to call.  It can be the name of a
                function or a |Funcref|.  It is called with one argument, which
                is the timer ID.  The callback is only invoked when Vim is
                waiting for input.

                {options} is a dictionary.  Supported entries:
                   "repeat"     Number of times to repeat calling the
                                callback.  -1 means forever.  When not present
                                the callback will be called once.
                                If the timer causes an error three times in a
                                row the repeat is cancelled.  This avoids that
                                Vim becomes unusable because of all the error
                                messages.

                Example: >
                        func MyHandler(timer)
                          echo 'Handler called'
```

```
                        endfunc
                        let timer = timer_start(500, 'MyHandler',
                                \ {'repeat': 3})
<               This will invoke MyHandler() three times at 500 msec
                intervals.

                {only available when compiled with the |+timers| feature}

timer_stop({timer})                                     *timer_stop()*
                Stop a timer.  The timer callback will no longer be invoked.
                {timer} is an ID returned by timer_start(), thus it must be a
                Number.  If {timer} does not exist there is no error.

                {only available when compiled with the |+timers| feature}

timer_stopall()                                         *timer_stopall()*
                Stop all timers.  The timer callbacks will no longer be
                invoked.  Useful if some timers is misbehaving.  If there are
                no timers there is no error.

                {only available when compiled with the |+timers| feature}

tolower({expr})                                         *tolower()*
                The result is a copy of the String given, with all uppercase
                characters turned into lowercase (just like applying |gu| to
                the string).

toupper({expr})                                         *toupper()*
                The result is a copy of the String given, with all lowercase
                characters turned into uppercase (just like applying |gU| to
                the string).

tr({src}, {fromstr}, {tostr})                           *tr()*
                The result is a copy of the {src} string with all characters
                which appear in {fromstr} replaced by the character in that
                position in the {tostr} string.  Thus the first character in
                {fromstr} is translated into the first character in {tostr}
                and so on.  Exactly like the unix "tr" command.
                This code also deals with multibyte characters properly.

                Examples: >
                        echo tr("hello there", "ht", "HT")
<               returns "Hello THere" >
                        echo tr("<blob>", "<>", "{}")
<               returns "{blob}"

trunc({expr})                                                   *trunc()*
                Return the largest integral value with magnitude less than or
                equal to {expr} as a |Float| (truncate towards zero).
                {expr} must evaluate to a |Float| or a |Number|.
                Examples: >
                        echo trunc(1.456)
<                       1.0  >
                        echo trunc(-5.456)
<                       -5.0  >
                        echo trunc(4.0)
<                       4.0
                {only available when compiled with the |+float| feature}

                                                        *type()*
type({expr})    The result is a Number representing the type of {expr}.
                Instead of using the number directly, it is better to use the
```

```
                v:t_ variable that has the value:
                        Number:     0  |v:t_number|
                        String:     1  |v:t_string|
                        Funcref:    2  |v:t_func|
                        List:       3  |v:t_list|
                        Dictionary: 4  |v:t_dict|
                        Float:      5  |v:t_float|
                        Boolean:    6  |v:t_bool| (v:false and v:true)
                        None        7  |v:t_none| (v:null and v:none)
                        Job         8  |v:t_job|
                        Channel     9  |v:t_channel|
                For backward compatibility, this method can be used: >
                        :if type(myvar) == type(0)
                        :if type(myvar) == type("")
                        :if type(myvar) == type(function("tr"))
                        :if type(myvar) == type([])
                        :if type(myvar) == type({})
                        :if type(myvar) == type(0.0)
                        :if type(myvar) == type(v:false)
                        :if type(myvar) == type(v:none)
<               To check if the v:t_ variables exist use this: >
                        :if exists('v:t_number')

undofile({name})                                            *undofile()*
                Return the name of the undo file that would be used for a file
                with name {name} when writing.  This uses the 'undodir'
                option, finding directories that exist.  It does not check if
                the undo file exists.
                {name} is always expanded to the full path, since that is what
                is used internally.
                If {name} is empty undofile() returns an empty string, since a
                buffer without a file name will not write an undo file.
                Useful in combination with |:wundo| and |:rundo|.
                When compiled without the +persistent_undo option this always
                returns an empty string.

undotree()                                                  *undotree()*
                Return the current state of the undo tree in a dictionary with
                the following items:
                  "seq_last"    The highest undo sequence number used.
                  "seq_cur"     The sequence number of the current position in
                                the undo tree.  This differs from "seq_last"
                                when some changes were undone.
                  "time_cur"    Time last used for |:earlier| and related
                                commands.  Use |strftime()| to convert to
                                something readable.
                  "save_last"   Number of the last file write.  Zero when no
                                write yet.
                  "save_cur"    Number of the current position in the undo
                                tree.
                  "synced"      Non-zero when the last undo block was synced.
                                This happens when waiting from input from the
                                user.  See |undo-blocks|.
                  "entries"     A list of dictionaries with information about
                                undo blocks.

                The first item in the "entries" list is the oldest undo item.
                Each List item is a Dictionary with these items:
                  "seq"         Undo sequence number.  Same as what appears in
                                |:undolist|.
                  "time"        Timestamp when the change happened.  Use
                                |strftime()| to convert to something readable.
```

```
                  "newhead"     Only appears in the item that is the last one
                                that was added.  This marks the last change
                                and where further changes will be added.
                  "curhead"     Only appears in the item that is the last one
                                that was undone.  This marks the current
                                position in the undo tree, the block that will
                                be used by a redo command.  When nothing was
                                undone after the last change this item will
                                not appear anywhere.
                  "save"        Only appears on the last block before a file
                                write.  The number is the write count.  The
                                first write has number 1, the last one the
                                "save_last" mentioned above.
                  "alt"         Alternate entry.  This is again a List of undo
                                blocks.  Each item may again have an "alt"
                                item.

uniq({list} [, {func} [, {dict}]])                      *uniq()* *E882*
                Remove second and succeeding copies of repeated adjacent
                {list} items in-place.  Returns {list}.  If you want a list
                to remain unmodified make a copy first: >
                        :let newlist = uniq(copy(mylist))
<               The default compare function uses the string representation of
                each item.  For the use of {func} and {dict} see |sort()|.

values({dict})                                          *values()*
                Return a |List| with all the values of {dict}.  The |List| is
                in arbitrary order.


virtcol({expr})                                         *virtcol()*
                The result is a Number, which is the screen column of the file
                position given with {expr}.  That is, the last screen position
                occupied by the character at that position, when the screen
                would be of unlimited width.  When there is a <Tab> at the
                position, the returned Number will be the column at the end of
                the <Tab>.  For example, for a <Tab> in column 1, with 'ts'
                set to 8, it returns 8. |conceal| is ignored.
                For the byte position use |col()|.
                For the use of {expr} see |col()|.
                When 'virtualedit' is used {expr} can be [lnum, col, off], where
                "off" is the offset in screen columns from the start of the
                character.  E.g., a position within a <Tab> or after the last
                character.  When "off" is omitted zero is used.
                When Virtual editing is active in the current mode, a position
                beyond the end of the line can be returned. |'virtualedit'|
                The accepted positions are:
                    .       the cursor position
                    $       the end of the cursor line (the result is the
                            number of displayed characters in the cursor line
                            plus one)
                    'x      position of mark x (if the mark is not set, 0 is
                            returned)
                    v       In Visual mode: the start of the Visual area (the
                            cursor is the end).  When not in Visual mode
                            returns the cursor position.  Differs from |'<| in
                            that it's updated right away.
                Note that only marks in the current file can be used.
                Examples: >
  virtcol(".")     with text "foo^Lbar", with cursor on the "^L", returns 5
  virtcol("$")     with text "foo^Lbar", returns 9
  virtcol("'t")    with text "    there", with 't at 'h', returns 6
```

```
<               The first column is 1.  0 is returned for an error.
                A more advanced example that echoes the maximum length of
                all lines: >
                    echo max(map(range(1, line('$')), "virtcol([v:val, '$'])"))


visualmode([expr])                                              *visualmode()*
                The result is a String, which describes the last Visual mode
                used in the current buffer.  Initially it returns an empty
                string, but once Visual mode has been used, it returns "v",
                "V", or "<CTRL-V>" (a single CTRL-V character) for
                character-wise, line-wise, or block-wise Visual mode
                respectively.
                Example: >
                        :exe "normal " . visualmode()
<               This enters the same Visual mode as before.  It is also useful
                in scripts if you wish to act differently depending on the
                Visual mode that was used.
                If Visual mode is active, use |mode()| to get the Visual mode
                (e.g., in a |:vmap|).
                If [expr] is supplied and it evaluates to a non-zero Number or
                a non-empty String, then the Visual mode will be cleared and
                the old value is returned.  See |non-zero-arg|.

wildmenumode()                                      *wildmenumode()*
                Returns |TRUE| when the wildmenu is active and |FALSE|
                otherwise.  See 'wildmenu' and 'wildmode'.
                This can be used in mappings to handle the 'wildcharm' option
                gracefully. (Makes only sense with |mapmode-c| mappings).

                For example to make <c-j> work like <down> in wildmode, use: >
    :cnoremap <expr> <C-j> wildmenumode() ? "\<Down>\<Tab>" : "\<c-j>"
<
                (Note, this needs the 'wildcharm' option set appropriately).


win_findbuf({bufnr})                                    *win_findbuf()*
                Returns a list with |window-ID|s for windows that contain
                buffer {bufnr}.  When there is none the list is empty.

win_getid([{win} [, {tab}]])                            *win_getid()*
                Get the |window-ID| for the specified window.
                When {win} is missing use the current window.
                With {win} this is the window number.  The top window has
                number 1.  Use `win_getid(winnr())` for the current window.
                Without {tab} use the current tab, otherwise the tab with
                number {tab}.  The first tab has number one.
                Return zero if the window cannot be found.

win_gotoid({expr})                                      *win_gotoid()*
                Go to window with ID {expr}.  This may also change the current
                tabpage.
                Return 1 if successful, 0 if the window cannot be found.

win_id2tabwin({expr})                                   *win_id2tabwin()*
                Return a list with the tab number and window number of window
                with ID {expr}: [tabnr, winnr].
                Return [0, 0] if the window cannot be found.

win_id2win({expr})                                      *win_id2win()*
                Return the window number of window with ID {expr}.
                Return 0 if the window cannot be found in the current tabpage.
```

winbufnr()
winbufnr({nr}) The result is a Number, which is the number of the buffer
associated with window {nr}. {nr} can be the window number or
the |window-ID|.
When {nr} is zero, the number of the buffer in the current
window is returned.
When window {nr} doesn't exist, -1 is returned.
Example: >
:echo "The file in the current window is " . bufname(winbufnr(0))
<
wincol()
wincol() The result is a Number, which is the virtual column of the
cursor in the window. This is counting screen cells from the
left side of the window. The leftmost column is one.

winheight({nr}) *winheight()*
The result is a Number, which is the height of window {nr}.
{nr} can be the window number or the |window-ID|.
When {nr} is zero, the height of the current window is
returned. When window {nr} doesn't exist, -1 is returned.
An existing window always has a height of zero or more.
This excludes any window toolbar line.
Examples: >
:echo "The current window has " . winheight(0) . " lines."
<
winline()
winline() The result is a Number, which is the screen line of the cursor
in the window. This is counting screen lines from the top of
the window. The first line is one.
If the cursor was moved the view on the file will be updated
first, this may cause a scroll.

winnr()
winnr([{arg}]) The result is a Number, which is the number of the current
window. The top window has number 1.
When the optional argument is "$", the number of the
last window is returned (the window count). >
let window_count = winnr('$')
< When the optional argument is "#", the number of the last
accessed window is returned (where |CTRL-W_p| goes to).
If there is no previous window or it is in another tab page 0
is returned.
The number can be used with |CTRL-W_w| and ":wincmd w"
|:wincmd|.
Also see |tabpagewinnr()| and |win_getid()|.

winrestcmd()
winrestcmd() Returns a sequence of |:resize| commands that should restore
the current window sizes. Only works properly when no windows
are opened or closed and the current window and tab page is
unchanged.
Example: >
:let cmd = winrestcmd()
:call MessWithWindowSizes()
:exe cmd
<
winrestview()
winrestview({dict})
Uses the |Dictionary| returned by |winsaveview()| to restore
the view of the current window.
Note: The {dict} does not have to contain all values, that are

```
                returned by |winsaveview()|. If values are missing, those
                settings won't be restored. So you can use: >
                    :call winrestview({'curswant': 4})
<
                This will only set the curswant value (the column the cursor
                wants to move on vertical movements) of the cursor to column 5
                (yes, that is 5), while all other settings will remain the
                same. This is useful, if you set the cursor position manually.

                If you have changed the values the result is unpredictable.
                If the window size changed the result won't be the same.

                                                        *winsaveview()*
winsaveview()   Returns a |Dictionary| that contains information to restore
                the view of the current window.  Use |winrestview()| to
                restore the view.
                This is useful if you have a mapping that jumps around in the
                buffer and you want to go back to the original view.
                This does not save fold information.  Use the 'foldenable'
                option to temporarily switch off folding, so that folds are
                not opened when moving around. This may have side effects.
                The return value includes:
                        lnum            cursor line number
                        col             cursor column (Note: the first column
                                        zero, as opposed to what getpos()
                                        returns)
                        coladd          cursor column offset for 'virtualedit'
                        curswant        column for vertical movement
                        topline         first line in the window
                        topfill         filler lines, only in diff mode
                        leftcol         first column displayed
                        skipcol         columns skipped
                Note that no option values are saved.


winwidth({nr})                                          *winwidth()*
                The result is a Number, which is the width of window {nr}.
                {nr} can be the window number or the |window-ID|.
                When {nr} is zero, the width of the current window is
                returned.  When window {nr} doesn't exist, -1 is returned.
                An existing window always has a width of zero or more.
                Examples: >
  :echo "The current window has " . winwidth(0) . " columns."
  :if winwidth(0) <= 50
  :  exe "normal 50\<C-W>|"
  :endif
<               For getting the terminal or screen size, see the 'columns'
                option.


wordcount()                                             *wordcount()*
                The result is a dictionary of byte/chars/word statistics for
                the current buffer.  This is the same info as provided by
                |g_CTRL-G|
                The return value includes:
                        bytes           Number of bytes in the buffer
                        chars           Number of chars in the buffer
                        words           Number of words in the buffer
                        cursor_bytes    Number of bytes before cursor position
                                        (not in Visual mode)
                        cursor_chars    Number of chars before cursor position
                                        (not in Visual mode)
```

```
                        cursor_words    Number of words before cursor position
                                        (not in Visual mode)
                        visual_bytes    Number of bytes visually selected
                                        (only in Visual mode)
                        visual_chars    Number of chars visually selected
                                        (only in Visual mode)
                        visual_words    Number of words visually selected
                                        (only in Visual mode)


                                                        *writefile()*
writefile({list}, {fname} [, {flags}])
                Write |List| {list} to file {fname}.  Each list item is
                separated with a NL.  Each list item must be a String or
                Number.
                When {flags} contains "b" then binary mode is used: There will
                not be a NL after the last list item.  An empty item at the
                end does cause the last line in the file to end in a NL.

                When {flags} contains "a" then append mode is used, lines are
                appended to the file: >
                        :call writefile(["foo"], "event.log", "a")
                        :call writefile(["bar"], "event.log", "a")
>
<               All NL characters are replaced with a NUL character.
                Inserting CR characters needs to be done before passing {list}
                to writefile().
                An existing file is overwritten, if possible.
                When the write fails -1 is returned, otherwise 0.  There is an
                error message if the file can't be created or when writing
                fails.
                Also see |readfile()|.
                To copy a file byte for byte: >
                        :let fl = readfile("foo", "b")
                        :call writefile(fl, "foocopy", "b")


xor({expr}, {expr})                                     *xor()*
                Bitwise XOR on the two arguments.  The arguments are converted
                to a number.  A List, Dict or Float argument causes an error.
                Example: >
                        :let bits = xor(bits, 0x80)
<


                                                        *feature-list*
There are four types of features:
1.  Features that are only supported when they have been enabled when Vim
    was compiled |+feature-list|.  Example: >
        :if has("cindent")
2.  Features that are only supported when certain conditions have been met.
    Example: >
        :if has("gui_running")
<                                                       *has-patch*
3.  Included patches.  The "patch123" feature means that patch 123 has been
    included.  Note that this form does not check the version of Vim, you need
    to inspect |v:version| for that.
    Example (checking version 6.2.148 or later): >
        :if v:version > 602 || v:version == 602 && has("patch148")
<    Note that it's possible for patch 147 to be omitted even though 148 is
    included.
```

```
4.  Beyond a certain version or at a certain version and including a specific
    patch.  The "patch-7.4.237" feature means that the Vim version is 7.5 or
    later, or it is version 7.4 and patch 237 was included.
    Note that this only works for patch 7.4.237 and later, before that you
    need to use the example above that checks v:version.  Example: >
        :if has("patch-7.4.248")
<    Note that it's possible for patch 147 to be omitted even though 148 is
    included.

Hint: To find out if Vim supports backslashes in a file name (MS-Windows),
use: `if exists('+shellslash')`


acl                     Compiled with |ACL| support.
all_builtin_terms       Compiled with all builtin terminals enabled.
amiga                   Amiga version of Vim.
arabic                  Compiled with Arabic support |Arabic|.
arp                     Compiled with ARP support (Amiga).
autocmd                 Compiled with autocommand support. |autocommand|
balloon_eval            Compiled with |balloon-eval| support.
balloon_multiline       GUI supports multiline balloons.
beos                    BeOS version of Vim.
browse                  Compiled with |:browse| support, and browse() will
                        work.
browsefilter            Compiled with support for |browsefilter|.
builtin_terms           Compiled with some builtin terminals.
byte_offset             Compiled with support for 'o' in 'statusline'
cindent                 Compiled with 'cindent' support.
clientserver            Compiled with remote invocation support |clientserver|.
clipboard               Compiled with 'clipboard' support.
cmdline_compl           Compiled with |cmdline-completion| support.
cmdline_hist            Compiled with |cmdline-history| support.
cmdline_info            Compiled with 'showcmd' and 'ruler' support.
comments                Compiled with |'comments'| support.
compatible              Compiled to be very Vi compatible.
cryptv                  Compiled with encryption support |encryption|.
cscope                  Compiled with |cscope| support.
debug                   Compiled with "DEBUG" defined.
dialog_con              Compiled with console dialog support.
dialog_gui              Compiled with GUI dialog support.
diff                    Compiled with |vimdiff| and 'diff' support.
digraphs                Compiled with support for digraphs.
directx                 Compiled with support for DirectX and 'renderoptions'.
dnd                     Compiled with support for the "~ register |quote_~|.
ebcdic                  Compiled on a machine with ebcdic character set.
emacs_tags              Compiled with support for Emacs tags.
eval                    Compiled with expression evaluation support.  Always
                        true, of course!
ex_extra                |+ex_extra|, always true now
extra_search            Compiled with support for |'incsearch'| and
                        |'hlsearch'|
farsi                   Compiled with Farsi support |farsi|.
file_in_path            Compiled with support for |gf| and |<cfile>|
filterpipe              When 'shelltemp' is off pipes are used for shell
                        read/write/filter commands
find_in_path            Compiled with support for include file searches
                        |+find_in_path|.
float                   Compiled with support for |Float|.
fname_case              Case in file names matters (for Amiga, MS-DOS, and
                        Windows this is not present).
folding                 Compiled with |folding| support.
footer                  Compiled with GUI footer support. |gui-footer|
```

```
fork                    Compiled to use fork()/exec() instead of system().
gettext                 Compiled with message translation |multi-lang|
gui                     Compiled with GUI enabled.
gui_athena              Compiled with Athena GUI.
gui_gnome               Compiled with Gnome support (gui_gtk is also defined).
gui_gtk                 Compiled with GTK+ GUI (any version).
gui_gtk2                Compiled with GTK+ 2 GUI (gui_gtk is also defined).
gui_gtk3                Compiled with GTK+ 3 GUI (gui_gtk is also defined).
gui_mac                 Compiled with Macintosh GUI.
gui_motif               Compiled with Motif GUI.
gui_photon              Compiled with Photon GUI.
gui_running             Vim is running in the GUI, or it will start soon.
gui_win32               Compiled with MS Windows Win32 GUI.
gui_win32s              idem, and Win32s system being used (Windows 3.1)
hangul_input            Compiled with Hangul input support. |hangul|
iconv                   Can use iconv() for conversion.
insert_expand           Compiled with support for CTRL-X expansion commands in
                        Insert mode.
jumplist                Compiled with |jumplist| support.
keymap                  Compiled with 'keymap' support.
lambda                  Compiled with |lambda| support.
langmap                 Compiled with 'langmap' support.
libcall                 Compiled with |libcall()| support.
linebreak               Compiled with 'linebreak', 'breakat', 'showbreak' and
                        'breakindent' support.
lispindent              Compiled with support for lisp indenting.
listcmds                Compiled with commands for the buffer list |:files|
                        and the argument list |arglist|.
localmap                Compiled with local mappings and abbr. |:map-local|
lua                     Compiled with Lua interface |Lua|.
mac                     Any Macintosh version of Vim, but not all OS X.
macunix                 Compiled for OS X, with |mac-darwin-feature|
osx                     Compiled for OS X, with or w/o |mac-darwin-feature|
menu                    Compiled with support for |:menu|.
mksession               Compiled with support for |:mksession|.
modify_fname            Compiled with file name modifiers. |filename-modifiers|
mouse                   Compiled with support mouse.
mouse_dec               Compiled with support for Dec terminal mouse.
mouse_gpm               Compiled with support for gpm (Linux console mouse)
mouse_netterm           Compiled with support for netterm mouse.
mouse_pterm             Compiled with support for qnx pterm mouse.
mouse_sysmouse          Compiled with support for sysmouse (*BSD console mouse)
mouse_sgr               Compiled with support for sgr mouse.
mouse_urxvt             Compiled with support for urxvt mouse.
mouse_xterm             Compiled with support for xterm mouse.
mouseshape              Compiled with support for 'mouseshape'.
multi_byte              Compiled with support for 'encoding'
multi_byte_encoding     'encoding' is set to a multi-byte encoding.
multi_byte_ime          Compiled with support for IME input method.
multi_lang              Compiled with support for multiple languages.
mzscheme                Compiled with MzScheme interface |mzscheme|.
netbeans_enabled        Compiled with support for |netbeans| and connected.
netbeans_intg           Compiled with support for |netbeans|.
num64                   Compiled with 64-bit |Number| support.
ole                     Compiled with OLE automation support for Win32.
packages                Compiled with |packages| support.
path_extra              Compiled with up/downwards search in 'path' and 'tags'
perl                    Compiled with Perl interface.
persistent_undo         Compiled with support for persistent undo history.
postscript              Compiled with PostScript file printing.
printer                 Compiled with |:hardcopy| support.
profile                 Compiled with |:profile| support.
```

```
python                  Compiled with Python 2.x interface. |has-python|
python3                 Compiled with Python 3.x interface. |has-python|
pythonx                 Compiled with |python_x| interface. |has-pythonx|
qnx                     QNX version of Vim.
quickfix                Compiled with |quickfix| support.
reltime                 Compiled with |reltime()| support.
rightleft               Compiled with 'rightleft' support.
ruby                    Compiled with Ruby interface |ruby|.
scrollbind              Compiled with 'scrollbind' support.
showcmd                 Compiled with 'showcmd' support.
signs                   Compiled with |:sign| support.
smartindent             Compiled with 'smartindent' support.
spell                   Compiled with spell checking support |spell|.
startuptime             Compiled with |--startuptime| support.
statusline              Compiled with support for 'statusline', 'rulerformat'
                        and special formats of 'titlestring' and 'iconstring'.
sun_workshop            Compiled with support for Sun |workshop|.
syntax                  Compiled with syntax highlighting support |syntax|.
syntax_items            There are active syntax highlighting items for the
                        current buffer.
system                  Compiled to use system() instead of fork()/exec().
tag_binary              Compiled with binary searching in tags files
                        |tag-binary-search|.
tag_old_static          Compiled with support for old static tags
                        |tag-old-static|.
tag_any_white           Compiled with support for any white characters in tags
                        files |tag-any-white|.
tcl                     Compiled with Tcl interface.
termguicolors           Compiled with true color in terminal support.
terminal                Compiled with |terminal| support.
terminfo                Compiled with terminfo instead of termcap.
termresponse            Compiled with support for |t_RV| and |v:termresponse|.
textobjects             Compiled with support for |text-objects|.
tgetent                 Compiled with tgetent support, able to use a termcap
                        or terminfo file.
timers                  Compiled with |timer_start()| support.
title                   Compiled with window title support |'title'|.
toolbar                 Compiled with support for |gui-toolbar|.
ttyin                   input is a terminal (tty)
ttyout                  output is a terminal (tty)
unix                    Unix version of Vim. *+unix*
unnamedplus             Compiled with support for "unnamedplus" in 'clipboard'
user_commands           User-defined commands.
vertsplit               Compiled with vertically split windows |:vsplit|.
vim_starting            True while initial source'ing takes place. |startup|
                        *vim_starting*
viminfo                 Compiled with viminfo support.
virtualedit             Compiled with 'virtualedit' option.
visual                  Compiled with Visual mode.
visualextra             Compiled with extra Visual mode commands.
                        |blockwise-operators|.
vms                     VMS version of Vim.
vreplace                Compiled with |gR| and |gr| commands.
wildignore              Compiled with 'wildignore' option.
wildmenu                Compiled with 'wildmenu' option.
win32                   Win32 version of Vim (MS-Windows 95 and later, 32 or
                        64 bits)
win32unix               Win32 version of Vim, using Unix files (Cygwin)
win64                   Win64 version of Vim (MS-Windows 64 bit).
win95                   Win32 version for MS-Windows 95/98/ME.
winaltkeys              Compiled with 'winaltkeys' option.
windows                 Compiled with support for more than one window.
```

```
writebackup             Compiled with 'writebackup' default on.
xfontset                Compiled with X fontset support |xfontset|.
xim                     Compiled with X input method support |xim|.
xpm                     Compiled with pixmap support.
xpm_w32                 Compiled with pixmap support for Win32. (Only for
                        backward compatibility. Use "xpm" instead.)
xsmp                    Compiled with X session management support.
xsmp_interact           Compiled with interactive X session management support.
xterm_clipboard         Compiled with support for xterm clipboard.
xterm_save              Compiled with support for saving and restoring the
                        xterm screen.
x11                     Compiled with X11 support.
```

string-match

Matching a pattern in a String

A regexp pattern as explained at |pattern| is normally used to find a match in the buffer lines. When a pattern is used to find a match in a String, almost everything works in the same way. The difference is that a String is handled like it is one line. When it contains a "\n" character, this is not seen as a line break for the pattern. It can be matched with a "\n" in the pattern, or with ".". Example: >

```
        :let a = "aaaa\nxxxx"
        :echo matchstr(a, "..\n..")
        aa
        xx
        :echo matchstr(a, "a.x")
        a
        x
```

Don't forget that "^" will only match at the first character of the String and "$" at the last character of the string. They don't match after or before a "\n".

==

5. Defining functions *user-functions*

New functions can be defined. These can be called just like builtin functions. The function executes a sequence of Ex commands. Normal mode commands can be executed with the |:normal| command.

The function name must start with an uppercase letter, to avoid confusion with builtin functions. To prevent from using the same name in different scripts avoid obvious, short names. A good habit is to start the function name with the name of the script, e.g., "HTMLcolor()".

It's also possible to use curly braces, see |curly-braces-names|. And the |autoload| facility is useful to define a function only when it's called.

local-function

A function local to a script must start with "s:". A local script function can only be called from within the script and from functions, user commands and autocommands defined in the script. It is also possible to call the function from a mapping defined in the script, but then |<SID>| must be used instead of "s:" when the mapping is expanded outside of the script.
There are only script-local functions, no buffer-local or window-local functions.

:fu *:function* *E128* *E129* *E123*

```
:fu[nction]             List all functions and their arguments.

:fu[nction] {name}      List function {name}.
```

```
                        {name} can also be a |Dictionary| entry that is a
                        |Funcref|: >
                                :function dict.init

:fu[nction] /{pattern}  List functions with a name matching {pattern}.
                        Example that lists all functions ending with "File": >
                                :function /File$
<
                                                        *:function-verbose*
When 'verbose' is non-zero, listing a function will also display where it was
last defined. Example: >

    :verbose function SetFileTypeSH
        function SetFileTypeSH(name)
            Last set from /usr/share/vim/vim-7.0/filetype.vim
<
See |:verbose-cmd| for more information.

                                                *E124* *E125* *E853* *E884*
:fu[nction][!] {name}([arguments]) [range] [abort] [dict] [closure]
                        Define a new function by the name {name}.  The name
                        must be made of alphanumeric characters and '_', and
                        must start with a capital or "s:" (see above).  Note
                        that using "b:" or "g:" is not allowed. (since patch
                        7.4.260 E884 is given if the function name has a colon
                        in the name, e.g. for "foo:bar()".  Before that patch
                        no error was given).

                        {name} can also be a |Dictionary| entry that is a
                        |Funcref|: >
                                :function dict.init(arg)
<                       "dict" must be an existing dictionary.  The entry
                        "init" is added if it didn't exist yet.  Otherwise [!]
                        is required to overwrite an existing function.  The
                        result is a |Funcref| to a numbered function.  The
                        function can only be used with a |Funcref| and will be
                        deleted if there are no more references to it.
                                                        *E127* *E122*
                        When a function by this name already exists and [!] is
                        not used an error message is given.  When [!] is used,
                        an existing function is silently replaced.  Unless it
                        is currently being executed, that is an error.
                        NOTE: Use ! wisely.  If used without care it can cause
                        an existing function to be replaced unexpectedly,
                        which is hard to debug.

                        For the {arguments} see |function-argument|.

                                *:func-range* *a:firstline* *a:lastline*
                        When the [range] argument is added, the function is
                        expected to take care of a range itself.  The range is
                        passed as "a:firstline" and "a:lastline".  If [range]
                        is excluded, ":{range}call" will call the function for
                        each line in the range, with the cursor on the start
                        of each line.  See |function-range-example|.
                        The cursor is still moved to the first line of the
                        range, as is the case with all Ex commands.
                                                        *:func-abort*
                        When the [abort] argument is added, the function will
                        abort as soon as an error is detected.
                                                        *:func-dict*
                        When the [dict] argument is added, the function must
```

```
                        be invoked through an entry in a |Dictionary|.  The
                        local variable "self" will then be set to the
                        dictionary.  See |Dictionary-function|.
                                                *:func-closure* *E932*
                        When the [closure] argument is added, the function
                        can access variables and arguments from the outer
                        scope.  This is usually called a closure.  In this
                        example Bar() uses "x" from the scope of Foo().  It
                        remains referenced even after Foo() returns: >
                                :function! Foo()
                                :  let x = 0
                                :  function! Bar() closure
                                :    let x += 1
                                :    return x
                                :  endfunction
                                :  return funcref('Bar')
                                :endfunction

                                :let F = Foo()
                                :echo F()
<                               1 >
                                :echo F()
<                               2 >
                                :echo F()
<                               3

                                        *function-search-undo*
                        The last used search pattern and the redo command "."
                        will not be changed by the function.  This also
                        implies that the effect of |:nohlsearch| is undone
                        when the function returns.

                                *:endf* *:endfunction* *E126* *E193* *W22*
:endf[unction] [argument]
                        The end of a function definition.  Best is to put it
                        on a line by its own, without [argument].

                        [argument] can be:
                                | command       command to execute next
                                \n command      command to execute next
                                " comment       always ignored
                                anything else   ignored, warning given when
                                                'verbose' is non-zero
                        The support for a following command was added in Vim
                        8.0.0654, before that any argument was silently
                        ignored.

                        To be able to define a function inside an `:execute`
                        command, use line breaks instead of |:bar|: >
                                :exe "func Foo()\necho 'foo'\nendfunc"
<
                                *:delf* *:delfunction* *E130* *E131* *E933*
:delf[unction][!] {name}
                        Delete function {name}.
                        {name} can also be a |Dictionary| entry that is a
                        |Funcref|: >
                                :delfunc dict.init
<                       This will remove the "init" entry from "dict".  The
                        function is deleted if there are no more references to
                        it.
                        With the ! there is no error if the function does not
                        exist.
```

:retu *:return* *E133*

:retu[rn] [expr] Return from a function. When "[expr]" is given, it is evaluated and returned as the result of the function. If "[expr]" is not given, the number 0 is returned. When a function ends without an explicit ":return", the number 0 is returned.
Note that there is no check for unreachable lines, thus there is no warning if commands follow ":return".

If the ":return" is used after a |:try| but before the matching |:finally| (if present), the commands following the ":finally" up to the matching |:endtry| are executed first. This process applies to all nested ":try"s inside the function. The function returns at the outermost ":endtry".

function-argument *a:var*

An argument can be defined by giving its name. In the function this can then be used as "a:name" ("a:" for argument).

a:0 *a:1* *a:000* *E740* *...*

Up to 20 arguments can be given, separated by commas. After the named arguments an argument "..." can be specified, which means that more arguments may optionally be following. In the function the extra arguments can be used as "a:1", "a:2", etc. "a:0" is set to the number of extra arguments (which can be 0). "a:000" is set to a |List| that contains these arguments. Note that "a:1" is the same as "a:000[0]".

E742

The a: scope and the variables in it cannot be changed, they are fixed. However, if a composite type is used, such as |List| or |Dictionary| , you can change their contents. Thus you can pass a |List| to a function and have the function add an item to it. If you want to make sure the function cannot change a |List| or |Dictionary| use |:lockvar|.

When not using "...", the number of arguments in a function call must be equal to the number of named arguments. When using "...", the number of arguments may be larger.

It is also possible to define a function without any arguments. You must still supply the () then. The body of the function follows in the next lines, until the matching |:endfunction|. It is allowed to define another function inside a function body.

local-variables

Inside a function local variables can be used. These will disappear when the function returns. Global variables need to be accessed with "g:".

Example: >

```
  :function Table(title, ...)
  :  echohl Title
  :  echo a:title
  :  echohl None
  :  echo a:0 . " items:"
  :  for s in a:000
  :    echon ' ' . s
  :  endfor
  :endfunction
```

This function can then be called with: >

```
  call Table("Table", "line1", "line2")
  call Table("Empty Table")
```

To return more than one value, return a |List|: >

```
  :function Compute(n1, n2)
  :  if a:n2 == 0
  :    return ["fail", 0]
  :  endif
  :  return ["ok", a:n1 / a:n2]
  :endfunction

This function can then be called with: >
  :let [success, div] = Compute(102, 6)
  :if success == "ok"
  :  echo div
  :endif
<
                                                *:cal* *:call* *E107* *E117*
:[range]cal[l] {name}([arguments])
               Call a function.  The name of the function and its arguments
               are as specified with |:function|.  Up to 20 arguments can be
               used.  The returned value is discarded.
               Without a range and for functions that accept a range, the
               function is called once.  When a range is given the cursor is
               positioned at the start of the first line before executing the
               function.
               When a range is given and the function doesn't handle it
               itself, the function is executed for each line in the range,
               with the cursor in the first column of that line.  The cursor
               is left at the last line (possibly moved by the last function
               call).  The arguments are re-evaluated for each line.  Thus
               this works:
                                                *function-range-example*  >
        :function Mynumber(arg)
        :  echo line(".") . " " . a:arg
        :endfunction
        :1,5call Mynumber(getline("."))
<
               The "a:firstline" and "a:lastline" are defined anyway, they
               can be used to do something different at the start or end of
               the range.

               Example of a function that handles the range itself: >

        :function Cont() range
        :  execute (a:firstline + 1) . "," . a:lastline . 's/^/\t\\ '
        :endfunction
        :4,8call Cont()
<
               This function inserts the continuation character "\" in front
               of all the lines in the range, except the first one.

               When the function returns a composite value it can be further
               dereferenced, but the range will not be used then.  Example: >
        :4,8call GetDict().method()
<              Here GetDict() gets the range but method() does not.

                                                              *E132*
The recursiveness of user functions is restricted with the |'maxfuncdepth'|
option.


AUTOMATICALLY LOADING FUNCTIONS ~
                                                          *autoload-functions*
When using many or large functions, it's possible to automatically define them
only when they are used.  There are two methods: with an autocommand and with
```

```
the "autoload" directory in 'runtimepath'.


Using an autocommand ~

This is introduced in the user manual, section |41.14|.

The autocommand is useful if you have a plugin that is a long Vim script file.
You can define the autocommand and quickly quit the script with |:finish|.
That makes Vim startup faster.  The autocommand should then load the same file
again, setting a variable to skip the |:finish| command.

Use the FuncUndefined autocommand event with a pattern that matches the
function(s) to be defined.  Example: >

        :au FuncUndefined BufNet* source ~/vim/bufnetfuncs.vim

The file "~/vim/bufnetfuncs.vim" should then define functions that start with
"BufNet".  Also see |FuncUndefined|.


Using an autoload script ~
                                                        *autoload* *E746*
This is introduced in the user manual, section |41.15|.

Using a script in the "autoload" directory is simpler, but requires using
exactly the right file name.  A function that can be autoloaded has a name
like this: >

        :call filename#funcname()

When such a function is called, and it is not defined yet, Vim will search the
"autoload" directories in 'runtimepath' for a script file called
"filename.vim".  For example "~/.vim/autoload/filename.vim".  That file should
then define the function like this: >

        function filename#funcname()
           echo "Done!"
        endfunction

The file name and the name used before the # in the function must match
exactly, and the defined function must have the name exactly as it will be
called.

It is possible to use subdirectories.  Every # in the function name works like
a path separator.  Thus when calling a function: >

        :call foo#bar#func()

Vim will look for the file "autoload/foo/bar.vim" in 'runtimepath'.

This also works when reading a variable that has not been set yet: >

        :let l = foo#bar#lvar

However, when the autoload script was already loaded it won't be loaded again
for an unknown variable.

When assigning a value to such a variable nothing special happens.  This can
be used to pass settings to the autoload script before it's loaded: >

        :let foo#bar#toggle = 1
```

```
        :call foo#bar#func()

Note that when you make a mistake and call a function that is supposed to be
defined in an autoload script, but the script doesn't actually define the
function, the script will be sourced every time you try to call the function.
And you will get an error message every time.

Also note that if you have two script files, and one calls a function in the
other and vice versa, before the used function is defined, it won't work.
Avoid using the autoload functionality at the toplevel.

Hint: If you distribute a bunch of scripts you can pack them together with the
|vimball| utility.  Also read the user manual |distribute-script|.

==============================================================================
6. Curly braces names                                   *curly-braces-names*

In most places where you can use a variable, you can use a "curly braces name"
variable.  This is a regular variable name with one or more expressions
wrapped in braces {} like this: >
        my_{adjective}_variable

When Vim encounters this, it evaluates the expression inside the braces, puts
that in place of the expression, and re-interprets the whole as a variable
name.  So in the above example, if the variable "adjective" was set to
"noisy", then the reference would be to "my_noisy_variable", whereas if
"adjective" was set to "quiet", then it would be to "my_quiet_variable".

One application for this is to create a set of variables governed by an option
value.  For example, the statement >
        echo my_{&background}_message

would output the contents of "my_dark_message" or "my_light_message" depending
on the current value of 'background'.

You can use multiple brace pairs: >
        echo my_{adverb}_{adjective}_message
..or even nest them: >
        echo my_{ad{end_of_word}}_message
where "end_of_word" is either "verb" or "jective".

However, the expression inside the braces must evaluate to a valid single
variable name, e.g. this is invalid: >
        :let foo='a + b'
        :echo c{foo}d
.. since the result of expansion is "ca + bd", which is not a variable name.

                                              *curly-braces-function-names*
You can call and define functions by an evaluated name in a similar way.
Example: >
        :let func_end='whizz'
        :call my_func_{func_end}(parameter)

This would call the function "my_func_whizz(parameter)".

This does NOT work: >
  :let i = 3
  :let @{i} = ''  " error
  :echo @{i}      " error

==============================================================================
7. Commands                                         *expression-commands*
```

```
:let {var-name} = {expr1}                               *:let* *E18*
                        Set internal variable {var-name} to the result of the
                        expression {expr1}.  The variable will get the type
                        from the {expr}.  If {var-name} didn't exist yet, it
                        is created.

:let {var-name}[{idx}] = {expr1}                        *E689*
                        Set a list item to the result of the expression
                        {expr1}.  {var-name} must refer to a list and {idx}
                        must be a valid index in that list.  For nested list
                        the index can be repeated.
                        This cannot be used to add an item to a |List|.
                        This cannot be used to set a byte in a String.  You
                        can do that like this: >
                                :let var = var[0:2] . 'X' . var[4:]
<
                                                        *E711* *E719*
:let {var-name}[{idx1}:{idx2}] = {expr1}                *E708* *E709* *E710*
                        Set a sequence of items in a |List| to the result of
                        the expression {expr1}, which must be a list with the
                        correct number of items.
                        {idx1} can be omitted, zero is used instead.
                        {idx2} can be omitted, meaning the end of the list.
                        When the selected range of items is partly past the
                        end of the list, items will be added.

                                        *:let+=* *:let-=* *:let.=* *E734*
:let {var} += {expr1}   Like ":let {var} = {var} + {expr1}".
:let {var} -= {expr1}   Like ":let {var} = {var} - {expr1}".
:let {var} .= {expr1}   Like ":let {var} = {var} . {expr1}".
                        These fail if {var} was not set yet and when the type
                        of {var} and {expr1} don't fit the operator.


:let ${env-name} = {expr1}                      *:let-environment* *:let-$*
                        Set environment variable {env-name} to the result of
                        the expression {expr1}.  The type is always String.
:let ${env-name} .= {expr1}
                        Append {expr1} to the environment variable {env-name}.
                        If the environment variable didn't exist yet this
                        works like "=".

:let @{reg-name} = {expr1}                      *:let-register* *:let-@*
                        Write the result of the expression {expr1} in register
                        {reg-name}.  {reg-name} must be a single letter, and
                        must be the name of a writable register (see
                        |registers|).  "@@" can be used for the unnamed
                        register, "@/" for the search pattern.
                        If the result of {expr1} ends in a <CR> or <NL>, the
                        register will be linewise, otherwise it will be set to
                        characterwise.
                        This can be used to clear the last search pattern: >
                                :let @/ = ""
<                       This is different from searching for an empty string,
                        that would match everywhere.

:let @{reg-name} .= {expr1}
                        Append {expr1} to register {reg-name}.  If the
                        register was empty it's like setting it to {expr1}.

:let &{option-name} = {expr1}                   *:let-option* *:let-&*
```

```
                        Set option {option-name} to the result of the
                        expression {expr1}.  A String or Number value is
                        always converted to the type of the option.
                        For an option local to a window or buffer the effect
                        is just like using the |:set| command: both the local
                        value and the global value are changed.
                        Example: >
                                :let &path = &path . ',/usr/local/include'
<                       This also works for terminal codes in the form t_xx.
                        But only for alphanumerical names.  Example: >
                                :let &t_k1 = "\<Esc>[234;"
<                       When the code does not exist yet it will be created as
                        a terminal key code, there is no error.

:let &{option-name} .= {expr1}
                        For a string option: Append {expr1} to the value.
                        Does not insert a comma like |:set+=|.

:let &{option-name} += {expr1}
:let &{option-name} -= {expr1}
                        For a number or boolean option: Add or subtract
                        {expr1}.

:let &l:{option-name} = {expr1}
:let &l:{option-name} .= {expr1}
:let &l:{option-name} += {expr1}
:let &l:{option-name} -= {expr1}
                        Like above, but only set the local value of an option
                        (if there is one).  Works like |:setlocal|.

:let &g:{option-name} = {expr1}
:let &g:{option-name} .= {expr1}
:let &g:{option-name} += {expr1}
:let &g:{option-name} -= {expr1}
                        Like above, but only set the global value of an option
                        (if there is one).  Works like |:setglobal|.

:let [{name1}, {name2}, ...] = {expr1}          *:let-unpack* *E687* *E688*
                        {expr1} must evaluate to a |List|.  The first item in
                        the list is assigned to {name1}, the second item to
                        {name2}, etc.
                        The number of names must match the number of items in
                        the |List|.
                        Each name can be one of the items of the ":let"
                        command as mentioned above.
                        Example: >
                                :let [s, item] = GetItem(s)
<                       Detail: {expr1} is evaluated first, then the
                        assignments are done in sequence.  This matters if
                        {name2} depends on {name1}.  Example: >
                                :let x = [0, 1]
                                :let i = 0
                                :let [i, x[i]] = [1, 2]
                                :echo x
<                       The result is [0, 2].

:let [{name1}, {name2}, ...] .= {expr1}
:let [{name1}, {name2}, ...] += {expr1}
:let [{name1}, {name2}, ...] -= {expr1}
                        Like above, but append/add/subtract the value for each
                        |List| item.
```

```
:let [{name}, ..., ; {lastname}] = {expr1}
                        Like |:let-unpack| above, but the |List| may have more
                        items than there are names.  A list of the remaining
                        items is assigned to {lastname}.  If there are no
                        remaining items {lastname} is set to an empty list.
                        Example: >
                                :let [a, b; rest] = ["aval", "bval", 3, 4]
<
:let [{name}, ..., ; {lastname}] .= {expr1}
:let [{name}, ..., ; {lastname}] += {expr1}
:let [{name}, ..., ; {lastname}] -= {expr1}
                        Like above, but append/add/subtract the value for each
                        |List| item.

                                                                *E121*
:let {var-name} ..      List the value of variable {var-name}.  Multiple
                        variable names may be given.  Special names recognized
                        here:                           *E738*
                          g:    global variables
                          b:    local buffer variables
                          w:    local window variables
                          t:    local tab page variables
                          s:    script-local variables
                          l:    local function variables
                          v:    Vim variables.

:let                    List the values of all variables.  The type of the
                        variable is indicated before the value:
                            <nothing>   String
                                #       Number
                                *       Funcref


:unl[et][!] {name} ...                          *:unlet* *:unl* *E108* *E795*
                        Remove the internal variable {name}.  Several variable
                        names can be given, they are all removed.  The name
                        may also be a |List| or |Dictionary| item.
                        With [!] no error message is given for non-existing
                        variables.
                        One or more items from a |List| can be removed: >
                                :unlet list[3]    " remove fourth item
                                :unlet list[3:]   " remove fourth item to last
<                       One item from a |Dictionary| can be removed at a time: >
                                :unlet dict['two']
                                :unlet dict.two
<                       This is especially useful to clean up used global
                        variables and script-local variables (these are not
                        deleted when the script ends).  Function-local
                        variables are automatically deleted when the function
                        ends.

:lockv[ar][!] [depth] {name} ...                        *:lockvar* *:lockv*
                        Lock the internal variable {name}.  Locking means that
                        it can no longer be changed (until it is unlocked).
                        A locked variable can be deleted: >
                                :lockvar v
                                :let v = 'asdf'         " fails!
                                :unlet v
<                                                       *E741* *E940*
                        If you try to change a locked variable you get an
                        error message: "E741: Value is locked: {name}".
                        If you try to lock or unlock a built-in variable you
```

```
                        get an error message: "E940: Cannot lock or unlock
                        variable {name}".

                        [depth] is relevant when locking a |List| or
                        |Dictionary|.  It specifies how deep the locking goes:
                                1       Lock the |List| or |Dictionary| itself,
                                        cannot add or remove items, but can
                                        still change their values.
                                2       Also lock the values, cannot change
                                        the items.  If an item is a |List| or
                                        |Dictionary|, cannot add or remove
                                        items, but can still change the
                                        values.
                                3       Like 2 but for the |List| /
                                        |Dictionary| in the |List| /
                                        |Dictionary|, one level deeper.
                        The default [depth] is 2, thus when {name} is a |List|
                        or |Dictionary| the values cannot be changed.
                                                                *E743*
                        For unlimited depth use [!] and omit [depth].
                        However, there is a maximum depth of 100 to catch
                        loops.

                        Note that when two variables refer to the same |List|
                        and you lock one of them, the |List| will also be
                        locked when used through the other variable.
                        Example: >
                                :let l = [0, 1, 2, 3]
                                :let cl = l
                                :lockvar l
                                :let cl[1] = 99         " won't work!
<                       You may want to make a copy of a list to avoid this.
                        See |deepcopy()|.


:unlo[ckvar][!] [depth] {name} ...                      *:unlockvar* *:unlo*
                        Unlock the internal variable {name}.  Does the
                        opposite of |:lockvar|.


:if {expr1}                     *:if* *:endif* *:en* *E171* *E579* *E580*
:en[dif]                Execute the commands until the next matching ":else"
                        or ":endif" if {expr1} evaluates to non-zero.

                        From Vim version 4.5 until 5.0, every Ex command in
                        between the ":if" and ":endif" is ignored.  These two
                        commands were just to allow for future expansions in a
                        backward compatible way.  Nesting was allowed.  Note
                        that any ":else" or ":elseif" was ignored, the "else"
                        part was not executed either.

                        You can use this to remain compatible with older
                        versions: >
                                :if version >= 500
                                :  version-5-specific-commands
                                :endif
<                       The commands still need to be parsed to find the
                        "endif".  Sometimes an older Vim has a problem with a
                        new command.  For example, ":silent" is recognized as
                        a ":substitute" command.  In that case ":execute" can
                        avoid problems: >
                                :if version >= 600
```

```
                                :  execute "silent 1,$delete"
                                :endif
<
                        NOTE: The ":append" and ":insert" commands don't work
                        properly in between ":if" and ":endif".

                                                *:else* *:el* *E581* *E583*
:el[se]                 Execute the commands until the next matching ":else"
                        or ":endif" if they previously were not being
                        executed.

                                            *:elseif* *:elsei* *E582* *E584*
:elsei[f] {expr1}       Short for ":else" ":if", with the addition that there
                        is no extra ":endif".

:wh[ile] {expr1}                            *:while* *:endwhile* *:wh* *:endw*
                                                *E170* *E585* *E588* *E733*
:endw[hile]             Repeat the commands between ":while" and ":endwhile",
                        as long as {expr1} evaluates to non-zero.
                        When an error is detected from a command inside the
                        loop, execution continues after the "endwhile".
                        Example: >
                                :let lnum = 1
                                :while lnum <= line("$")
                                   :call FixLine(lnum)
                                   :let lnum = lnum + 1
                                :endwhile
<
                        NOTE: The ":append" and ":insert" commands don't work
                        properly inside a ":while" and ":for" loop.

:for {var} in {list}                                    *:for* *E690* *E732*
:endfo[r]                                               *:endfo* *:endfor*
                        Repeat the commands between ":for" and ":endfor" for
                        each item in {list}.  Variable {var} is set to the
                        value of each item.
                        When an error is detected for a command inside the
                        loop, execution continues after the "endfor".
                        Changing {list} inside the loop affects what items are
                        used.  Make a copy if this is unwanted: >
                                :for item in copy(mylist)
<                       When not making a copy, Vim stores a reference to the
                        next item in the list, before executing the commands
                        with the current item.  Thus the current item can be
                        removed without effect.  Removing any later item means
                        it will not be found.  Thus the following example
                        works (an inefficient way to make a list empty): >
                                for item in mylist
                                   call remove(mylist, 0)
                                endfor
<                       Note that reordering the list (e.g., with sort() or
                        reverse()) may have unexpected effects.

:for [{var1}, {var2}, ...] in {listlist}
:endfo[r]
                        Like ":for" above, but each item in {listlist} must be
                        a list, of which each item is assigned to {var1},
                        {var2}, etc.  Example: >
                                :for [lnum, col] in [[1, 3], [2, 5], [3, 8]]
                                   :echo getline(lnum)[col]
                                :endfor
<
```

:continue *:con* *E586*
:con[tinue] When used inside a ":while" or ":for" loop, jumps back
to the start of the loop.
If it is used after a |:try| inside the loop but
before the matching |:finally| (if present), the
commands following the ":finally" up to the matching
|:endtry| are executed first. This process applies to
all nested ":try"s inside the loop. The outermost
":endtry" then jumps back to the start of the loop.

:break *:brea* *E587*
:brea[k] When used inside a ":while" or ":for" loop, skips to
the command after the matching ":endwhile" or
":endfor".
If it is used after a |:try| inside the loop but
before the matching |:finally| (if present), the
commands following the ":finally" up to the matching
|:endtry| are executed first. This process applies to
all nested ":try"s inside the loop. The outermost
":endtry" then jumps to the command after the loop.

:try *:try* *:endt* *:endtry* *E600* *E601* *E602*
:endt[ry] Change the error handling for the commands between
":try" and ":endtry" including everything being
executed across ":source" commands, function calls,
or autocommand invocations.

When an error or interrupt is detected and there is
a |:finally| command following, execution continues
after the ":finally". Otherwise, or when the
":endtry" is reached thereafter, the next
(dynamically) surrounding ":try" is checked for
a corresponding ":finally" etc. Then the script
processing is terminated. (Whether a function
definition has an "abort" argument does not matter.)
Example: >
:try | edit too much | finally | echo "cleanup" | endtry
:echo "impossible" " not reached, script terminated above
<
Moreover, an error or interrupt (dynamically) inside
":try" and ":endtry" is converted to an exception. It
can be caught as if it were thrown by a |:throw|
command (see |:catch|). In this case, the script
processing is not terminated.

The value "Vim:Interrupt" is used for an interrupt
exception. An error in a Vim command is converted
to a value of the form "Vim({command}):{errmsg}",
other errors are converted to a value of the form
"Vim:{errmsg}". {command} is the full command name,
and {errmsg} is the message that is displayed if the
error exception is not caught, always beginning with
the error number.
Examples: >
:try | sleep 100 | catch /^Vim:Interrupt$/ | endtry
:try | edit | catch /^Vim(edit):E\d\+/ | echo "error" | endtry
<
:cat *:catch* *E603* *E604* *E605*
:cat[ch] /{pattern}/ The following commands until the next |:catch|,
|:finally|, or |:endtry| that belongs to the same
|:try| as the ":catch" are executed when an exception
matching {pattern} is being thrown and has not yet

```
                        been caught by a previous ":catch".  Otherwise, these
                        commands are skipped.
                        When {pattern} is omitted all errors are caught.
                        Examples: >
                :catch /^Vim:Interrupt$/        " catch interrupts (CTRL-C)
                :catch /^Vim\%((\a\+)\)\=:E/    " catch all Vim errors
                :catch /^Vim\%((\a\+)\)\=:/     " catch errors and interrupts
                :catch /^Vim(write):/           " catch all errors in :write
                :catch /^Vim\%((\a\+)\)\=:E123/ " catch error E123
                :catch /my-exception/           " catch user exception
                :catch /.*/                     " catch everything
                :catch                          " same as /.*/
<
                        Another character can be used instead of / around the
                        {pattern}, so long as it does not have a special
                        meaning (e.g., '|' or '"') and doesn't occur inside
                        {pattern}.
                        Information about the exception is available in
                        |v:exception|.  Also see |throw-variables|.
                        NOTE: It is not reliable to ":catch" the TEXT of
                        an error message because it may vary in different
                        locales.

                                        *:fina* *:finally* *E606* *E607*
:fina[lly]              The following commands until the matching |:endtry|
                        are executed whenever the part between the matching
                        |:try| and the ":finally" is left:  either by falling
                        through to the ":finally" or by a |:continue|,
                        |:break|, |:finish|, or |:return|, or by an error or
                        interrupt or exception (see |:throw|).

                                                        *:th* *:throw* *E608*
:th[row] {expr1}        The {expr1} is evaluated and thrown as an exception.
                        If the ":throw" is used after a |:try| but before the
                        first corresponding |:catch|, commands are skipped
                        until the first ":catch" matching {expr1} is reached.
                        If there is no such ":catch" or if the ":throw" is
                        used after a ":catch" but before the |:finally|, the
                        commands following the ":finally" (if present) up to
                        the matching |:endtry| are executed.  If the ":throw"
                        is after the ":finally", commands up to the ":endtry"
                        are skipped.  At the ":endtry", this process applies
                        again for the next dynamically surrounding ":try"
                        (which may be found in a calling function or sourcing
                        script), until a matching ":catch" has been found.
                        If the exception is not caught, the command processing
                        is terminated.
                        Example: >
                :try | throw "oops" | catch /^oo/ | echo "caught" | endtry
<                       Note that "catch" may need to be on a separate line
                        for when an error causes the parsing to skip the whole
                        line and not see the "|" that separates the commands.

                                                        *:ec* *:echo*
:ec[ho] {expr1} ..      Echoes each {expr1}, with a space in between.  The
                        first {expr1} starts on a new line.
                        Also see |:comment|.
                        Use "\n" to start a new line.  Use "\r" to move the
                        cursor to the first column.
                        Uses the highlighting set by the |:echohl| command.
                        Cannot be followed by a comment.
                        Example: >
```

```
                :echo "the value of 'shell' is" &shell
<                                                       *:echo-redraw*
                        A later redraw may make the message disappear again.
                        And since Vim mostly postpones redrawing until it's
                        finished with a sequence of commands this happens
                        quite often.  To avoid that a command from before the
                        ":echo" causes a redraw afterwards (redraws are often
                        postponed until you type something), force a redraw
                        with the |:redraw| command.  Example: >
                :new | redraw | echo "there is a new window"
<
                                                        *:echon*
:echon {expr1} ..       Echoes each {expr1}, without anything added.  Also see
                        |:comment|.
                        Uses the highlighting set by the |:echohl| command.
                        Cannot be followed by a comment.
                        Example: >
                                :echon "the value of 'shell' is " &shell
<
                        Note the difference between using ":echo", which is a
                        Vim command, and ":!echo", which is an external shell
                        command: >
                :!echo %                --> filename
<                       The arguments of ":!" are expanded, see |:_%|. >
                :!echo "%"              --> filename or "filename"
<                       Like the previous example.  Whether you see the double
                        quotes or not depends on your 'shell'. >
                :echo %                 --> nothing
<                       The '%' is an illegal character in an expression. >
                :echo "%"               --> %
<                       This just echoes the '%' character. >
                :echo expand("%")       --> filename
<                       This calls the expand() function to expand the '%'.

                                                        *:echoh* *:echohl*
:echoh[l] {name}        Use the highlight group {name} for the following
                        |:echo|, |:echon| and |:echomsg| commands.  Also used
                        for the |input()| prompt.  Example: >
                :echohl WarningMsg | echo "Don't panic!" | echohl None
<                       Don't forget to set the group back to "None",
                        otherwise all following echo's will be highlighted.

                                                        *:echom* *:echomsg*
:echom[sg] {expr1} ..   Echo the expression(s) as a true message, saving the
                        message in the |message-history|.
                        Spaces are placed between the arguments as with the
                        |:echo| command.  But unprintable characters are
                        displayed, not interpreted.
                        The parsing works slightly different from |:echo|,
                        more like |:execute|.  All the expressions are first
                        evaluated and concatenated before echoing anything.
                        The expressions must evaluate to a Number or String, a
                        Dictionary or List causes an error.
                        Uses the highlighting set by the |:echohl| command.
                        Example: >
                :echomsg "It's a Zizzer Zazzer Zuzz, as you can plainly see."
<                       See |:echo-redraw| to avoid the message disappearing
                        when the screen is redrawn.
                                                        *:echoe* *:echoerr*
:echoe[rr] {expr1} ..   Echo the expression(s) as an error message, saving the
                        message in the |message-history|.  When used in a
                        script or function the line number will be added.
```

```
                        Spaces are placed between the arguments as with the
                        :echo command.  When used inside a try conditional,
                        the message is raised as an error exception instead
                        (see |try-echoerr|).
                        Example: >
                :echoerr "This script just failed!"
<                       If you just want a highlighted message use |:echohl|.
                        And to get a beep: >
                :exe "normal \<Esc>"
<
                                                        *:exe* *:execute*
:exe[cute] {expr1} ..   Executes the string that results from the evaluation
                        of {expr1} as an Ex command.
                        Multiple arguments are concatenated, with a space in
                        between.  To avoid the extra space use the "."
                        operator to concatenate strings into one argument.
                        {expr1} is used as the processed command, command line
                        editing keys are not recognized.
                        Cannot be followed by a comment.
                        Examples: >
                :execute "buffer" nextbuf
                :execute "normal" count . "w"
<
                        ":execute" can be used to append a command to commands
                        that don't accept a '|'.  Example: >
                :execute '!ls' | echo "theend"

<                       ":execute" is also a nice way to avoid having to type
                        control characters in a Vim script for a ":normal"
                        command: >
                :execute "normal ixxx\<Esc>"
<                       This has an <Esc> character, see |expr-string|.

                        Be careful to correctly escape special characters in
                        file names.  The |fnameescape()| function can be used
                        for Vim commands, |shellescape()| for |:!| commands.
                        Examples: >
                :execute "e " . fnameescape(filename)
                :execute "!ls " . shellescape(filename, 1)
<
                        Note: The executed string may be any command-line, but
                        starting or ending "if", "while" and "for" does not
                        always work, because when commands are skipped the
                        ":execute" is not evaluated and Vim loses track of
                        where blocks start and end.  Also "break" and
                        "continue" should not be inside ":execute".
                        This example does not work, because the ":execute" is
                        not evaluated and Vim does not see the "while", and
                        gives an error for finding an ":endwhile": >
                :if 0
                : execute 'while i > 5'
                :  echo "test"
                : endwhile
                :endif
<
                        It is allowed to have a "while" or "if" command
                        completely in the executed string: >
                :execute 'while i < 5 | echo i | let i = i + 1 | endwhile'
<

                                                        *:exe-comment*
                        ":execute", ":echo" and ":echon" cannot be followed by
```

a comment directly, because they see the '"' as the
start of a string. But, you can use '|' followed by a
comment. Example: >
:echo "foo" | "this is a comment

==
8. Exception handling *exception-handling*

The Vim script language comprises an exception handling feature. This section explains how it can be used in a Vim script.

Exceptions may be raised by Vim on an error or on interrupt, see |catch-errors| and |catch-interrupt|. You can also explicitly throw an exception by using the ":throw" command, see |throw-catch|.

TRY CONDITIONALS *try-conditionals*

Exceptions can be caught or can cause cleanup code to be executed. You can use a try conditional to specify catch clauses (that catch exceptions) and/or a finally clause (to be executed for cleanup).
A try conditional begins with a |:try| command and ends at the matching |:endtry| command. In between, you can use a |:catch| command to start a catch clause, or a |:finally| command to start a finally clause. There may be none or multiple catch clauses, but there is at most one finally clause, which must not be followed by any catch clauses. The lines before the catch clauses and the finally clause is called a try block. >

```
:try
:  ...
:  ...                              TRY BLOCK
:  ...
:catch /{pattern}/
:  ...
:  ...                              CATCH CLAUSE
:  ...
:catch /{pattern}/
:  ...
:  ...                              CATCH CLAUSE
:  ...
:finally
:  ...
:  ...                              FINALLY CLAUSE
:  ...
:endtry
```

The try conditional allows to watch code for exceptions and to take the appropriate actions. Exceptions from the try block may be caught. Exceptions from the try block and also the catch clauses may cause cleanup actions.
When no exception is thrown during execution of the try block, the control is transferred to the finally clause, if present. After its execution, the script continues with the line following the ":endtry".
When an exception occurs during execution of the try block, the remaining lines in the try block are skipped. The exception is matched against the patterns specified as arguments to the ":catch" commands. The catch clause after the first matching ":catch" is taken, other catch clauses are not executed. The catch clause ends when the next ":catch", ":finally", or ":endtry" command is reached - whatever is first. Then, the finally clause (if present) is executed. When the ":endtry" is reached, the script execution continues in the following line as usual.
When an exception that does not match any of the patterns specified by the ":catch" commands is thrown in the try block, the exception is not caught by

that try conditional and none of the catch clauses is executed. Only the finally clause, if present, is taken. The exception pends during execution of the finally clause. It is resumed at the ":endtry", so that commands after the ":endtry" are not executed and the exception might be caught elsewhere, see |try-nesting|.
When during execution of a catch clause another exception is thrown, the remaining lines in that catch clause are not executed. The new exception is not matched against the patterns in any of the ":catch" commands of the same try conditional and none of its catch clauses is taken. If there is, however, a finally clause, it is executed, and the exception pends during its execution. The commands following the ":endtry" are not executed. The new exception might, however, be caught elsewhere, see |try-nesting|.
When during execution of the finally clause (if present) an exception is thrown, the remaining lines in the finally clause are skipped. If the finally clause has been taken because of an exception from the try block or one of the catch clauses, the original (pending) exception is discarded. The commands following the ":endtry" are not executed, and the exception from the finally clause is propagated and can be caught elsewhere, see |try-nesting|.

The finally clause is also executed, when a ":break" or ":continue" for a ":while" loop enclosing the complete try conditional is executed from the try block or a catch clause. Or when a ":return" or ":finish" is executed from the try block or a catch clause of a try conditional in a function or sourced script, respectively. The ":break", ":continue", ":return", or ":finish" pends during execution of the finally clause and is resumed when the ":endtry" is reached. It is, however, discarded when an exception is thrown from the finally clause.
When a ":break" or ":continue" for a ":while" loop enclosing the complete try conditional or when a ":return" or ":finish" is encountered in the finally clause, the rest of the finally clause is skipped, and the ":break", ":continue", ":return" or ":finish" is executed as usual. If the finally clause has been taken because of an exception or an earlier ":break", ":continue", ":return", or ":finish" from the try block or a catch clause, this pending exception or command is discarded.

For examples see |throw-catch| and |try-finally|.

NESTING OF TRY CONDITIONALS *try-nesting*

Try conditionals can be nested arbitrarily. That is, a complete try conditional can be put into the try block, a catch clause, or the finally clause of another try conditional. If the inner try conditional does not catch an exception thrown in its try block or throws a new exception from one of its catch clauses or its finally clause, the outer try conditional is checked according to the rules above. If the inner try conditional is in the try block of the outer try conditional, its catch clauses are checked, but otherwise only the finally clause is executed. It does not matter for nesting, whether the inner try conditional is directly contained in the outer one, or whether the outer one sources a script or calls a function containing the inner try conditional.

When none of the active try conditionals catches an exception, just their finally clauses are executed. Thereafter, the script processing terminates. An error message is displayed in case of an uncaught exception explicitly thrown by a ":throw" command. For uncaught error and interrupt exceptions implicitly raised by Vim, the error message(s) or interrupt message are shown as usual.

For examples see |throw-catch|.

```
EXAMINING EXCEPTION HANDLING CODE                       *except-examine*

Exception handling code can get tricky.  If you are in doubt what happens, set
'verbose' to 13 or use the ":13verbose" command modifier when sourcing your
script file.  Then you see when an exception is thrown, discarded, caught, or
finished.  When using a verbosity level of at least 14, things pending in
a finally clause are also shown.  This information is also given in debug mode
(see |debug-scripts|).


THROWING AND CATCHING EXCEPTIONS                        *throw-catch*

You can throw any number or string as an exception.  Use the |:throw| command
and pass the value to be thrown as argument: >
        :throw 4711
        :throw "string"
<                                                       *throw-expression*
You can also specify an expression argument.  The expression is then evaluated
first, and the result is thrown: >
        :throw 4705 + strlen("string")
        :throw strpart("strings", 0, 6)

An exception might be thrown during evaluation of the argument of the ":throw"
command.  Unless it is caught there, the expression evaluation is abandoned.
The ":throw" command then does not throw a new exception.
   Example: >

        :function! Foo(arg)
        :  try
        :    throw a:arg
        :  catch /foo/
        :  endtry
        :  return 1
        :endfunction
        :
        :function! Bar()
        :  echo "in Bar"
        :  return 4710
        :endfunction
        :
        :throw Foo("arrgh") + Bar()

This throws "arrgh", and "in Bar" is not displayed since Bar() is not
executed. >
        :throw Foo("foo") + Bar()
however displays "in Bar" and throws 4711.

Any other command that takes an expression as argument might also be
abandoned by an (uncaught) exception during the expression evaluation.  The
exception is then propagated to the caller of the command.
   Example: >

        :if Foo("arrgh")
        :  echo "then"
        :else
        :  echo "else"
        :endif

Here neither of "then" or "else" is displayed.

                                                        *catch-order*
Exceptions can be caught by a try conditional with one or more |:catch|
```

```
commands, see |try-conditionals|.   The values to be caught by each ":catch"
command can be specified as a pattern argument.  The subsequent catch clause
gets executed when a matching exception is caught.
   Example: >

        :function! Foo(value)
        :  try
        :    throw a:value
        :  catch /^\d\+$/
        :    echo "Number thrown"
        :  catch /.*/
        :    echo "String thrown"
        :  endtry
        :endfunction
        :
        :call Foo(0x1267)
        :call Foo('string')

The first call to Foo() displays "Number thrown", the second "String thrown".
An exception is matched against the ":catch" commands in the order they are
specified.  Only the first match counts.  So you should place the more
specific ":catch" first.  The following order does not make sense: >

        :  catch /.*/
        :    echo "String thrown"
        :  catch /^\d\+$/
        :    echo "Number thrown"

The first ":catch" here matches always, so that the second catch clause is
never taken.

                                                              *throw-variables*
If you catch an exception by a general pattern, you may access the exact value
in the variable |v:exception|: >

        :  catch /^\d\+$/
        :    echo "Number thrown.  Value is" v:exception

You may also be interested where an exception was thrown.  This is stored in
|v:throwpoint|.  Note that "v:exception" and "v:throwpoint" are valid for the
exception most recently caught as long it is not finished.
   Example: >

        :function! Caught()
        :  if v:exception != ""
        :    echo 'Caught "' . v:exception . '" in ' . v:throwpoint
        :  else
        :    echo 'Nothing caught'
        :  endif
        :endfunction
        :
        :function! Foo()
        :  try
        :    try
        :      try
        :        throw 4711
        :      finally
        :        call Caught()
        :      endtry
        :    catch /.*/
        :      call Caught()
        :      throw "oops"
```

```
        :    endtry
        :  catch /.*/
        :    call Caught()
        :  finally
        :    call Caught()
        :  endtry
        :endfunction
        :
        :call Foo()

This displays >

        Nothing caught
        Caught "4711" in function Foo, line 4
        Caught "oops" in function Foo, line 10
        Nothing caught

A practical example:  The following command ":LineNumber" displays the line
number in the script or function where it has been used: >

        :function! LineNumber()
        :    return substitute(v:throwpoint, '.*\D\(\d\+\).*', '\1', "")
        :endfunction
        :command! LineNumber try | throw "" | catch | echo LineNumber() | endtry
<
                                                        *try-nested*
An exception that is not caught by a try conditional can be caught by
a surrounding try conditional: >

        :try
        :  try
        :    throw "foo"
        :  catch /foobar/
        :    echo "foobar"
        :  finally
        :    echo "inner finally"
        :  endtry
        :catch /foo/
        :  echo "foo"
        :endtry

The inner try conditional does not catch the exception, just its finally
clause is executed.  The exception is then caught by the outer try
conditional.  The example displays "inner finally" and then "foo".

                                                        *throw-from-catch*
You can catch an exception and throw a new one to be caught elsewhere from the
catch clause: >

        :function! Foo()
        :  throw "foo"
        :endfunction
        :
        :function! Bar()
        :  try
        :    call Foo()
        :  catch /foo/
        :    echo "Caught foo, throw bar"
        :    throw "bar"
        :  endtry
        :endfunction
        :
```

```
        :try
        :  call Bar()
        :catch /.*/
        :  echo "Caught" v:exception
        :endtry
```

This displays "Caught foo, throw bar" and then "Caught bar".

rethrow

There is no real rethrow in the Vim script language, but you may throw "v:exception" instead: >

```
        :function! Bar()
        :  try
        :    call Foo()
        :  catch /.*/
        :    echo "Rethrow" v:exception
        :    throw v:exception
        :  endtry
        :endfunction
```

<

try-echoerr

Note that this method cannot be used to "rethrow" Vim error or interrupt exceptions, because it is not possible to fake Vim internal exceptions. Trying so causes an error exception. You should throw your own exception denoting the situation. If you want to cause a Vim error exception containing the original error exception value, you can use the |:echoerr| command: >

```
        :try
        :  try
        :    asdf
        :  catch /.*/
        :    echoerr v:exception
        :  endtry
        :catch /.*/
        :  echo v:exception
        :endtry
```

This code displays

```
        Vim(echoerr):Vim:E492: Not an editor command:   asdf ~
```

CLEANUP CODE

try-finally

Scripts often change global settings and restore them at their end. If the user however interrupts the script by pressing CTRL-C, the settings remain in an inconsistent state. The same may happen to you in the development phase of a script when an error occurs or you explicitly throw an exception without catching it. You can solve these problems by using a try conditional with a finally clause for restoring the settings. Its execution is guaranteed on normal control flow, on error, on an explicit ":throw", and on interrupt. (Note that errors and interrupts from inside the try conditional are converted to exceptions. When not caught, they terminate the script after the finally clause has been executed.)
Example: >

```
        :try
        :  let s:saved_ts = &ts
        :  set ts=17
        :
        :  " Do the hard work here.
        :
```

```
        :finally
        :  let &ts = s:saved_ts
        :  unlet s:saved_ts
        :endtry

This method should be used locally whenever a function or part of a script
changes global settings which need to be restored on failure or normal exit of
that function or script part.

                                                          *break-finally*
Cleanup code works also when the try block or a catch clause is left by
a ":continue", ":break", ":return", or ":finish".
   Example: >

        :let first = 1
        :while 1
        :  try
        :    if first
        :      echo "first"
        :      let first = 0
        :      continue
        :    else
        :      throw "second"
        :    endif
        :  catch /.*/
        :    echo v:exception
        :    break
        :  finally
        :    echo "cleanup"
        :  endtry
        :  echo "still in while"
        :endwhile
        :echo "end"

This displays "first", "cleanup", "second", "cleanup", and "end". >

        :function! Foo()
        :  try
        :    return 4711
        :  finally
        :    echo "cleanup\n"
        :  endtry
        :  echo "Foo still active"
        :endfunction
        :
        :echo Foo() "returned by Foo"

This displays "cleanup" and "4711 returned by Foo".  You don't need to add an
extra ":return" in the finally clause.  (Above all, this would override the
return value.)

                                                          *except-from-finally*
Using either of ":continue", ":break", ":return", ":finish", or ":throw" in
a finally clause is possible, but not recommended since it abandons the
cleanup actions for the try conditional.  But, of course, interrupt and error
exceptions might get raised from a finally clause.
   Example where an error in the finally clause stops an interrupt from
working correctly: >

        :try
        :  try
        :    echo "Press CTRL-C for interrupt"
```

```
        :    while 1
        :    endwhile
        :  finally
        :    unlet novar
        :  endtry
        :catch /novar/
        :endtry
        :echo "Script still running"
        :sleep 1
```

If you need to put commands that could fail into a finally clause, you should think about catching or ignoring the errors in these commands, see |catch-errors| and |ignore-errors|.

CATCHING ERRORS *catch-errors*

If you want to catch specific errors, you just have to put the code to be watched in a try block and add a catch clause for the error message. The presence of the try conditional causes all errors to be converted to an exception. No message is displayed and |v:errmsg| is not set then. To find the right pattern for the ":catch" command, you have to know how the format of the error exception is.
Error exceptions have the following format: >

```
        Vim({cmdname}):{errmsg}
or >
        Vim:{errmsg}
```

{cmdname} is the name of the command that failed; the second form is used when the command name is not known. {errmsg} is the error message usually produced when the error occurs outside try conditionals. It always begins with a capital "E", followed by a two or three-digit error number, a colon, and a space.

Examples:

The command >

```
        :unlet novar
```

normally produces the error message >

```
        E108: No such variable: "novar"
```

which is converted inside try conditionals to an exception >

```
        Vim(unlet):E108: No such variable: "novar"
```

The command >

```
        :dwim
```

normally produces the error message >

```
        E492: Not an editor command: dwim
```

which is converted inside try conditionals to an exception >

```
        Vim:E492: Not an editor command: dwim
```

You can catch all ":unlet" errors by a >

```
        :catch /^Vim(unlet):/
```

or all errors for misspelled command names by a >

```
        :catch /^Vim:E492:/
```

Some error messages may be produced by different commands: >

```
        :function nofunc
```

and >

```
        :delfunction nofunc
```

both produce the error message >

```
        E128: Function name must start with a capital: nofunc
```

```
which is converted inside try conditionals to an exception >
        Vim(function):E128: Function name must start with a capital: nofunc
or >
        Vim(delfunction):E128: Function name must start with a capital: nofunc
respectively.  You can catch the error by its number independently on the
command that caused it if you use the following pattern: >
        :catch /^Vim(\a\+):E128:/

Some commands like >
        :let x = novar
produce multiple error messages, here: >
        E121: Undefined variable: novar
        E15: Invalid expression:  novar
Only the first is used for the exception value, since it is the most specific
one (see |except-several-errors|).  So you can catch it by >
        :catch /^Vim(\a\+):E121:/

You can catch all errors related to the name "nofunc" by >
        :catch /\<nofunc\>/

You can catch all Vim errors in the ":write" and ":read" commands by >
        :catch /^Vim(\(write\|read\)):E\d\+:/

You can catch all Vim errors by the pattern >
        :catch /^Vim\((\a\+)\)\=:E\d\+:/
<
                                                        *catch-text*
NOTE: You should never catch the error message text itself: >
        :catch /No such variable/
only works in the English locale, but not when the user has selected
a different language by the |:language| command.  It is however helpful to
cite the message text in a comment: >
        :catch /^Vim(\a\+):E108:/   " No such variable


IGNORING ERRORS                                         *ignore-errors*

You can ignore errors in a specific Vim command by catching them locally: >

        :try
        :  write
        :catch
        :endtry

But you are strongly recommended NOT to use this simple form, since it could
catch more than you want.  With the ":write" command, some autocommands could
be executed and cause errors not related to writing, for instance: >

        :au BufWritePre * unlet novar

There could even be such errors you are not responsible for as a script
writer: a user of your script might have defined such autocommands.  You would
then hide the error from the user.
   It is much better to use >

        :try
        :  write
        :catch /^Vim(write):/
        :endtry

which only catches real write errors.  So catch only what you'd like to ignore
intentionally.
```

For a single command that does not cause execution of autocommands, you could
even suppress the conversion of errors to exceptions by the ":silent!"
command: >
        :silent! nunmap k
This works also when a try conditional is active.

CATCHING INTERRUPTS *catch-interrupt*

When there are active try conditionals, an interrupt (CTRL-C) is converted to
the exception "Vim:Interrupt". You can catch it like every exception. The
script is not terminated, then.
   Example: >

```
        :function! TASK1()
        :  sleep 10
        :endfunction

        :function! TASK2()
        :  sleep 20
        :endfunction

        :while 1
        :  let command = input("Type a command: ")
        :  try
        :    if command == ""
        :      continue
        :    elseif command == "END"
        :      break
        :    elseif command == "TASK1"
        :      call TASK1()
        :    elseif command == "TASK2"
        :      call TASK2()
        :    else
        :      echo "\nIllegal command:" command
        :      continue
        :    endif
        :  catch /^Vim:Interrupt$/
        :    echo "\nCommand interrupted"
        :    " Caught the interrupt.  Continue with next prompt.
        :  endtry
        :endwhile
```

You can interrupt a task here by pressing CTRL-C; the script then asks for
a new command. If you press CTRL-C at the prompt, the script is terminated.

For testing what happens when CTRL-C would be pressed on a specific line in
your script, use the debug mode and execute the |>quit| or |>interrupt|
command on that line. See |debug-scripts|.

CATCHING ALL *catch-all*

The commands >

```
        :catch /.*/
        :catch //
        :catch
```

catch everything, error exceptions, interrupt exceptions and exceptions
explicitly thrown by the |:throw| command. This is useful at the top level of

```
a script in order to catch unexpected things.
   Example: >

        :try
        :
        :  " do the hard work here
        :
        :catch /MyException/
        :
        :  " handle known problem
        :
        :catch /^Vim:Interrupt$/
        :    echo "Script interrupted"
        :catch /.*/
        :  echo "Internal error (" . v:exception . ")"
        :  echo " - occurred at " . v:throwpoint
        :endtry
        :" end of script

Note: Catching all might catch more things than you want.  Thus, you are
strongly encouraged to catch only for problems that you can really handle by
specifying a pattern argument to the ":catch".
   Example: Catching all could make it nearly impossible to interrupt a script
by pressing CTRL-C: >

        :while 1
        :  try
        :    sleep 1
        :  catch
        :  endtry
        :endwhile


EXCEPTIONS AND AUTOCOMMANDS                                *except-autocmd*

Exceptions may be used during execution of autocommands.  Example: >

        :autocmd User x try
        :autocmd User x   throw "Oops!"
        :autocmd User x catch
        :autocmd User x   echo v:exception
        :autocmd User x endtry
        :autocmd User x throw "Arrgh!"
        :autocmd User x echo "Should not be displayed"
        :
        :try
        :  doautocmd User x
        :catch
        :  echo v:exception
        :endtry

This displays "Oops!" and "Arrgh!".

                                                           *except-autocmd-Pre*
For some commands, autocommands get executed before the main action of the
command takes place.  If an exception is thrown and not caught in the sequence
of autocommands, the sequence and the command that caused its execution are
abandoned and the exception is propagated to the caller of the command.
   Example: >

        :autocmd BufWritePre * throw "FAIL"
        :autocmd BufWritePre * echo "Should not be displayed"
```

```
        :
        :try
        :  write
        :catch
        :  echo "Caught:" v:exception "from" v:throwpoint
        :endtry
```

Here, the ":write" command does not write the file currently being edited (as you can see by checking 'modified'), since the exception from the BufWritePre autocommand abandons the ":write". The exception is then caught and the script displays: >

```
        Caught: FAIL from BufWrite Auto commands for "*"
<
```

except-autocmd-Post

For some commands, autocommands get executed after the main action of the command has taken place. If this main action fails and the command is inside an active try conditional, the autocommands are skipped and an error exception is thrown that can be caught by the caller of the command.
Example: >

```
        :autocmd BufWritePost * echo "File successfully written!"
        :
        :try
        :  write /i/m/p/o/s/s/i/b/l/e
        :catch
        :  echo v:exception
        :endtry
```

This just displays: >

```
        Vim(write):E212: Can't open file for writing (/i/m/p/o/s/s/i/b/l/e)
```

If you really need to execute the autocommands even when the main action fails, trigger the event from the catch clause.
Example: >

```
        :autocmd BufWritePre  * set noreadonly
        :autocmd BufWritePost * set readonly
        :
        :try
        :  write /i/m/p/o/s/s/i/b/l/e
        :catch
        :  doautocmd BufWritePost /i/m/p/o/s/s/i/b/l/e
        :endtry
<
```

You can also use ":silent!": >

```
        :let x = "ok"
        :let v:errmsg = ""
        :autocmd BufWritePost * if v:errmsg != ""
        :autocmd BufWritePost *   let x = "after fail"
        :autocmd BufWritePost * endif
        :try
        :  silent! write /i/m/p/o/s/s/i/b/l/e
        :catch
        :endtry
        :echo x
```

This displays "after fail".

If the main action of the command does not fail, exceptions from the

```
autocommands will be catchable by the caller of the command:  >

        :autocmd BufWritePost * throw ":-("
        :autocmd BufWritePost * echo "Should not be displayed"
        :
        :try
        :  write
        :catch
        :  echo v:exception
        :endtry
<
                                                        *except-autocmd-Cmd*
For some commands, the normal action can be replaced by a sequence of
autocommands.  Exceptions from that sequence will be catchable by the caller
of the command.
   Example:  For the ":write" command, the caller cannot know whether the file
had actually been written when the exception occurred.  You need to tell it in
some way. >

        :if !exists("cnt")
        :  let cnt = 0
        :
        :  autocmd BufWriteCmd * if &modified
        :  autocmd BufWriteCmd *   let cnt = cnt + 1
        :  autocmd BufWriteCmd *   if cnt % 3 == 2
        :  autocmd BufWriteCmd *     throw "BufWriteCmdError"
        :  autocmd BufWriteCmd *   endif
        :  autocmd BufWriteCmd *   write | set nomodified
        :  autocmd BufWriteCmd *   if cnt % 3 == 0
        :  autocmd BufWriteCmd *     throw "BufWriteCmdError"
        :  autocmd BufWriteCmd *   endif
        :  autocmd BufWriteCmd *   echo "File successfully written!"
        :  autocmd BufWriteCmd * endif
        :endif
        :
        :try
        :       write
        :catch /^BufWriteCmdError$/
        :  if &modified
        :    echo "Error on writing (file contents not changed)"
        :  else
        :    echo "Error after writing"
        :  endif
        :catch /^Vim(write):/
        :    echo "Error on writing"
        :endtry

When this script is sourced several times after making changes, it displays
first >
        File successfully written!
then >
        Error on writing (file contents not changed)
then >
        Error after writing
etc.

                                                        *except-autocmd-ill*
You cannot spread a try conditional over autocommands for different events.
The following code is ill-formed: >

        :autocmd BufWritePre  * try
        :
```

```
        :autocmd BufWritePost * catch
        :autocmd BufWritePost *   echo v:exception
        :autocmd BufWritePost * endtry
        :
        :write
```

EXCEPTION HIERARCHIES AND PARAMETERIZED EXCEPTIONS *except-hier-param*

Some programming languages allow to use hierarchies of exception classes or to pass additional information with the object of an exception class. You can do similar things in Vim.

In order to throw an exception from a hierarchy, just throw the complete class name with the components separated by a colon, for instance throw the string "EXCEPT:MATHERR:OVERFLOW" for an overflow in a mathematical library.

When you want to pass additional information with your exception class, add it in parentheses, for instance throw the string "EXCEPT:IO:WRITEERR(myfile)" for an error when writing "myfile".

With the appropriate patterns in the ":catch" command, you can catch for base classes or derived classes of your hierarchy. Additional information in parentheses can be cut out from |v:exception| with the ":substitute" command.

Example: >

```
        :function! CheckRange(a, func)
        :  if a:a < 0
        :    throw "EXCEPT:MATHERR:RANGE(" . a:func . ")"
        :  endif
        :endfunction
        :
        :function! Add(a, b)
        :  call CheckRange(a:a, "Add")
        :  call CheckRange(a:b, "Add")
        :  let c = a:a + a:b
        :  if c < 0
        :    throw "EXCEPT:MATHERR:OVERFLOW"
        :  endif
        :  return c
        :endfunction
        :
        :function! Div(a, b)
        :  call CheckRange(a:a, "Div")
        :  call CheckRange(a:b, "Div")
        :  if (a:b == 0)
        :    throw "EXCEPT:MATHERR:ZERODIV"
        :  endif
        :  return a:a / a:b
        :endfunction
        :
        :function! Write(file)
        :  try
        :    execute "write" fnameescape(a:file)
        :  catch /^Vim(write):/
        :    throw "EXCEPT:IO(" . getcwd() . ", " . a:file . "):WRITEERR"
        :  endtry
        :endfunction
        :
        :try
        :
        :  " something with arithmetics and I/O
        :
        :catch /^EXCEPT:MATHERR:RANGE/
        :  let function = substitute(v:exception, '.*(\(\a\+\)).*', '\1', "")
```

```
        :  echo "Range error in" function
        :
        :catch /^EXCEPT:MATHERR/        " catches OVERFLOW and ZERODIV
        :  echo "Math error"
        :
        :catch /^EXCEPT:IO/
        :  let dir = substitute(v:exception, '.*(\(.\+\),\s*.\+).*', '\1', "")
        :  let file = substitute(v:exception, '.*(.\+,\s*\(.\+\)).*', '\1', "")
        :  if file !~ '^/'
        :    let file = dir . "/" . file
        :  endif
        :  echo 'I/O error for "' . file . '"'
        :
        :catch /^EXCEPT/
        :  echo "Unspecified error"
        :
        :endtry
```

The exceptions raised by Vim itself (on error or when pressing CTRL-C) use a flat hierarchy: they are all in the "Vim" class. You cannot throw yourself exceptions with the "Vim" prefix; they are reserved for Vim.

Vim error exceptions are parameterized with the name of the command that failed, if known. See |catch-errors|.

PECULIARITIES

except-compat

The exception handling concept requires that the command sequence causing the exception is aborted immediately and control is transferred to finally clauses and/or a catch clause.

In the Vim script language there are cases where scripts and functions continue after an error: in functions without the "abort" flag or in a command after ":silent!", control flow goes to the following line, and outside functions, control flow goes to the line following the outermost ":endwhile" or ":endif". On the other hand, errors should be catchable as exceptions (thus, requiring the immediate abortion).

This problem has been solved by converting errors to exceptions and using immediate abortion (if not suppressed by ":silent!") only when a try conditional is active. This is no restriction since an (error) exception can be caught only from an active try conditional. If you want an immediate termination without catching the error, just use a try conditional without catch clause. (You can cause cleanup code being executed before termination by specifying a finally clause.)

When no try conditional is active, the usual abortion and continuation behavior is used instead of immediate abortion. This ensures compatibility of scripts written for Vim 6.1 and earlier.

However, when sourcing an existing script that does not use exception handling commands (or when calling one of its functions) from inside an active try conditional of a new script, you might change the control flow of the existing script on error. You get the immediate abortion on error and can catch the error in the new script. If however the sourced script suppresses error messages by using the ":silent!" command (checking for errors by testing |v:errmsg| if appropriate), its execution path is not changed. The error is not converted to an exception. (See |:silent|.) So the only remaining cause where this happens is for scripts that don't care about errors and produce error messages. You probably won't want to use such code from your new scripts.

except-syntax-err
Syntax errors in the exception handling commands are never caught by any of
the ":catch" commands of the try conditional they belong to. Its finally
clauses, however, is executed.
   Example: >

```
        :try
        :  try
        :    throw 4711
        :  catch /\(/
        :    echo "in catch with syntax error"
        :  catch
        :    echo "inner catch-all"
        :  finally
        :    echo "inner finally"
        :  endtry
        :catch
        :  echo 'outer catch-all caught "' . v:exception . '"'
        :  finally
        :    echo "outer finally"
        :endtry
```

This displays: >
```
    inner finally
    outer catch-all caught "Vim(catch):E54: Unmatched \("
    outer finally
```
The original exception is discarded and an error exception is raised, instead.

except-single-line
The ":try", ":catch", ":finally", and ":endtry" commands can be put on
a single line, but then syntax errors may make it difficult to recognize the
"catch" line, thus you better avoid this.
   Example: >
```
        :try | unlet! foo # | catch | endtry
```
raises an error exception for the trailing characters after the ":unlet!"
argument, but does not see the ":catch" and ":endtry" commands, so that the
error exception is discarded and the "E488: Trailing characters" message gets
displayed.

except-several-errors
When several errors appear in a single command, the first error message is
usually the most specific one and therefor converted to the error exception.
   Example: >
```
        echo novar
```
causes >
```
        E121: Undefined variable: novar
        E15: Invalid expression: novar
```
The value of the error exception inside try conditionals is: >
```
        Vim(echo):E121: Undefined variable: novar
```
<
except-syntax-error
But when a syntax error is detected after a normal error in the same command,
the syntax error is used for the exception being thrown.
   Example: >
```
        unlet novar #
```
causes >
```
        E108: No such variable: "novar"
        E488: Trailing characters
```
The value of the error exception inside try conditionals is: >
```
        Vim(unlet):E488: Trailing characters
```
This is done because the syntax error might change the execution path in a way
not intended by the user. Example: >
```
        try
```

```
            try | unlet novar # | catch | echo v:exception | endtry
        catch /.*/
            echo "outer catch:" v:exception
        endtry
This displays "outer catch: Vim(unlet):E488: Trailing characters", and then
a "E600: Missing :endtry" error message is given, see |except-single-line|.

==============================================================================
9. Examples                                                *eval-examples*

Printing in Binary ~
>
  :" The function Nr2Bin() returns the binary string representation of a number.
  :func Nr2Bin(nr)
  :  let n = a:nr
  :  let r = ""
  :  while n
  :    let r = '01'[n % 2] . r
  :    let n = n / 2
  :  endwhile
  :  return r
  :endfunc

  :" The function String2Bin() converts each character in a string to a
  :" binary string, separated with dashes.
  :func String2Bin(str)
  :  let out = ''
  :  for ix in range(strlen(a:str))
  :    let out = out . '-' . Nr2Bin(char2nr(a:str[ix]))
  :  endfor
  :  return out[1:]
  :endfunc

Example of its use: >
  :echo Nr2Bin(32)
result: "100000" >
  :echo String2Bin("32")
result: "110011-110010"


Sorting lines ~

This example sorts lines with a specific compare function. >

  :func SortBuffer()
  :  let lines = getline(1, '$')
  :  call sort(lines, function("Strcmp"))
  :  call setline(1, lines)
  :endfunction

As a one-liner: >
  :call setline(1, sort(getline(1, '$'), function("Strcmp")))


scanf() replacement ~
                                                           *sscanf*
There is no sscanf() function in Vim.  If you need to extract parts from a
line, you can use matchstr() and substitute() to do it.  This example shows
how to get the file name, line number and column number out of a line like
"foobar.txt, 123, 45". >
   :" Set up the match bit
   :let mx='\(\f\+\),\s*\(\d\+\),\s*\(\d\+\)'
```

```
   :"get the part matching the whole expression
   :let l = matchstr(line, mx)
   :"get each item out of the match
   :let file = substitute(l, mx, '\1', '')
   :let lnum = substitute(l, mx, '\2', '')
   :let col = substitute(l, mx, '\3', '')

The input is in the variable "line", the results in the variables "file",
"lnum" and "col". (idea from Michael Geddes)


getting the scriptnames in a Dictionary ~
                                                    *scriptnames-dictionary*
The |:scriptnames| command can be used to get a list of all script files that
have been sourced.  There is no equivalent function or variable for this
(because it's rarely needed).  In case you need to manipulate the list this
code can be used: >
    " Get the output of ":scriptnames" in the scriptnames_output variable.
    let scriptnames_output = ''
    redir => scriptnames_output
    silent scriptnames
    redir END

    " Split the output into lines and parse each line.  Add an entry to the
    " "scripts" dictionary.
    let scripts = {}
    for line in split(scriptnames_output, "\n")
      " Only do non-blank lines.
      if line =~ '\S'
        " Get the first number in the line.
        let nr = matchstr(line, '\d\+')
        " Get the file name, remove the script number " 123: ".
        let name = substitute(line, '.\+:\s*', '', '')
        " Add an item to the Dictionary
        let scripts[nr] = name
      endif
    endfor
    unlet scriptnames_output

==============================================================================
10. No +eval feature                                *no-eval-feature*

When the |+eval| feature was disabled at compile time, none of the expression
evaluation commands are available.  To prevent this from causing Vim scripts
to generate all kinds of errors, the ":if" and ":endif" commands are still
recognized, though the argument of the ":if" and everything between the ":if"
and the matching ":endif" is ignored.  Nesting of ":if" blocks is allowed, but
only if the commands are at the start of the line.  The ":else" command is not
recognized.

Example of how to avoid executing commands when the |+eval| feature is
missing: >

        :if 1
        :  echo "Expression evaluation is compiled in"
        :else
        :  echo "You will _never_ see this message"
        :endif

To execute a command only when the |+eval| feature is disabled requires a trick,
as this example shows: >
```

```
        silent! while 0
          set history=111
        silent! endwhile

When the |+eval| feature is available the command is skipped because of the
"while 0".  Without the |+eval| feature the "while 0" is an error, which is
silently ignored, and the command is executed.

==============================================================================
11. The sandbox                                    *eval-sandbox* *sandbox* *E48*

The 'foldexpr', 'formatexpr', 'includeexpr', 'indentexpr', 'statusline' and
'foldtext' options may be evaluated in a sandbox.  This means that you are
protected from these expressions having nasty side effects.  This gives some
safety for when these options are set from a modeline.  It is also used when
the command from a tags file is executed and for CTRL-R = in the command line.
The sandbox is also used for the |:sandbox| command.

These items are not allowed in the sandbox:
        - changing the buffer text
        - defining or changing mapping, autocommands, functions, user commands
        - setting certain options (see |option-summary|)
        - setting certain v: variables (see |v:var|)  *E794*
        - executing a shell command
        - reading or writing a file
        - jumping to another buffer or editing a file
        - executing Python, Perl, etc. commands
This is not guaranteed 100% secure, but it should block most attacks.

                                                        *:san* *:sandbox*
:san[dbox] {cmd}        Execute {cmd} in the sandbox.  Useful to evaluate an
                        option that may have been set from a modeline, e.g.
                        'foldexpr'.

                                                        *sandbox-option*
A few options contain an expression.  When this expression is evaluated it may
have to be done in the sandbox to avoid a security risk.  But the sandbox is
restrictive, thus this only happens when the option was set from an insecure
location.  Insecure in this context are:
- sourcing a .vimrc or .exrc in the current directory
- while executing in the sandbox
- value coming from a modeline

Note that when in the sandbox and saving an option value and restoring it, the
option will still be marked as it was set in the sandbox.

==============================================================================
12. Textlock                                                    *textlock*

In a few situations it is not allowed to change the text in the buffer, jump
to another window and some other things that might confuse or break what Vim
is currently doing.  This mostly applies to things that happen when Vim is
actually doing something else.  For example, evaluating the 'balloonexpr' may
happen any moment the mouse cursor is resting at some position.

This is not allowed when the textlock is active:
        - changing the buffer text
        - jumping to another buffer or window
        - editing another file
        - closing a window or quitting Vim
        - etc.
```

```
==============================================================================
13. Testing                                                     *testing*

Vim can be tested after building it, usually with "make test".
The tests are located in the directory "src/testdir".

There are several types of tests added over time:
        test33.in               oldest, don't add any more
        test_something.in       old style tests
        test_something.vim      new style tests

                                                *new-style-testing*
New tests should be added as new style tests.  These use functions such as
|assert_equal()| to keep the test commands and the expected result in one
place.
                                                *old-style-testing*
In some cases an old style test needs to be used.  E.g. when testing Vim
without the |+eval| feature.

Find more information in the file src/testdir/README.txt.


 vim:tw=78:ts=8:ft=help:norl:
*channel.txt*      For Vim version 8.0.  Last change: 2017 Aug 11


                  VIM REFERENCE MANUAL    by Bram Moolenaar


                      Inter-process communication               *channel*

Vim uses channels to communicate with other processes.
A channel uses a socket or pipes.                       *socket-interface*
Jobs can be used to start processes and communicate with them.
The Netbeans interface also uses a channel. |netbeans|

1. Overview                             |job-channel-overview|
2. Channel demo                         |channel-demo|
3. Opening a channel                    |channel-open|
4. Using a JSON or JS channel           |channel-use|
5. Channel commands                     |channel-commands|
6. Using a RAW or NL channel            |channel-raw|
7. More channel functions               |channel-more|
8. Starting a job with a channel        |job-start|
9. Starting a job without a channel     |job-start-nochannel|
10. Job options                         |job-options|
11. Controlling a job                   |job-control|

{Vi does not have any of these features}
{only when compiled with the |+channel| feature for channel stuff}
        You can check this with: `has('channel')`
{only when compiled with the |+job| feature for job stuff}
        You can check this with: `has('job')`

==============================================================================
1. Overview                                             *job-channel-overview*

There are four main types of jobs:
1. A daemon, serving several Vim instances.
   Vim connects to it with a socket.
2. One job working with one Vim instance, asynchronously.
   Uses a socket or pipes.
```

```
3. A job performing some work for a short time, asynchronously.
   Uses a socket or pipes.
4. Running a filter, synchronously.
   Uses pipes.

For when using sockets See |job-start|, |job-start-nochannel| and
|channel-open|.  For 2 and 3, one or more jobs using pipes, see |job-start|.
For 4 use the ":{range}!cmd" command, see |filter|.

Over the socket and pipes these protocols are available:
RAW     nothing known, Vim cannot tell where a message ends
NL      every message ends in a NL (newline) character
JSON    JSON encoding |json_encode()|
JS      JavaScript style JSON-like encoding |js_encode()|

Common combination are:
- Using a job connected through pipes in NL mode.  E.g., to run a style
  checker and receive errors and warnings.
- Using a deamon, connecting over a socket in JSON mode.  E.g. to lookup
  cross-references in a database.

==============================================================================
2. Channel demo                             *channel-demo* *demoserver.py*

This requires Python.  The demo program can be found in
$VIMRUNTIME/tools/demoserver.py
Run it in one terminal.  We will call this T1.

Run Vim in another terminal.  Connect to the demo server with: >
        let channel = ch_open('localhost:8765')

In T1 you should see:
        === socket opened === ~

You can now send a message to the server: >
        echo ch_evalexpr(channel, 'hello!')

The message is received in T1 and a response is sent back to Vim.
You can see the raw messages in T1.  What Vim sends is:
        [1,"hello!"] ~
And the response is:
        [1,"got it"] ~
The number will increase every time you send a message.

The server can send a command to Vim.  Type this on T1 (literally, including
the quotes):
        ["ex","echo 'hi there'"] ~
And you should see the message in Vim. You can move the cursor a word forward:
        ["normal","w"] ~

To handle asynchronous communication a callback needs to be used: >
        func MyHandler(channel, msg)
          echo "from the handler: " . a:msg
        endfunc
        call ch_sendexpr(channel, 'hello!', {'callback': "MyHandler"})
Vim will not wait for a response.  Now the server can send the response later
and MyHandler will be invoked.

Instead of giving a callback with every send call, it can also be specified
when opening the channel: >
        call ch_close(channel)
        let channel = ch_open('localhost:8765', {'callback': "MyHandler"})
```

```
        call ch_sendexpr(channel, 'hello!')

When trying out channels it's useful to see what is going on.  You can tell
Vim to write lines in log file: >
        call ch_logfile('channellog', 'w')
See |ch_logfile()|.

==============================================================================
3. Opening a channel                                    *channel-open*

To open a channel: >
    let channel = ch_open({address} [, {options}])
    if ch_status(channel) == "open"
      " use the channel

Use |ch_status()| to see if the channel could be opened.

{address} has the form "hostname:port".  E.g., "localhost:8765".

{options} is a dictionary with optional entries:        *channel-open-options*

"mode" can be:                                          *channel-mode*
        "json" - Use JSON, see below; most convenient way. Default.
        "js"   - Use JS (JavaScript) encoding, more efficient than JSON.
        "nl"   - Use messages that end in a NL character
        "raw"  - Use raw messages
                                                *channel-callback* *E921*
"callback"      A function that is called when a message is received that is
                not handled otherwise.  It gets two arguments: the channel
                and the received message. Example: >
        func Handle(channel, msg)
          echo 'Received: ' . a:msg
        endfunc
        let channel = ch_open("localhost:8765", {"callback": "Handle"})
<
                When "mode" is "json" or "js" the "msg" argument is the body
                of the received message, converted to Vim types.
                When "mode" is "nl" the "msg" argument is one message,
                excluding the NL.
                When "mode" is "raw" the "msg" argument is the whole message
                as a string.

                For all callbacks: Use |function()| to bind it to arguments
                and/or a Dictionary.  Or use the form "dict.function" to bind
                the Dictionary.

                Callbacks are only called at a "safe" moment, usually when Vim
                is waiting for the user to type a character.  Vim does not use
                multi-threading.

                                                        *close_cb*
"close_cb"      A function that is called when the channel gets closed, other
                than by calling ch_close().  It should be defined like this: >
        func MyCloseHandler(channel)
<               Vim will invoke callbacks that handle data before invoking
                close_cb, thus when this function is called no more data will
                be passed to the callbacks.
                                                        *channel-drop*
"drop"          Specifies when to drop messages:
                    "auto"      When there is no callback to handle a message.
                                The "close_cb" is also considered for this.
                    "never"     All messages will be kept.
```

waittime
"waittime" The time to wait for the connection to be made in milliseconds. A negative number waits forever.

The default is zero, don't wait, which is useful if a local server is supposed to be running already. On Unix Vim actually uses a 1 msec timeout, that is required on many systems. Use a larger value for a remote server, e.g. 10 msec at least.

channel-timeout
"timeout" The time to wait for a request when blocking, E.g. when using ch_evalexpr(). In milliseconds. The default is 2000 (2 seconds).

When "mode" is "json" or "js" the "callback" is optional. When omitted it is only possible to receive a message after sending one.

To change the channel options after opening it use |ch_setoptions()|. The arguments are similar to what is passed to |ch_open()|, but "waittime" cannot be given, since that only applies to opening the channel.

For example, the handler can be added or changed: >
    call ch_setoptions(channel, {'callback': callback})
When "callback" is empty (zero or an empty string) the handler is removed.

After a callback has been invoked Vim will update the screen and put the cursor back where it belongs. Thus the callback should not need to do `:redraw`.

The timeout can be changed: >
    call ch_setoptions(channel, {'timeout': msec})
<
channel-close *E906*
Once done with the channel, disconnect it like this: >
    call ch_close(channel)
When a socket is used this will close the socket for both directions. When pipes are used (stdin/stdout/stderr) they are all closed. This might not be what you want! Stopping the job with job_stop() might be better.
All readahead is discarded, callbacks will no longer be invoked.

Note that a channel is closed in three stages:
- The I/O ends, log message: "Closing channel". There can still be queued messages to read or callbacks to invoke.
- The readahead is cleared, log message: "Clearing channel". Some variables may still reference the channel.
- The channel is freed, log message: "Freeing channel".

When the channel can't be opened you will get an error message. There is a difference between MS-Windows and Unix: On Unix when the port doesn't exist ch_open() fails quickly. On MS-Windows "waittime" applies.
E898 *E901* *E902*

If there is an error reading or writing a channel it will be closed.
E630 *E631*

==
4. Using a JSON or JS channel *channel-use*

If mode is JSON then a message can be sent synchronously like this: >
    let response = ch_evalexpr(channel, {expr})
This awaits a response from the other side.

When mode is JS this works the same, except that the messages use
JavaScript encoding. See |js_encode()| for the difference.

To send a message, without handling a response or letting the channel callback
handle the response: >
    call ch_sendexpr(channel, {expr})

To send a message and letting the response handled by a specific function,
asynchronously: >
    call ch_sendexpr(channel, {expr}, {'callback': Handler})

Vim will match the response with the request using the message ID. Once the
response is received the callback will be invoked. Further responses with the
same ID will be ignored. If your server sends back multiple responses you
need to send them with ID zero, they will be passed to the channel callback.

The {expr} is converted to JSON and wrapped in an array. An example of the
message that the receiver will get when {expr} is the string "hello":
        [12,"hello"] ~

The format of the JSON sent is:
    [{number},{expr}]

In which {number} is different every time. It must be used in the response
(if any):

    [{number},{response}]

This way Vim knows which sent message matches with which received message and
can call the right handler. Also when the messages arrive out of order.

A newline character is terminating the JSON text. This can be used to
separate the read text. For example, in Python:
        splitidx = read_text.find('\n')
        message = read_text[:splitidx]
        rest = read_text[splitidx + 1:]

The sender must always send valid JSON to Vim. Vim can check for the end of
the message by parsing the JSON. It will only accept the message if the end
was received. A newline after the message is optional.

When the process wants to send a message to Vim without first receiving a
message, it must use the number zero:
    [0,{response}]

Then channel handler will then get {response} converted to Vim types. If the
channel does not have a handler the message is dropped.

It is also possible to use ch_sendraw() and ch_evalraw() on a JSON or JS
channel. The caller is then completely responsible for correct encoding and
decoding.

==
5. Channel commands *channel-commands*

With a JSON channel the process can send commands to Vim that will be
handled by Vim internally, it does not require a handler for the channel.

Possible commands are: *E903* *E904* *E905*
    ["redraw", {forced}]
    ["ex", {Ex command}]

```
    ["normal", {Normal mode command}]
    ["expr",   {expression}, {number}]
    ["expr",   {expression}]
    ["call",   {func name}, {argument list}, {number}]
    ["call",   {func name}, {argument list}]

With all of these: Be careful what these commands do!  You can easily
interfere with what the user is doing.  To avoid trouble use |mode()| to check
that the editor is in the expected state.  E.g., to send keys that must be
inserted as text, not executed as a command:
    ["ex","if mode() == 'i' | call feedkeys('ClassName') | endif"] ~

Errors in these commands are normally not reported to avoid them messing up
the display.  If you do want to see them, set the 'verbose' option to 3 or
higher.


Command "redraw" ~

The other commands do not update the screen, so that you can send a sequence
of commands without the cursor moving around.  You must end with the "redraw"
command to show any changed text and show the cursor where it belongs.

The argument is normally an empty string:
        ["redraw", ""] ~
To first clear the screen pass "force":
        ["redraw", "force"] ~


Command "ex" ~

The "ex" command is executed as any Ex command.  There is no response for
completion or error.  You could use functions in an |autoload| script:
        ["ex","call myscript#MyFunc(arg)"]

You can also use "call |feedkeys()|" to insert any key sequence.

When there is an error a message is written to the channel log, if it exists,
and v:errmsg is set to the error.


Command "normal" ~

The "normal" command is executed like with ":normal!", commands are not
mapped.  Example to open the folds under the cursor:
        ["normal" "zO"]


Command "expr"  with response ~

The "expr" command can be used to get the result of an expression.  For
example, to get the number of lines in the current buffer:
        ["expr","line('$')", -2] ~

It will send back the result of the expression:
        [-2, "last line"] ~
The format is:
        [{number}, {result}]

Here {number} is the same as what was in the request.  Use a negative number
to avoid confusion with message that Vim sends.  Use a different number on
every request to be able to match the request with the response.
```

{result} is the result of the evaluation and is JSON encoded. If the
evaluation fails or the result can't be encoded in JSON it is the string
"ERROR".

Command "expr" without a response ~

This command is similar to "expr" above, but does not send back any response.
Example:
        ["expr","setline('$', ['one', 'two', 'three'])"] ~
There is no third argument in the request.

Command "call" ~

This is similar to "expr", but instead of passing the whole expression as a
string this passes the name of a function and a list of arguments. This
avoids the conversion of the arguments to a string and escaping and
concatenating them. Example:
        ["call", "line", ["$"], -2] ~

Leave out the fourth argument if no response is to be sent:
        ["call", "setline", ["$", ["one", "two", "three"]]] ~

==
6. Using a RAW or NL channel *channel-raw*

If mode is RAW or NL then a message can be sent like this: >
    let response = ch_evalraw(channel, {string})

The {string} is sent as-is. The response will be what can be read from the
channel right away. Since Vim doesn't know how to recognize the end of the
message you need to take care of it yourself. The timeout applies for reading
the first byte, after that it will not wait for anything more.

If mode is "nl" you can send a message in a similar way. You are expected
to put in the NL after each message. Thus you can also send several messages
ending in a NL at once. The response will be the text up to and including the
first NL. This can also be just the NL for an empty response.
If no NL was read before the channel timeout an empty string is returned.

To send a message, without expecting a response: >
    call ch_sendraw(channel, {string})
The process can send back a response, the channel handler will be called with
it.

To send a message and letting the response handled by a specific function,
asynchronously: >
    call ch_sendraw(channel, {string}, {'callback': 'MyHandler'})

This {string} can also be JSON, use |json_encode()| to create it and
|json_decode()| to handle a received JSON message.

It is not possible to use |ch_evalexpr()| or |ch_sendexpr()| on a raw channel.

A String in Vim cannot contain NUL bytes. To send or receive NUL bytes read
or write from a buffer. See |in_io-buffer| and |out_io-buffer|.

==
7. More channel functions *channel-more*

```
To obtain the status of a channel: ch_status(channel).  The possible results
are:
        "fail"          Failed to open the channel.
        "open"          The channel can be used.
        "buffered"      The channel was closed but there is data to read.
        "closed"        The channel was closed.

To obtain the job associated with a channel: ch_getjob(channel)

To read one message from a channel: >
        let output = ch_read(channel)
This uses the channel timeout.  To read without a timeout, just get any
message that is available: >
        let output = ch_read(channel, {'timeout': 0})
When no message was available then the result is v:none for a JSON or JS mode
channels, an empty string for a RAW or NL channel.  You can use |ch_canread()|
to check if there is something to read.

Note that when there is no callback, messages are dropped.  To avoid that add
a close callback to the channel.

To read all output from a RAW channel that is available: >
        let output = ch_readraw(channel)
To read the error output: >
        let output = ch_readraw(channel, {"part": "err"})

ch_read() and ch_readraw() use the channel timeout.  When there is nothing to
read within that time an empty string is returned.  To specify a different
timeout in msec use the "timeout" option:
        {"timeout": 123} ~
To read from the error output use the "part" option:
        {"part": "err"} ~
To read a message with a specific ID, on a JS or JSON channel:
        {"id": 99} ~
When no ID is specified or the ID is -1, the first message is returned. This
overrules any callback waiting for this message.

For a RAW channel this returns whatever is available, since Vim does not know
where a message ends.
For a NL channel this returns one message.
For a JS or JSON channel this returns one decoded message.
This includes any sequence number.

==============================================================================
8. Starting a job with a channel                        *job-start* *job*

To start a job and open a channel for stdin/stdout/stderr: >
    let job = job_start(command, {options})

You can get the channel with: >
    let channel = job_getchannel(job)

The channel will use NL mode.  If you want another mode it's best to specify
this in {options}.  When changing the mode later some text may have already
been received and not parsed correctly.

If the command produces a line of output that you want to deal with, specify
a handler for stdout: >
    let job = job_start(command, {"out_cb": "MyHandler"})
The function will be called with the channel and a message. You would define
it like this: >
    func MyHandler(channel, msg)
```

```
Without the handler you need to read the output with |ch_read()| or
|ch_readraw()|. You can do this in the close callback, see |read-in-close-cb|.

Note that if the job exits before you read the output, the output may be lost.
This depends on the system (on Unix this happens because closing the write end
of a pipe causes the read end to get EOF).  To avoid this make the job sleep
for a short while before it exits.

The handler defined for "out_cb" will not receive stderr.  If you want to
handle that separately, add an "err_cb" handler: >
    let job = job_start(command, {"out_cb": "MyHandler",
            \                     "err_cb": "ErrHandler"})

If you want to handle both stderr and stdout with one handler use the
"callback" option: >
    let job = job_start(command, {"callback": "MyHandler"})

Depending on the system, starting a job can put Vim in the background, the
started job gets the focus.  To avoid that, use the `foreground()` function.
This might not always work when called early, put in the callback handler or
use a timer to call it after the job has started.

You can send a message to the command with ch_evalraw().  If the channel is in
JSON or JS mode you can use ch_evalexpr().

There are several options you can use, see |job-options|.
For example, to start a job and write its output in buffer "dummy": >
        let logjob = job_start("tail -f /tmp/log",
                               \ {'out_io': 'buffer', 'out_name': 'dummy'})
        sbuf dummy


Job input from a buffer ~
                                                        *in_io-buffer*
To run a job that reads from a buffer: >
        let job = job_start({command},
            \ {'in_io': 'buffer', 'in_name': 'mybuffer'})
<
                                                        *E915* *E918*
The buffer is found by name, similar to |bufnr()|. The buffer must exist and
be loaded when job_start() is called.

By default this reads the whole buffer.  This can be changed with the "in_top"
and "in_bot" options.

A special mode is when "in_top" is set to zero and "in_bot" is not set: Every
time a line is added to the buffer, the last-but-one line will be sent to the
job stdin.  This allows for editing the last line and sending it when pressing
Enter.
                                                        *channel-close-in*
When not using the special mode the pipe or socket will be closed after the
last line has been written.  This signals the reading end that the input
finished.  You can also use |ch_close_in()| to close it sooner.

NUL bytes in the text will be passed to the job (internally Vim stores these
as NL bytes).


Reading job output in the close callback ~
                                                        *read-in-close-cb*
If the job can take some time and you don't need intermediate results, you can
```

```
add a close callback and read the output there: >

        func! CloseHandler(channel)
          while ch_status(a:channel, {'part': 'out'}) == 'buffered'
            echomsg ch_read(a:channel)
          endwhile
        endfunc
        let job = job_start(command, {'close_cb': 'CloseHandler'})

You will want to do something more useful than "echomsg".

==============================================================================
9. Starting a job without a channel                     *job-start-nochannel*

To start another process without creating a channel: >
    let job = job_start(command,
        \ {"in_io": "null", "out_io": "null", "err_io": "null"})

This starts {command} in the background, Vim does not wait for it to finish.

When Vim sees that neither stdin, stdout or stderr are connected, no channel
will be created.  Often you will want to include redirection in the command to
avoid it getting stuck.

There are several options you can use, see |job-options|.

                                                        *job-start-if-needed*
To start a job only when connecting to an address does not work, do something
like this: >
        let channel = ch_open(address, {"waittime": 0})
        if ch_status(channel) == "fail"
          let job = job_start(command)
          let channel = ch_open(address, {"waittime": 1000})
        endif

Note that the waittime for ch_open() gives the job one second to make the port
available.

==============================================================================
10. Job options                                         *job-options*

The {options} argument in job_start() is a dictionary.  All entries are
optional.  Some options can be used after the job has started, using
job_setoptions(job, {options}).  Many options can be used with the channel
related to the job, using ch_setoptions(channel, {options}).
See |job_setoptions()| and |ch_setoptions()|.

                                                *in_mode* *out_mode* *err_mode*
"in_mode"               mode specifically for stdin, only when using pipes
"out_mode"              mode specifically for stdout, only when using pipes
"err_mode"              mode specifically for stderr, only when using pipes
                        See |channel-mode| for the values.

                        Note: when setting "mode" the part specific mode is
                        overwritten.  Therefore set "mode" first and the part
                        specific mode later.

                        Note: when writing to a file or buffer and when
                        reading from a buffer NL mode is used by default.

                                                *job-callback*
"callback": handler     Callback for something to read on any part of the
```

```
                        channel.
                                                *job-out_cb* *out_cb*
"out_cb": handler       Callback for when there is something to read on
                        stdout.  Only for when the channel uses pipes.  When
                        "out_cb" wasn't set the channel callback is used.
                        The two arguments are the channel and the message.

                                                *job-err_cb* *err_cb*
"err_cb": handler       Callback for when there is something to read on
                        stderr.  Only for when the channel uses pipes.  When
                        "err_cb" wasn't set the channel callback is used.
                        The two arguments are the channel and the message.
                                                *job-close_cb*
"close_cb": handler     Callback for when the channel is closed.  Same as
                        "close_cb" on |ch_open()|, see |close_cb|.
                                                *job-drop*
"drop": when            Specifies when to drop messages.  Same as "drop" on
                        |ch_open()|, see |channel-drop|.  For "auto" the
                        exit_cb is not considered.
                                                *job-exit_cb*
"exit_cb": handler      Callback for when the job ends.  The arguments are the
                        job and the exit status.
                        Vim checks up to 10 times per second for jobs that
                        ended.  The check can also be triggered by calling
                        |job_status()|, which may then invoke the exit_cb
                        handler.
                        Note that data can be buffered, callbacks may still be
                        called after the process ends.
                                                        *job-timeout*
"timeout": time         The time to wait for a request when blocking, E.g.
                        when using ch_evalexpr().  In milliseconds.  The
                        default is 2000 (2 seconds).
                                                *out_timeout* *err_timeout*
"out_timeout": time     Timeout for stdout.  Only when using pipes.
"err_timeout": time     Timeout for stderr.  Only when using pipes.
                        Note: when setting "timeout" the part specific mode is
                        overwritten.  Therefore set "timeout" first and the
                        part specific mode later.

                                                *job-stoponexit*
"stoponexit": {signal}  Send {signal} to the job when Vim exits.  See
                        |job_stop()| for possible values.
"stoponexit": ""        Do not stop the job when Vim exits.
                        The default is "term".

                                                *job-term*
"term": "open"          Start a terminal in a new window and connect the job
                        stdin/stdout/stderr to it.  Similar to using
                        `:terminal`.
                        NOTE: Not implemented yet!

"channel": {channel}    Use an existing channel instead of creating a new one.
                        The parts of the channel that get used for the new job
                        will be disconnected from what they were used before.
                        If the channel was still used by another job this may
                        cause I/O errors.
                        Existing callbacks and other settings remain.

"pty": 1                Use a pty (pseudo-tty) instead of a pipe when
                        possible.  This is most useful in combination with a
                        terminal window, see |terminal|.
                        {only on Unix and Unix-like systems}
```

```
                                *job-in_io* *in_top* *in_bot* *in_name* *in_buf*
"in_io": "null"         disconnect stdin (read from /dev/null)
"in_io": "pipe"         stdin is connected to the channel (default)
"in_io": "file"         stdin reads from a file
"in_io": "buffer"       stdin reads from a buffer
"in_top": number        when using "buffer": first line to send (default: 1)
"in_bot": number        when using "buffer": last line to send (default: last)
"in_name": "/path/file" the name of the file or buffer to read from
"in_buf": number        the number of the buffer to read from

                                *job-out_io* *out_name* *out_buf*
"out_io": "null"        disconnect stdout (goes to /dev/null)
"out_io": "pipe"        stdout is connected to the channel (default)
"out_io": "file"        stdout writes to a file
"out_io": "buffer"      stdout appends to a buffer (see below)
"out_name": "/path/file" the name of the file or buffer to write to
"out_buf": number       the number of the buffer to write to
"out_modifiable": 0     when writing to a buffer, 'modifiable' will be off
                        (see below)
"out_msg": 0            when writing to a new buffer, the first line will be
                        set to "Reading from channel output..."

                                *job-err_io* *err_name* *err_buf*
"err_io": "out"         stderr messages to go to stdout
"err_io": "null"        disconnect stderr  (goes to /dev/null)
"err_io": "pipe"        stderr is connected to the channel (default)
"err_io": "file"        stderr writes to a file
"err_io": "buffer"      stderr appends to a buffer (see below)
"err_name": "/path/file" the name of the file or buffer to write to
"err_buf": number       the number of the buffer to write to
"err_modifiable": 0     when writing to a buffer, 'modifiable' will be off
                        (see below)
"err_msg": 0            when writing to a new buffer, the first line will be
                        set to "Reading from channel error..."

"block_write": number   only for testing: pretend every other write to stdin
                        will block

"env": dict             environment variables for the new process
"cwd": "/path/to/dir"   current working directory for the new process;
                        if the directory does not exist an error is given


Writing to a buffer ~
                                                        *out_io-buffer*
When the out_io or err_io mode is "buffer" and there is a callback, the text
is appended to the buffer before invoking the callback.

When a buffer is used both for input and output, the output lines are put
above the last line, since the last line is what is written to the channel
input.  Otherwise lines are appended below the last line.

When using JS or JSON mode with "buffer", only messages with zero or negative
ID will be added to the buffer, after decoding + encoding.  Messages with a
positive number will be handled by a callback, commands are handled as usual.

The name of the buffer from "out_name" or "err_name" is compared the full name
of existing buffers, also after expanding the name for the current directory.
E.g., when a buffer was created with ":edit somename" and the buffer name is
"somename" it will use that buffer.
```

If there is no matching buffer a new buffer is created. Use an empty name to always create a new buffer. |ch_getbufnr()| can then be used to get the buffer number.

For a new buffer 'buftype' is set to "nofile" and 'bufhidden' to "hide". If you prefer other settings, create the buffer first and pass the buffer number.
out_modifiable *err_modifiable*
The "out_modifiable" and "err_modifiable" options can be used to set the 'modifiable' option off, or write to a buffer that has 'modifiable' off. That means that lines will be appended to the buffer, but the user can't easily change the buffer.
out_msg *err_msg*
The "out_msg" option can be used to specify whether a new buffer will have the first line set to "Reading from channel output...". The default is to add the message. "err_msg" does the same for channel error.

When an existing buffer is to be written where 'modifiable' is off and the "out_modifiable" or "err_modifiable" options is not zero, an error is given and the buffer will not be written to.

When the buffer written to is displayed in a window and the cursor is in the first column of the last line, the cursor will be moved to the newly added line and the window is scrolled up to show the cursor if needed.

Undo is synced for every added line. NUL bytes are accepted (internally Vim stores these as NL bytes).

Writing to a file ~
E920
The file is created with permissions 600 (read-write for the user, not accessible for others). Use |setfperm()| to change this.

If the file already exists it is truncated.

==
11. Controlling a job *job-control*

To get the status of a job: >
```
echo job_status(job)
```

To make a job stop running: >
```
job_stop(job)
```

This is the normal way to end a job. On Unix it sends a SIGTERM to the job. It is possible to use other ways to stop the job, or even send arbitrary signals. E.g. to force a job to stop, "kill it": >
```
job_stop(job, "kill")
```

For more options see |job_stop()|.

vim:tw=78:ts=8:ft=help:norl:
fold.txt For Vim version 8.0. Last change: 2017 Mar 18

VIM REFERENCE MANUAL by Bram Moolenaar

Folding *Folding* *folding* *folds*

You can find an introduction on folding in chapter 28 of the user manual.

|usr_28.txt|

{Vi has no Folding}
{not available when compiled without the |+folding| feature}

==
1. Fold methods   *fold-methods*

The folding method can be set with the 'foldmethod' option.

When setting 'foldmethod' to a value other than "manual", all folds are
deleted and new ones created. Switching to the "manual" method doesn't remove
the existing folds. This can be used to first define the folds automatically
and then change them manually.

There are six methods to select folds:
        manual   manually define folds
        indent   more indent means a higher fold level
        expr   specify an expression to define folds
        syntax   folds defined by syntax highlighting
        diff   folds for unchanged text
        marker   folds defined by markers in the text

MANUAL   *fold-manual*

Use commands to manually define the fold regions. This can also be used by a
script that parses text to find folds.

The level of a fold is only defined by its nesting. To increase the fold
level of a fold for a range of lines, define a fold inside it that has the
same lines.

The manual folds are lost when you abandon the file. To save the folds use
the |:mkview| command. The view can be restored later with |:loadview|.

INDENT   *fold-indent*

The folds are automatically defined by the indent of the lines.

The foldlevel is computed from the indent of the line, divided by the
'shiftwidth' (rounded down). A sequence of lines with the same or higher fold
level form a fold, with the lines with a higher level forming a nested fold.

The nesting of folds is limited with 'foldnestmax'.

Some lines are ignored and get the fold level of the line above or below it,
whichever is lower. These are empty or white lines and lines starting
with a character in 'foldignore'. White space is skipped before checking for
characters in 'foldignore'. For C use "#" to ignore preprocessor lines.

When you want to ignore lines in another way, use the "expr" method. The
|indent()| function can be used in 'foldexpr' to get the indent of a line.

EXPR   *fold-expr*

```
The folds are automatically defined by their foldlevel, like with the "indent"
method.  The value of the 'foldexpr' option is evaluated to get the foldlevel
of a line.  Examples:
This will create a fold for all consecutive lines that start with a tab: >
        :set foldexpr=getline(v:lnum)[0]==\"\\t\"
This will call a function to compute the fold level: >
        :set foldexpr=MyFoldLevel(v:lnum)
This will make a fold out of paragraphs separated by blank lines: >
        :set foldexpr=getline(v:lnum)=~'^\\s*$'&&getline(v:lnum+1)=~'\\S'?'<1':1
This does the same: >
        :set foldexpr=getline(v:lnum-1)=~'^\\s*$'&&getline(v:lnum)=~'\\S'?'>1':1

Note that backslashes must be used to escape characters that ":set" handles
differently (space, backslash, double quote, etc., see |option-backslash|).

These are the conditions with which the expression is evaluated:
- The current buffer and window are set for the line.
- The variable "v:lnum" is set to the line number.
- The result is used for the fold level in this way:
  value                 meaning ~
  0                     the line is not in a fold
  1, 2, ..              the line is in a fold with this level
  -1                    the fold level is undefined, use the fold level of a
                        line before or after this line, whichever is the
                        lowest.
  "="                   use fold level from the previous line
  "a1", "a2", ..        add one, two, .. to the fold level of the previous
                        line, use the result for the current line
  "s1", "s2", ..        subtract one, two, .. from the fold level of the
                        previous line, use the result for the next line
  "<1", "<2", ..        a fold with this level ends at this line
  ">1", ">2", ..        a fold with this level starts at this line

It is not required to mark the start (end) of a fold with ">1" ("<1"), a fold
will also start (end) when the fold level is higher (lower) than the fold
level of the previous line.

There must be no side effects from the expression.  The text in the buffer,
cursor position, the search patterns, options etc. must not be changed.
You can change and restore them if you are careful.

If there is some error in the expression, or the resulting value isn't
recognized, there is no error message and the fold level will be zero.
For debugging the 'debug' option can be set to "msg", the error messages will
be visible then.

Note: Since the expression has to be evaluated for every line, this fold
method can be very slow!

Try to avoid the "=", "a" and "s" return values, since Vim often has to search
backwards for a line for which the fold level is defined.  This can be slow.

An example of using "a1" and "s1": For a multi-line C comment, a line
containing "/*" would return "a1" to start a fold, and a line containing "*/"
would return "s1" to end the fold after that line: >
  if match(thisline, '/\*') >= 0
    return 'a1'
  elseif match(thisline, '\*/') >= 0
    return 's1'
  else
    return '='
```

```
  endif
```
However, this won't work for single line comments, strings, etc.

|foldlevel()| can be useful to compute a fold level relative to a previous fold level. But note that foldlevel() may return -1 if the level is not known yet. And it returns the level at the start of the line, while a fold might end in that line.

It may happen that folds are not updated properly. You can use |zx| or |zX| to force updating folds.

SYNTAX *fold-syntax*

A fold is defined by syntax items that have the "fold" argument. |:syn-fold|

The fold level is defined by nesting folds. The nesting of folds is limited with 'foldnestmax'.

Be careful to specify proper syntax syncing. If this is not done right, folds may differ from the displayed highlighting. This is especially relevant when using patterns that match more than one line. In case of doubt, try using brute-force syncing: >
```
:syn sync fromstart
```

DIFF *fold-diff*

The folds are automatically defined for text that is not part of a change or close to a change.

This method only works properly when the 'diff' option is set for the current window and changes are being displayed. Otherwise the whole buffer will be one big fold.

The 'diffopt' option can be used to specify the context. That is, the number of lines between the fold and a change that are not included in the fold. For example, to use a context of 8 lines: >
```
:set diffopt=filler,context:8
```
The default context is six lines.

When 'scrollbind' is also set, Vim will attempt to keep the same folds open in other diff windows, so that the same text is visible.

MARKER *fold-marker*

Markers in the text tell where folds start and end. This allows you to precisely specify the folds. This will allow deleting and putting a fold, without the risk of including the wrong lines. The 'foldtext' option is normally set such that the text before the marker shows up in the folded line. This makes it possible to give a name to the fold.

Markers can have a level included, or can use matching pairs. Including a level is easier, you don't have to add end markers and avoid problems with non-matching marker pairs. Example: >
```
/* global variables {{{1 */
int varA, varB;

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

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

A fold starts at a "{{{" marker. The following number specifies the fold level. What happens depends on the difference between the current fold level and the level given by the marker:
1. If a marker with the same fold level is encountered, the previous fold ends and another fold with the same level starts.
2. If a marker with a higher fold level is found, a nested fold is started.
3. If a marker with a lower fold level is found, all folds up to and including this level end and a fold with the specified level starts.

The number indicates the fold level. A zero cannot be used (a marker with level zero is ignored). You can use "}}}" with a digit to indicate the level of the fold that ends. The fold level of the following line will be one less than the indicated level. Note that Vim doesn't look back to the level of the matching marker (that would take too much time). Example: >

```
        {{{1
        fold level here is 1
        {{{3
        fold level here is 3
        }}}3
        fold level here is 2
```

You can also use matching pairs of "{{{" and "}}}" markers to define folds. Each "{{{" increases the fold level by one, each "}}}" decreases the fold level by one. Be careful to keep the markers matching! Example: >

```
        {{{
        fold level here is 1
        {{{
        fold level here is 2
        }}}
        fold level here is 1
```

You can mix using markers with a number and without a number. A useful way of doing this is to use numbered markers for large folds, and unnumbered markers locally in a function. For example use level one folds for the sections of your file like "structure definitions", "local variables" and "functions". Use level 2 markers for each definition and function, Use unnumbered markers inside functions. When you make changes in a function to split up folds, you don't have to renumber the markers.

The markers can be set with the 'foldmarker' option. It is recommended to keep this at the default value of "{{{,}}}", so that files can be exchanged between Vim users. Only change it when it is required for the file (e.g., it contains markers from another folding editor, or the default markers cause trouble for the language of the file).

fold-create-marker

"zf" can be used to create a fold defined by markers. Vim will insert the markers for you. Vim will append the start and end marker, as specified with 'foldmarker'. The markers are appended to the end of the line. 'commentstring' is used if it isn't empty.
This does not work properly when:
- The line already contains a marker with a level number. Vim then doesn't know what to do.
- Folds nearby use a level number in their marker which gets in the way.
- The line is inside a comment, 'commentstring' isn't empty and nested comments don't work. For example with C: adding /* {{{ */ inside a comment

  will truncate the existing comment. Either put the marker before or after
  the comment, or add the marker manually.
Generally it's not a good idea to let Vim create markers when you already have
markers with a level number.

                                                        fold-delete-marker
"zd" can be used to delete a fold defined by markers. Vim will delete the
markers for you. Vim will search for the start and end markers, as specified
with 'foldmarker', at the start and end of the fold. When the text around the
marker matches with 'commentstring', that text is deleted as well.
This does not work properly when:
- A line contains more than one marker and one of them specifies a level.
  Only the first one is removed, without checking if this will have the
  desired effect of deleting the fold.
- The marker contains a level number and is used to start or end several folds
  at the same time.

==
2. Fold commands   *fold-commands* *E490*

All folding commands start with "z". Hint: the "z" looks like a folded piece
of paper, if you look at it from the side.

CREATING AND DELETING FOLDS ~
                                                        zf *E350*
zf{motion} or
{Visual}zf   Operator to create a fold.
                This only works when 'foldmethod' is "manual" or "marker".
                The new fold will be closed for the "manual" method.
                'foldenable' will be set.
                Also see |fold-create-marker|.

                                                        zF
zF   Create a fold for [count] lines. Works like "zf".

:{range}fo[ld]   *:fold* *:fo*
                Create a fold for the lines in {range}. Works like "zf".

                                                        zd *E351*
zd   Delete one fold at the cursor. When the cursor is on a folded
                line, that fold is deleted. Nested folds are moved one level
                up. In Visual mode one level of all folds (partially) in the
                selected area are deleted.
                Careful: This easily deletes more folds than you expect and
                there is no undo for manual folding.
                This only works when 'foldmethod' is "manual" or "marker".
                Also see |fold-delete-marker|.

                                                        zD
zD   Delete folds recursively at the cursor. In Visual mode all
                folds (partially) in the selected area and all nested folds in
                them are deleted.
                This only works when 'foldmethod' is "manual" or "marker".
                Also see |fold-delete-marker|.

                                                        zE *E352*
zE   Eliminate all folds in the window.
                This only works when 'foldmethod' is "manual" or "marker".
                Also see |fold-delete-marker|.

OPENING AND CLOSING FOLDS ~

A fold smaller than 'foldminlines' will always be displayed like it was open.
Therefore the commands below may work differently on small folds.

zo
zo Open one fold under the cursor. When a count is given, that many folds deep will be opened. In Visual mode one level of folds is opened for all lines in the selected area.

zO
zO Open all folds under the cursor recursively. Folds that don't contain the cursor line are unchanged.
In Visual mode it opens all folds that are in the selected area, also those that are only partly selected.

zc
zc Close one fold under the cursor. When a count is given, that many folds deep are closed. In Visual mode one level of folds is closed for all lines in the selected area.
'foldenable' will be set.

zC
zC Close all folds under the cursor recursively. Folds that don't contain the cursor line are unchanged.
In Visual mode it closes all folds that are in the selected area, also those that are only partly selected.
'foldenable' will be set.

za
za When on a closed fold: open it. When folds are nested, you may have to use "za" several times. When a count is given, that many closed folds are opened.
When on an open fold: close it and set 'foldenable'. This will only close one level, since using "za" again will open the fold. When a count is given that many folds will be closed (that's not the same as repeating "za" that many times).

zA
zA When on a closed fold: open it recursively.
When on an open fold: close it recursively and set 'foldenable'.

zv
zv View cursor line: Open just enough folds to make the line in which the cursor is located not folded.

zx
zx Update folds: Undo manually opened and closed folds: re-apply 'foldlevel', then do "zv": View cursor line.
Also forces recomputing folds. This is useful when using 'foldexpr' and the buffer is changed in a way that results in folds not to be updated properly.

zX
zX Undo manually opened and closed folds: re-apply 'foldlevel'.
Also forces recomputing folds, like |zx|.

zm
zm Fold more: Subtract |v:count1| from 'foldlevel'. If 'foldlevel' was already zero nothing happens.

'foldenable' will be set.

zM
zM Close all folds: set 'foldlevel' to 0.
'foldenable' will be set.

zr
zr Reduce folding: Add |v:count1| to 'foldlevel'.

zR
zR Open all folds. This sets 'foldlevel' to highest fold level.

:foldo *:foldopen*
:{range}foldo[pen][!]
Open folds in {range}. When [!] is added all folds are
opened. Useful to see all the text in {range}. Without [!]
one level of folds is opened.

:foldc *:foldclose*
:{range}foldc[lose][!]
Close folds in {range}. When [!] is added all folds are
closed. Useful to hide all the text in {range}. Without [!]
one level of folds is closed.

zn
zn Fold none: reset 'foldenable'. All folds will be open.

zN
zN Fold normal: set 'foldenable'. All folds will be as they
were before.

zi
zi Invert 'foldenable'.

MOVING OVER FOLDS ~

[z
[z Move to the start of the current open fold. If already at the
start, move to the start of the fold that contains it. If
there is no containing fold, the command fails.
When a count is used, repeats the command [count] times.

]z
]z Move to the end of the current open fold. If already at the
end, move to the end of the fold that contains it. If there
is no containing fold, the command fails.
When a count is used, repeats the command [count] times.

zj
zj Move downwards to the start of the next fold. A closed fold
is counted as one fold.
When a count is used, repeats the command [count] times.
This command can be used after an |operator|.

zk
zk Move upwards to the end of the previous fold. A closed fold
is counted as one fold.
When a count is used, repeats the command [count] times.
This command can be used after an |operator|.

EXECUTING COMMANDS ON FOLDS ~

```
:[range]foldd[oopen] {cmd}                      *:foldd* *:folddoopen*
                Execute {cmd} on all lines that are not in a closed fold.
                When [range] is given, only these lines are used.
                Each time {cmd} is executed the cursor is positioned on the
                line it is executed for.
                This works like the ":global" command: First all lines that
                are not in a closed fold are marked.  Then the {cmd} is
                executed for all marked lines.  Thus when {cmd} changes the
                folds, this has no influence on where it is executed (except
                when lines are deleted, of course).
                Example: >
                        :folddoopen s/end/loop_end/ge
<               Note the use of the "e" flag to avoid getting an error message
                where "end" doesn't match.

:[range]folddoc[losed] {cmd}                    *:folddoc* *:folddoclosed*
                Execute {cmd} on all lines that are in a closed fold.
                Otherwise like ":folddoopen".

==============================================================================
3. Fold options                                 *fold-options*

COLORS                                                  *fold-colors*

The colors of a closed fold are set with the Folded group |hl-Folded|.  The
colors of the fold column are set with the FoldColumn group |hl-FoldColumn|.
Example to set the colors: >

        :highlight Folded guibg=grey guifg=blue
        :highlight FoldColumn guibg=darkgrey guifg=white


FOLDLEVEL                                               *fold-foldlevel*

'foldlevel' is a number option: The higher the more folded regions are open.
When 'foldlevel' is 0, all folds are closed.
When 'foldlevel' is positive, some folds are closed.
When 'foldlevel' is very high, all folds are open.
'foldlevel' is applied when it is changed.  After that manually folds can be
opened and closed.
When increased, folds above the new level are opened.  No manually opened
folds will be closed.
When decreased, folds above the new level are closed.  No manually closed
folds will be opened.


FOLDTEXT                                                *fold-foldtext*

'foldtext' is a string option that specifies an expression.  This expression
is evaluated to obtain the text displayed for a closed fold.  Example: >

    :set foldtext=v:folddashes.substitute(getline(v:foldstart),'/\\*\\\|\\*/\\\|{{{\
\d\\=','','g')

This shows the first line of the fold, with "/*", "*/" and "{{{" removed.
Note the use of backslashes to avoid some characters to be interpreted by the
":set" command.  It's simpler to define a function and call that: >

    :set foldtext=MyFoldText()
    :function MyFoldText()
    :  let line = getline(v:foldstart)
```

```
    :  let sub = substitute(line, '/\*\|\*/\|{{{\d\=', '', 'g')
    :  return v:folddashes . sub
    :endfunction
```

Evaluating 'foldtext' is done in the |sandbox|. The current window is set to the window that displays the line. Errors are ignored.

The default value is |foldtext()|. This returns a reasonable text for most types of folding. If you don't like it, you can specify your own 'foldtext' expression. It can use these special Vim variables:

v:foldstart	line number of first line in the fold
v:foldend	line number of last line in the fold
v:folddashes	a string that contains dashes to represent the foldlevel.
v:foldlevel	the foldlevel of the fold

In the result a TAB is replaced with a space and unprintable characters are made into printable characters.

The resulting line is truncated to fit in the window, it never wraps.
When there is room after the text, it is filled with the character specified by 'fillchars'.

Note that backslashes need to be used for characters that the ":set" command handles differently: Space, backslash and double-quote. |option-backslash|

FOLDCOLUMN *fold-foldcolumn*

'foldcolumn' is a number, which sets the width for a column on the side of the window to indicate folds. When it is zero, there is no foldcolumn. A normal value is 4 or 5. The minimal useful value is 2, although 1 still provides some information. The maximum is 12.

An open fold is indicated with a column that has a '-' at the top and '|' characters below it. This column stops where the open fold stops. When folds nest, the nested fold is one character right of the fold it's contained in.

A closed fold is indicated with a '+'.

Where the fold column is too narrow to display all nested folds, digits are shown to indicate the nesting level.

The mouse can also be used to open and close folds by clicking in the fold column:
- Click on a '+' to open the closed fold at this row.
- Click on any other non-blank character to close the open fold at this row.

OTHER OPTIONS

'foldenable'	'fen':	Open all folds while not set.
'foldexpr'	'fde':	Expression used for "expr" folding.
'foldignore'	'fdi':	Characters used for "indent" folding.
'foldmarker'	'fmr':	Defined markers used for "marker" folding.
'foldmethod'	'fdm':	Name of the current folding method.
'foldminlines'	'fml':	Minimum number of screen lines for a fold to be displayed closed.
'foldnestmax'	'fdn':	Maximum nesting for "indent" and "syntax" folding.
'foldopen'	'fdo':	Which kinds of commands open closed folds.
'foldclose'	'fcl':	When the folds not under the cursor are closed.

==
4. Behavior of folds   *fold-behavior*

When moving the cursor upwards or downwards and when scrolling, the cursor
will move to the first line of a sequence of folded lines. When the cursor is
already on a folded line, it moves to the next unfolded line or the next
closed fold.

While the cursor is on folded lines, the cursor is always displayed in the
first column. The ruler does show the actual cursor position, but since the
line is folded, it cannot be displayed there.

Many movement commands handle a sequence of folded lines like an empty line.
For example, the "w" command stops once in the first column.

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

When using an operator, a closed fold is included as a whole. Thus "dl"
deletes the whole closed fold under the cursor.

For Ex commands that work on buffer lines the range is adjusted to always
start at the first line of a closed fold and end at the last line of a closed
fold. Thus this command: >
        :s/foo/bar/g
when used with the cursor on a closed fold, will replace "foo" with "bar" in
all lines of the fold.
This does not happen for |:folddoopen| and |:folddoclosed|.

When editing a buffer that has been edited before, the last used folding
settings are used again. For manual folding the defined folds are restored.
For all folding methods the manually opened and closed folds are restored.
If this buffer has been edited in this window, the values from back then are
used. Otherwise the values from the window where the buffer was edited last
are used.

==
 vim:tw=78:ts=8:ft=help:norl: