Basic editing ~
|starting.txt| starting Vim, Vim command arguments, initialisation
|editing.txt| editing and writing files
|motion.txt| commands for moving around
|scroll.txt| scrolling the text in the window
|insert.txt| Insert and Replace mode
|change.txt| deleting and replacing text
|indent.txt| automatic indenting for C and other languages
|undo.txt| Undo and Redo
|repeat.txt| repeating commands, Vim scripts and debugging
|visual.txt| using the Visual mode (selecting a text area)
|various.txt| various remaining commands
|recover.txt| recovering from a crash

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

                  VIM REFERENCE MANUAL   by Bram Moolenaar

Starting Vim   *starting*

1. Vim arguments |vim-arguments|
2. Vim on the Amiga |starting-amiga|
3. Running eVim |evim-keys|
4. Initialization |initialization|
5. $VIM and $VIMRUNTIME |$VIM|
6. Suspending |suspend|
7. Exiting |exiting|
8. Saving settings |save-settings|
9. Views and Sessions |views-sessions|
10. The viminfo file |viminfo-file|

==
1. Vim arguments   *vim-arguments*

Most often, Vim is started to edit a single file with the command

        vim filename   *-vim*

More generally, Vim is started with:

        vim [option | filename] ..

Option arguments and file name arguments can be mixed, and any number of them
can be given. However, watch out for options that take an argument.

For compatibility with various Vi versions, see |cmdline-arguments|.

Exactly one out of the following five items may be used to choose how to
start editing:

                                                        -file *---*
filename   One or more file names. The first one will be the current
                file and read into the buffer. The cursor will be positioned
                on the first line of the buffer.
                To avoid a file name starting with a '-' being interpreted as
                an option, precede the arglist with "--", e.g.: >
                        vim -- -filename
<   All arguments after the "--" will be interpreted as file names,
                no other options or "+command" argument can follow.

```
                For behavior of quotes on MS-Windows, see |win32-quotes|.

                                                        *--*
-               This argument can mean two things, depending on whether Ex
                mode is to be used.

                Starting in Normal mode: >
                        vim -
                        ex -v -
<               Start editing a new buffer, which is filled with text
                that is read from stdin.  The commands that would normally be
                read from stdin will now be read from stderr.  Example: >
                        find . -name "*.c" -print | vim -
<               The buffer will be marked modified, because it contains text
                that needs to be saved.  Except when in readonly mode, then
                the buffer is not marked modified.  Example: >
                        ls | view -
<
                Starting in Ex mode: >
                        ex -
                        vim -e -
                        exim -
                        vim -E
<               Start editing in silent mode.  See |-s-ex|.

                                                        *-t* *-tag*
-t {tag}        A tag.  "tag" is looked up in the tags file, the associated
                file becomes the current file, and the associated command is
                executed.  Mostly this is used for C programs, in which case
                "tag" often is a function name.  The effect is that the file
                containing that function becomes the current file and the
                cursor is positioned on the start of the function (see
                |tags|).

                                                        *-q* *-qf*
-q [errorfile]  QuickFix mode.  The file with the name [errorfile] is read
                and the first error is displayed.  See |quickfix|.
                If [errorfile] is not given, the 'errorfile' option is used
                for the file name.  See 'errorfile' for the default value.
                {not in Vi}

(nothing)       Without one of the four items above, Vim will start editing a
                new buffer.  It's empty and doesn't have a file name.


The startup mode can be changed by using another name instead of "vim", which
is equal to giving options:
ex      vim -e      Start in Ex mode (see |Ex-mode|).               *ex*
exim    vim -E      Start in improved Ex mode (see |Ex-mode|).      *exim*
                        (normally not installed)
view    vim -R      Start in read-only mode (see |-R|).             *view*
gvim    vim -g      Start the GUI (see |gui|).                      *gvim*
gex     vim -eg     Start the GUI in Ex mode.                     *gex*
gview   vim -Rg     Start the GUI in read-only mode.              *gview*
rvim    vim -Z      Like "vim", but in restricted mode (see |-Z|)   *rvim*
rview   vim -RZ     Like "view", but in restricted mode.          *rview*
rgvim   vim -gZ     Like "gvim", but in restricted mode.          *rgvim*
rgview  vim -RgZ    Like "gview", but in restricted mode.         *rgview*
evim    vim -y      Easy Vim: set 'insertmode' (see |-y|)           *evim*
eview   vim -yR     Like "evim" in read-only mode                 *eview*
vimdiff vim -d      Start in diff mode |diff-mode|
gvimdiff vim -gd    Start in diff mode |diff-mode|
```

```
Additional characters may follow, they are ignored.  For example, you can have
"gvim-5" to start the GUI.  You must have an executable by that name then, of
course.

On Unix, you would normally have one executable called Vim, and links from the
different startup-names to that executable.  If your system does not support
links and you do not want to have several copies of the executable, you could
use an alias instead.  For example: >
        alias view   vim -R
        alias gvim   vim -g
<
                                                        *startup-options*
The option arguments may be given in any order.  Single-letter options can be
combined after one dash.  There can be no option arguments after the "--"
argument.

On VMS all option arguments are assumed to be lowercase, unless preceded with
a slash.  Thus "-R" means recovery and "-/R" readonly.

--help                                                  *-h* *--help*
-h              Give usage (help) message and exit.  {not in Vi}
                See |info-message| about capturing the text.

                                                        *--version*
--version       Print version information and exit.  Same output as for
                |:version| command.  {not in Vi}
                See |info-message| about capturing the text.

                                                        *--noplugin*
--noplugin      Skip loading plugins.  Resets the 'loadplugins' option.
                {not in Vi}
                Note that the |-u| argument may also disable loading plugins:
                        argument   load: vimrc files  plugins  defaults.vim ~
                        (nothing)            yes        yes       yes
                        -u NONE              no         no        no
                        -u DEFAULTS          no         no        yes
                        -u NORC              no         yes       no
                        --noplugin           yes        no        yes

--startuptime {fname}                                   *--startuptime*
                During startup write timing messages to the file {fname}.
                This can be used to find out where time is spent while loading
                your .vimrc, plugins and opening the first file.
                When {fname} already exists new messages are appended.
                (Only available when compiled with the |+startuptime|
                feature).

                                                        *--literal*
--literal       Take file names literally, don't expand wildcards.  Not needed
                for Unix, because Vim always takes file names literally (the
                shell expands wildcards).
                Applies to all the names, also the ones that come before this
                argument.

                                                        *-+*
+[num]          The cursor will be positioned on line "num" for the first
                file being edited.  If "num" is missing, the cursor will be
                positioned on the last line.

                                                        *-+/*
+/{pat}         The cursor will be positioned on the first line containing
```

```
                "pat" in the first file being edited (see |pattern| for the
                available search patterns).  The search starts at the cursor
                position, which can be the first line or the cursor position
                last used from |viminfo|. To force a search from the first
                line use "+1 +/pat".

+{command}                                                  *-+c* *-c*
-c {command}    {command} will be executed after the first file has been
                read (and after autocommands and modelines for that file have
                been processed).  "command" is interpreted as an Ex command.
                If the "command" contains spaces, it must be enclosed in
                double quotes (this depends on the shell that is used).
                Example: >
                        vim  "+set si"  main.c
                        vim  "+find stdio.h"
                        vim  -c "set ff=dos"  -c wq  mine.mak
<
                Note: You can use up to 10 "+" or "-c" arguments in a Vim
                command.  They are executed in the order given.  A "-S"
                argument counts as a "-c" argument as well.
                {Vi only allows one command}

--cmd {command}                                             *--cmd*
                {command} will be executed before processing any vimrc file.
                Otherwise it acts like -c {command}.  You can use up to 10 of
                these commands, independently from "-c" commands.
                {not in Vi}

                                                            *-S*
-S {file}       The {file} will be sourced after the first file has been read.
                This is an easy way to do the equivalent of: >
                        -c "source {file}"
<               It can be mixed with "-c" arguments and repeated like "-c".
                The limit of 10 "-c" arguments applies here as well.
                {file} cannot start with a "-".
                {not in Vi}

-S              Works like "-S Session.vim".  Only when used as the last
                argument or when another "-" option follows.

                                                            *-r*
-r              Recovery mode.  Without a file name argument, a list of
                existing swap files is given.  With a file name, a swap file
                is read to recover a crashed editing session.  See
                |crash-recovery|.

                                                            *-L*
-L              Same as -r.  {only in some versions of Vi: "List recoverable
                edit sessions"}

                                                            *-R*
-R              Readonly mode.  The 'readonly' option will be set for all the
                files being edited.  You can still edit the buffer, but will
                be prevented from accidentally overwriting a file.  If you
                forgot that you are in View mode and did make some changes,
                you can overwrite a file by adding an exclamation mark to
                the Ex command, as in ":w!".  The 'readonly' option can be
                reset with ":set noro" (see the options chapter, |options|).
                Subsequent edits will not be done in readonly mode.  Calling
                the executable "view" has the same effect as the -R argument.
                The 'updatecount' option will be set to 10000, meaning that
                the swap file will not be updated automatically very often.
```

```
                See |-M| for disallowing modifications.

                                                        *-m*
-m              Modifications not allowed to be written.  The 'write' option
                will be reset, so that writing files is disabled.  However,
                the 'write' option can be set to enable writing again.
                {not in Vi}

                                                        *-M*
-M              Modifications not allowed.  The 'modifiable' option will be
                reset, so that changes are not allowed.  The 'write' option
                will be reset, so that writing files is disabled.  However,
                the 'modifiable' and 'write' options can be set to enable
                changes and writing.
                {not in Vi}

                                                *-Z* *restricted-mode* *E145*
-Z              Restricted mode.  All commands that make use of an external
                shell are disabled.  This includes suspending with CTRL-Z,
                ":sh", filtering, the system() function, backtick expansion,
                delete(), rename(), mkdir(), writefile(), libcall(),
                job_start(), etc.
                {not in Vi}

                                                        *-g*
-g              Start Vim in GUI mode.  See |gui|. For the opposite see |-v|.
                {not in Vi}

                                                        *-v*
-v              Start Ex in Vi mode.  Only makes a difference when the
                executable is called "ex" or "gvim".  For gvim the GUI is not
                started if possible.

                                                        *-e*
-e              Start Vim in Ex mode |Q|.  Only makes a difference when the
                executable is not called "ex".

                                                        *-E*
-E              Start Vim in improved Ex mode |gQ|.  Only makes a difference
                when the executable is not called "exim".
                {not in Vi}

                                                        *-s-ex*
-s              Silent or batch mode.  Only when Vim was started as "ex" or
                when preceded with the "-e" argument.  Otherwise see |-s|,
                which does take an argument while this use of "-s" doesn't.
                To be used when Vim is used to execute Ex commands from a file
                instead of a terminal.  Switches off most prompts and
                informative messages.  Also warnings and error messages.
                The output of these commands is displayed (to stdout):
                        :print
                        :list
                        :number
                        :set      to display option values.
                When 'verbose' is non-zero messages are printed (for
                debugging, to stderr).
                'term' and $TERM are not used.
                If Vim appears to be stuck try typing "qa!<Enter>".  You don't
                get a prompt thus you can't see Vim is waiting for you to type
                something.
                Initializations are skipped (except the ones given with the
                "-u" argument).
```

```
                Example: >
                        vim -e -s  < thefilter  thefile
<
                                                        *-b*
-b              Binary mode.  File I/O will only recognize <NL> to separate
                lines.  The 'expandtab' option will be reset.  The 'textwidth'
                option is set to 0.  'modeline' is reset.  The 'binary' option
                is set.  This is done after reading the vimrc/exrc files but
                before reading any file in the arglist.  See also
                |edit-binary|.  {not in Vi}

                                                        *-l*
-l              Lisp mode.  Sets the 'lisp' and 'showmatch' options on.

                                                        *-A*
-A              Arabic mode.  Sets the 'arabic' option on.  (Only when
                compiled with the |+arabic| features (which include
                |+rightleft|), otherwise Vim gives an error message
                and exits.)  {not in Vi}

                                                        *-F*
-F              Farsi mode.  Sets the 'fkmap' and 'rightleft' options on.
                (Only when compiled with |+rightleft| and |+farsi| features,
                otherwise Vim gives an error message and exits.)  {not in Vi}

                                                        *-H*
-H              Hebrew mode.  Sets the 'hkmap' and 'rightleft' options on.
                (Only when compiled with the |+rightleft| feature, otherwise
                Vim gives an error message and exits.)  {not in Vi}

                                                        *-V* *verbose*
-V[N]           Verbose.  Sets the 'verbose' option to [N] (default: 10).
                Messages will be given for each file that is ":source"d and
                for reading or writing a viminfo file.  Can be used to find
                out what is happening upon startup and exit.  {not in Vi}
                Example: >
                        vim -V8 foobar

-V[N]{filename}
                Like -V and set 'verbosefile' to {filename}.  The result is
                that messages are not displayed but written to the file
                {filename}.  {filename} must not start with a digit.
                Example: >
                        vim -V20vimlog foobar
<
                                                        *-D*
-D              Debugging.  Go to debugging mode when executing the first
                command from a script. |debug-mode|
                {not available when compiled without the |+eval| feature}
                {not in Vi}

                                                        *-C*
-C              Compatible mode.  Sets the 'compatible' option.  You can use
                this to get 'compatible', even though a .vimrc file exists.
                Keep in mind that the command ":set nocompatible" in some
                plugin or startup script overrules this, so you may end up
                with 'nocompatible' anyway.  To find out, use: >
                        :verbose set compatible?
<               Several plugins won't work with 'compatible' set.  You may
                want to set it after startup this way: >
                        vim "+set cp" filename
<               Also see |compatible-default|.  {not in Vi}
```

-N

-N Not compatible mode. Resets the 'compatible' option. You can use this to get 'nocompatible', when there is no .vimrc file or when using "-u NONE".
Also see |compatible-default|. {not in Vi}

-y *easy*

-y Easy mode. Implied for |evim| and |eview|. Starts with 'insertmode' set and behaves like a click-and-type editor. This sources the script $VIMRUNTIME/evim.vim. Mappings are set up to work like most click-and-type editors, see |evim-keys|. The GUI is started when available.
{not in Vi}

-n

-n No swap file will be used. Recovery after a crash will be impossible. Handy if you want to view or edit a file on a very slow medium (e.g., a floppy).
Can also be done with ":set updatecount=0". You can switch it on again by setting the 'updatecount' option to some value, e.g., ":set uc=100".
NOTE: Don't combine -n with -b, making -nb, because that has a different meaning: |-nb|.
'updatecount' is set to 0 AFTER executing commands from a vimrc file, but before the GUI initializations. Thus it overrides a setting for 'updatecount' in a vimrc file, but not in a gvimrc file. See |startup|.
When you want to reduce accesses to the disk (e.g., for a laptop), don't use "-n", but set 'updatetime' and 'updatecount' to very big numbers, and type ":preserve" when you want to save your work. This way you keep the possibility for crash recovery.
{not in Vi}

-o

-o[N] Open N windows, split horizontally. If [N] is not given, one window is opened for every file given as argument. If there is not enough room, only the first few files get a window. If there are more windows than arguments, the last few windows will be editing an empty file.
{not in Vi}

-O

-O[N] Open N windows, split vertically. Otherwise it's like -o. If both the -o and the -O option are given, the last one on the command line determines how the windows will be split.
{not in Vi}

-p

-p[N] Open N tab pages. If [N] is not given, one tab page is opened for every file given as argument. The maximum is set with 'tabpagemax' pages (default 10). If there are more tab pages than arguments, the last few tab pages will be editing an empty file. Also see |tabpage|.
{not in Vi}

-T

-T {terminal} Set the terminal type to "terminal". This influences the codes that Vim will send to your terminal. This is normally not needed, because Vim will be able to find out what type of terminal you are using. (See |terminal-info|.) {not in Vi}

--not-a-term
--not-a-term Tells Vim that the user knows that the input and/or output is
not connected to a terminal. This will avoid the warning and
the two second delay that would happen. {not in Vi}

--ttyfail
--ttyfail When the stdin or stdout is not a terminal (tty) then exit
right away.

-d
-d Start in diff mode, like |vimdiff|.
{not in Vi} {not available when compiled without the |+diff|
feature}

-d {device} Only on the Amiga and when not compiled with the |+diff|
feature. Works like "-dev".
-dev
-dev {device} Only on the Amiga: The {device} is opened to be used for
editing.
Normally you would use this to set the window position and
size: "-d con:x/y/width/height", e.g.,
"-d con:30/10/600/150". But you can also use it to start
editing on another device, e.g., AUX:. {not in Vi}
-f
-f GUI: Do not disconnect from the program that started Vim.
'f' stands for "foreground". If omitted, the GUI forks a new
process and exits the current one. "-f" should be used when
gvim is started by a program that will wait for the edit
session to finish (e.g., mail or readnews). If you want gvim
never to fork, include 'f' in 'guioptions' in your |gvimrc|.
Careful: You can use "-gf" to start the GUI in the foreground,
but "-fg" is used to specify the foreground color. |gui-fork|

Amiga: Do not restart Vim to open a new window. This
option should be used when Vim is started by a program that
will wait for the edit session to finish (e.g., mail or
readnews). See |amiga-window|.

MS-Windows: This option is not supported. However, when
running Vim with an installed vim.bat or gvim.bat file it
works.
{not in Vi}

--nofork
--nofork GUI: Do not fork. Same as |-f|.
-u *E282*
-u {vimrc} The file {vimrc} is read for initializations. Most other
initializations are skipped; see |initialization|.

This can be used to start Vim in a special mode, with special
mappings and settings. A shell alias can be used to make
this easy to use. For example: >
alias vimc vim -u ~/.c_vimrc !*
< Also consider using autocommands; see |autocommand|.

When {vimrc} is equal to "NONE" (all uppercase), all
initializations from files and environment variables are
skipped, including reading the |gvimrc| file when the GUI
starts. Loading plugins is also skipped.

```
                When {vimrc} is equal to "NORC" (all uppercase), this has the
                same effect as "NONE", but loading plugins is not skipped.

                When {vimrc} is equal to "DEFAULTS" (all uppercase), this has
                the same effect as "NONE", but the |defaults.vim| script is
                loaded, which will also set 'nocompatible'.

                Using the "-u" argument with another argument than DEFAULTS
                has the side effect that the 'compatible' option will be on by
                default.  This can have unexpected effects.  See
                |'compatible'|.
                {not in Vi}

                                                        *-U* *E230*
-U {gvimrc}     The file {gvimrc} is read for initializations when the GUI
                starts.  Other GUI initializations are skipped.  When {gvimrc}
                is equal to "NONE", no file is read for GUI initializations at
                all.  |gui-init|
                Exception: Reading the system-wide menu file is always done.
                {not in Vi}

                                                        *-i*
-i {viminfo}    The file "viminfo" is used instead of the default viminfo
                file.  If the name "NONE" is used (all uppercase), no viminfo
                file is read or written, even if 'viminfo' is set or when
                ":rv" or ":wv" are used.  See also |viminfo-file|.
                {not in Vi}

                                                        *--clean*
--clean         Equal to "-u DEFAULTS -i NONE":
                - initializations from files and environment variables is
                  skipped
                - the |defaults.vim| script is loaded, which implies
                  'nocompatible': use Vim defaults
                - no viminfo file is read or written
                                                        *-x*
-x              Use encryption to read/write files.  Will prompt for a key,
                which is then stored in the 'key' option.  All writes will
                then use this key to encrypt the text.  The '-x' argument is
                not needed when reading a file, because there is a check if
                the file that is being read has been encrypted, and Vim asks
                for a key automatically. |encryption|

                                                        *-X*
-X              Do not try connecting to the X server to get the current
                window title and copy/paste using the X clipboard.  This
                avoids a long startup time when running Vim in a terminal
                emulator and the connection to the X server is slow.
                See |--startuptime| to find out if affects you.
                Only makes a difference on Unix or VMS, when compiled with the
                |+X11| feature.  Otherwise it's ignored.
                To disable the connection only for specific terminals, see the
                'clipboard' option.
                When the X11 Session Management Protocol (XSMP) handler has
                been built in, the -X option also disables that connection as
                it, too, may have undesirable delays.
                When the connection is desired later anyway (e.g., for
                client-server messages), call the |serverlist()| function.
                This does not enable the XSMP handler though.
                {not in Vi}

                                                        *-s*
```

```
-s {scriptin}   The script file "scriptin" is read.  The characters in the
                file are interpreted as if you had typed them.  The same can
                be done with the command ":source! {scriptin}".  If the end
                of the file is reached before the editor exits, further
                characters are read from the keyboard.  Only works when not
                started in Ex mode, see |-s-ex|.  See also |complex-repeat|.
                {not in Vi}

                                                        *-w_nr*
-w {number}
-w{number}      Set the 'window' option to {number}.

                                                        *-w*
-w {scriptout}  All the characters that you type are recorded in the file
                "scriptout", until you exit Vim.  This is useful if you want
                to create a script file to be used with "vim -s" or
                ":source!".  When the "scriptout" file already exists, new
                characters are appended.  See also |complex-repeat|.
                {scriptout} cannot start with a digit.
                {not in Vi}

                                                        *-W*
-W {scriptout}  Like -w, but do not append, overwrite an existing file.
                {not in Vi}

--remote [+{cmd}] {file} ...
                Open the {file} in another Vim that functions as a server.
                Any non-file arguments must come before this.
                See |--remote|. {not in Vi}

--remote-silent [+{cmd}] {file} ...
                Like --remote, but don't complain if there is no server.
                See |--remote-silent|. {not in Vi}

--remote-wait [+{cmd}] {file} ...
                Like --remote, but wait for the server to finish editing the
                file(s).
                See |--remote-wait|. {not in Vi}

--remote-wait-silent [+{cmd}] {file} ...
                Like --remote-wait, but don't complain if there is no server.
                See |--remote-wait-silent|. {not in Vi}

--servername {name}
                Specify the name of the Vim server to send to or to become.
                See |--servername|. {not in Vi}

--remote-send {keys}
                Send {keys} to a Vim server and exit.
                See |--remote-send|. {not in Vi}

--remote-expr {expr}
                Evaluate {expr} in another Vim that functions as a server.
                The result is printed on stdout.
                See |--remote-expr|. {not in Vi}

--serverlist    Output a list of Vim server names and exit.  See
                |--serverlist|. {not in Vi}

--socketid {id}                                         *--socketid*
                GTK+ GUI Vim only.  Make gvim try to use GtkPlug mechanism, so
                that it runs inside another window.  See |gui-gtk-socketid|
```

```
                for details. {not in Vi}

--windowid {id}                                         *--windowid*
                Win32 GUI Vim only.  Make gvim try to use the window {id} as a
                parent, so that it runs inside that window.  See
                |gui-w32-windowid| for details. {not in Vi}

--echo-wid                                              *--echo-wid*
                GTK+ GUI Vim only.  Make gvim echo the Window ID on stdout,
                which can be used to run gvim in a kpart widget.  The format
                of the output is: >
                        WID: 12345\n
<               {not in Vi}

--role {role}                                           *--role*
                GTK+ 2 GUI only.  Set the role of the main window to {role}.
                The window role can be used by a window manager to uniquely
                identify a window, in order to restore window placement and
                such.  The --role argument is passed automatically when
                restoring the session on login.  See |gui-gnome-session|
                {not in Vi}

-P {parent-title}                                 *-P* *MDI* *E671* *E672*
                Win32 only: Specify the title of the parent application.  When
                possible, Vim will run in an MDI window inside the
                application.
                {parent-title} must appear in the window title of the parent
                application.  Make sure that it is specific enough.
                Note that the implementation is still primitive.  It won't
                work with all applications and the menu doesn't work.

-nb                                                     *-nb*
-nb={fname}
-nb:{hostname}:{addr}:{password}
                Attempt connecting to Netbeans and become an editor server for
                it.  The second form specifies a file to read connection info
                from.  The third form specifies the hostname, address and
                password for connecting to Netbeans. |netbeans-run|
                {only available when compiled with the |+netbeans_intg|
                feature; if not then -nb will make Vim exit}

If the executable is called "view", Vim will start in Readonly mode.  This is
useful if you can make a hard or symbolic link from "view" to "vim".
Starting in Readonly mode can also be done with "vim -R".

If the executable is called "ex", Vim will start in "Ex" mode.  This means it
will accept only ":" commands.  But when the "-v" argument is given, Vim will
start in Normal mode anyway.

Additional arguments are available on unix like systems when compiled with
X11 GUI support.  See |gui-resources|.

==============================================================================
2. Vim on the Amiga                                     *starting-amiga*

Starting Vim from the Workbench                         *workbench*
-------------------------------

Vim can be started from the Workbench by clicking on its icon twice.  It will
then start with an empty buffer.

Vim can be started to edit one or more files by using a "Project" icon.  The
```

"Default Tool" of the icon must be the full pathname of the Vim executable.
The name of the ".info" file must be the same as the name of the text file.
By clicking on this icon twice, Vim will be started with the file name as
current file name, which will be read into the buffer (if it exists). You can
edit multiple files by pressing the shift key while clicking on icons, and
clicking twice on the last one. The "Default Tool" for all these icons must
be the same.

It is not possible to give arguments to Vim, other than file names, from the
workbench.

Vim window *amiga-window*

Vim will run in the CLI window where it was started. If Vim was started with
the "run" or "runback" command, or if Vim was started from the workbench, it
will open a window of its own.

Technical detail:
        To open the new window a little trick is used. As soon as Vim
        recognizes that it does not run in a normal CLI window, it will
        create a script file in "t:". This script file contains the same
        command as the one Vim was started with, and an "endcli" command.
        This script file is then executed with a "newcli" command (the "c:run"
        and "c:newcli" commands are required for this to work). The script
        file will hang around until reboot, or until you delete it. This
        method is required to get the ":sh" and ":!" commands to work
        correctly. But when Vim was started with the -f option (foreground
        mode), this method is not used. The reason for this is that
        when a program starts Vim with the -f option it will wait for Vim to
        exit. With the script trick, the calling program does not know when
        Vim exits. The -f option can be used when Vim is started by a mail
        program which also waits for the edit session to finish. As a
        consequence, the ":sh" and ":!" commands are not available when the
        -f option is used.

Vim will automatically recognize the window size and react to window
resizing. Under Amiga DOS 1.3, it is advised to use the fastfonts program,
"FF", to speed up display redrawing.

==
3. Running eVim *evim-keys*

EVim runs Vim as click-and-type editor. This is very unlike the original Vi
idea. But it helps for people that don't use Vim often enough to learn the
commands. Hopefully they will find out that learning to use Normal mode
commands will make their editing much more effective.

In Evim these options are changed from their default value:

        :set nocompatible   Use Vim improvements
        :set insertmode   Remain in Insert mode most of the time
        :set hidden   Keep invisible buffers loaded
        :set backup   Keep backup files (not for VMS)
        :set backspace=2   Backspace over everything
        :set autoindent   auto-indent new lines
        :set history=50   keep 50 lines of Ex commands
        :set ruler   show the cursor position
        :set incsearch   show matches halfway typing a pattern
        :set mouse=a   use the mouse in all modes
        :set hlsearch   highlight all matches for a search pattern
        :set whichwrap+=<,>,[,] <Left> and <Right> wrap around line breaks

```
        :set guioptions-=a      non-Unix only: don't do auto-select

Key mappings:
        <Down>          moves by screen lines rather than file lines
        <Up>            idem
        Q               does "gq", formatting, instead of Ex mode
        <BS>            in Visual mode: deletes the selection
        CTRL-X          in Visual mode: Cut to clipboard
        <S-Del>         idem
        CTRL-C          in Visual mode: Copy to clipboard
        <C-Insert>      idem
        CTRL-V          Pastes from the clipboard (in any mode)
        <S-Insert>      idem
        CTRL-Q          do what CTRL-V used to do
        CTRL-Z          undo
        CTRL-Y          redo
        <M-Space>       system menu
        CTRL-A          select all
        <C-Tab>         next window, CTRL-W w
        <C-F4>          close window, CTRL-W c

Additionally:
- ":behave mswin" is used |:behave|
- syntax highlighting is enabled
- filetype detection is enabled, filetype plugins and indenting is enabled
- in a text file 'textwidth' is set to 78

One hint: If you want to go to Normal mode to be able to type a sequence of
commands, use CTRL-L. |i_CTRL-L|

==============================================================================
4. Initialization                                   *initialization* *startup*

This section is about the non-GUI version of Vim.  See |gui-fork| for
additional initialization when starting the GUI.

At startup, Vim checks environment variables and files and sets values
accordingly.  Vim proceeds in this order:

1. Set the 'shell' and 'term' option            *SHELL* *COMSPEC* *TERM*
        The environment variable SHELL, if it exists, is used to set the
        'shell' option.  On MS-DOS and Win32, the COMSPEC variable is used
        if SHELL is not set.
        The environment variable TERM, if it exists, is used to set the 'term'
        option.  However, 'term' will change later when starting the GUI (step
        8 below).

2. Process the arguments
        The options and file names from the command that start Vim are
        inspected.  Buffers are created for all files (but not loaded yet).
        The |-V| argument can be used to display or log what happens next,
        useful for debugging the initializations.

3. Execute Ex commands, from environment variables and/or files
        An environment variable is read as one Ex command line, where multiple
        commands must be separated with '|' or "<NL>".
                                                                *vimrc* *exrc*
        A file that contains initialization commands is called a "vimrc" file.
        Each line in a vimrc file is executed as an Ex command line.  It is
        sometimes also referred to as "exrc" file.  They are the same type of
        file, but "exrc" is what Vi always used, "vimrc" is a Vim specific
        name.  Also see |vimrc-intro|.
```

```
   Places for your personal initializations:
           Unix            $HOME/.vimrc or $HOME/.vim/vimrc
           OS/2            $HOME/.vimrc, $HOME/vimfiles/vimrc
                           or $VIM/.vimrc (or _vimrc)
           MS-Windows      $HOME/_vimrc, $HOME/vimfiles/vimrc
                           or $VIM/_vimrc
           Amiga           s:.vimrc, home:.vimrc, home:vimfiles:vimrc
                           or $VIM/.vimrc

   The files are searched in the order specified above and only the first
   one that is found is read.

   RECOMMENDATION: Put all your Vim configuration stuff in the
   $HOME/.vim/ directory ($HOME/vimfiles/ for MS-Windows). That makes it
   easy to copy it to another system.

   If Vim was started with "-u filename", the file "filename" is used.
   All following initializations until 4. are skipped. $MYVIMRC is not
   set.
   "vim -u NORC" can be used to skip these initializations without
   reading a file.  "vim -u NONE" also skips loading plugins.  |-u|

   If Vim was started in Ex mode with the "-s" argument, all following
   initializations until 4. are skipped.  Only the "-u" option is
   interpreted.
                                                          *evim.vim*
a. If vim was started as |evim| or |eview| or with the |-y| argument, the
   script $VIMRUNTIME/evim.vim will be loaded.
                                                          *system-vimrc*
b. For Unix, MS-DOS, MS-Windows, OS/2, VMS, Macintosh, RISC-OS and Amiga
   the system vimrc file is read for initializations.  The path of this
   file is shown with the ":version" command.  Mostly it's "$VIM/vimrc".
   Note that this file is ALWAYS read in 'compatible' mode, since the
   automatic resetting of 'compatible' is only done later.  Add a ":set
   nocp" command if you like.
   For the Macintosh the $VIMRUNTIME/macmap.vim is read.

     *VIMINIT* *.vimrc* *_vimrc* *EXINIT* *.exrc* *_exrc* *$MYVIMRC*
c. Five places are searched for initializations.  The first that exists
   is used, the others are ignored.  The $MYVIMRC environment variable is
   set to the file that was first found, unless $MYVIMRC was already set
   and when using VIMINIT.
   I   The environment variable VIMINIT (see also |compatible-default|) (*)
       The value of $VIMINIT is used as an Ex command line.
   II  The user vimrc file(s):
               "$HOME/.vimrc"         (for Unix and OS/2) (*)
               "$HOME/.vim/vimrc"     (for Unix and OS/2) (*)
               "s:.vimrc"             (for Amiga) (*)
               "home:.vimrc"          (for Amiga) (*)
               "home:vimfiles:vimrc"  (for Amiga) (*)
               "$VIM/.vimrc"          (for OS/2 and Amiga) (*)
               "$HOME/_vimrc"         (for MS-DOS and Win32) (*)
               "$HOME/vimfiles/vimrc" (for MS-DOS and Win32) (*)
               "$VIM/_vimrc"          (for MS-DOS and Win32) (*)
           Note: For Unix, OS/2 and Amiga, when ".vimrc" does not exist,
           "_vimrc" is also tried, in case an MS-DOS compatible file
           system is used.  For MS-DOS and Win32 ".vimrc" is checked
           after "_vimrc", in case long file names are used.
           Note: For MS-DOS and Win32, "$HOME" is checked first.  If no
           "_vimrc" or ".vimrc" is found there, "$VIM" is tried.
           See |$VIM| for when $VIM is not set.
```

```
        III The environment variable EXINIT.
            The value of $EXINIT is used as an Ex command line.
        IV  The user exrc file(s).  Same as for the user vimrc file, but with
            "vimrc" replaced by "exrc".  But only one of ".exrc" and "_exrc" is
            used, depending on the system.  And without the (*)!
        V   The default vimrc file, $VIMRUNTIME/defaults.vim.  This sets up
            options values and has "syntax on" and "filetype on" commands,
            which is what most new users will want.  See |defaults.vim|.

     d. If the 'exrc' option is on (which is NOT the default), the current
        directory is searched for three files.  The first that exists is used,
        the others are ignored.
        -  The file ".vimrc" (for Unix, Amiga and OS/2) (*)
                   "_vimrc" (for MS-DOS and Win32) (*)
        -  The file "_vimrc" (for Unix, Amiga and OS/2) (*)
                   ".vimrc" (for MS-DOS and Win32) (*)
        -  The file ".exrc"  (for Unix, Amiga and OS/2)
                   "_exrc"  (for MS-DOS and Win32)

     (*) Using this file or environment variable will cause 'compatible' to be
         off by default.  See |compatible-default|.

4. Load the plugin scripts.                                    *load-plugins*
        This does the same as the command: >
                :runtime! plugin/**/*.vim
<       The result is that all directories in the 'runtimepath' option will be
        searched for the "plugin" sub-directory and all files ending in ".vim"
        will be sourced (in alphabetical order per directory), also in
        subdirectories.
        However, directories in 'runtimepath' ending in "after" are skipped
        here and only loaded after packages, see below.
        Loading plugins won't be done when:
        - The 'loadplugins' option was reset in a vimrc file.
        - The |--noplugin| command line argument is used.
        - The |--clean| command line argument is used.
        - The "-u NONE" command line argument is used |-u|.
        - When Vim was compiled without the |+eval| feature.
        Note that using "-c 'set noloadplugins'" doesn't work, because the
        commands from the command line have not been executed yet.  You can
        use "--cmd 'set noloadplugins'" or "--cmd 'set loadplugins'" |--cmd|.

        Packages are loaded.  These are plugins, as above, but found in the
        "start" directory of each entry in 'packpath'.  Every plugin directory
        found is added in 'runtimepath' and then the plugins are sourced.  See
        |packages|.

        The plugins scripts are loaded, as above, but now only the directories
        ending in "after" are used.  Note that 'runtimepath' will have changed
        if packages have been found, but that should not add a directory
        ending in "after".

5. Set 'shellpipe' and 'shellredir'
        The 'shellpipe' and 'shellredir' options are set according to the
        value of the 'shell' option, unless they have been set before.
        This means that Vim will figure out the values of 'shellpipe' and
        'shellredir' for you, unless you have set them yourself.

6. Set 'updatecount' to zero, if "-n" command argument used

7. Set binary options
        If the "-b" flag was given to Vim, the options for binary editing will
        be set now.  See |-b|.
```

8. Perform GUI initializations
        Only when starting "gvim", the GUI initializations will be done. See
        |gui-init|.

9. Read the viminfo file
        If the 'viminfo' option is not empty, the viminfo file is read. See
        |viminfo-file|.

10. Read the quickfix file
        If the "-q" flag was given to Vim, the quickfix file is read. If this
        fails, Vim exits.

11. Open all windows
        When the |-o| flag was given, windows will be opened (but not
        displayed yet).
        When the |-p| flag was given, tab pages will be created (but not
        displayed yet).
        When switching screens, it happens now. Redrawing starts.
        If the "-q" flag was given to Vim, the first error is jumped to.
        Buffers for all windows will be loaded.

12. Execute startup commands
        If a "-t" flag was given to Vim, the tag is jumped to.
        The commands given with the |-c| and |+cmd| arguments are executed.
        If the 'insertmode' option is set, Insert mode is entered.
        The starting flag is reset, has("vim_starting") will now return zero.
        The |v:vim_did_enter| variable is set to 1.
        The |VimEnter| autocommands are executed.

The $MYVIMRC or $MYGVIMRC file will be set to the first found vimrc and/or
gvimrc file.

Some hints on using initializations ~

Standard setup:
Create a vimrc file to set the default settings and mappings for all your edit
sessions. Put it in a place so that it will be found by 3b:
        ~/.vimrc   (Unix and OS/2)
        s:.vimrc   (Amiga)
        $VIM_vimrc   (MS-DOS and Win32)
Note that creating a vimrc file will cause the 'compatible' option to be off
by default. See |compatible-default|.

Local setup:
Put all commands that you need for editing a specific directory only into a
vimrc file and place it in that directory under the name ".vimrc" ("_vimrc"
for MS-DOS and Win32). NOTE: To make Vim look for these special files you
have to turn on the option 'exrc'. See |trojan-horse| too.

System setup:
This only applies if you are managing a Unix system with several users and
want to set the defaults for all users. Create a vimrc file with commands
for default settings and mappings and put it in the place that is given with
the ":version" command.

Saving the current state of Vim to a file ~

Whenever you have changed values of options or when you have created a
mapping, then you may want to save them in a vimrc file for later use. See

|save-settings| about saving the current state of settings to a file.

Avoiding setup problems for Vi users ~

Vi uses the variable EXINIT and the file "~/.exrc". So if you do not want to interfere with Vi, then use the variable VIMINIT and the file "vimrc" instead.

Amiga environment variables ~

On the Amiga, two types of environment variables exist. The ones set with the DOS 1.3 (or later) setenv command are recognized. See the AmigaDos 1.3 manual. The environment variables set with the old Manx Set command (before version 5.0) are not recognized.

MS-DOS line separators ~

On MS-DOS-like systems (MS-DOS itself, Win32, and OS/2), Vim assumes that all the vimrc files have <CR> <NL> pairs as line separators. This will give problems if you have a file with only <NL>s and have a line like ":map xx yy^M". The trailing ^M will be ignored.

Vi compatible default value ~

compatible-default

When Vim starts, the 'compatible' option is on. This will be used when Vim starts its initializations. But as soon as:
- a user vimrc file is found, or
- a vimrc file in the current directory, or
- the "VIMINIT" environment variable is set, or
- the "-N" command line argument is given, or
- the "--clean" command line argument is given, or
  even when no vimrc file exists.
- the |defaults.vim| script is loaded, or
- gvimrc file was found,

then it will be set to 'nocompatible'.

Note that this does NOT happen when a system-wide vimrc file was found.

This has the side effect of setting or resetting other options (see 'compatible'). But only the options that have not been set or reset will be changed. This has the same effect like the value of 'compatible' had this value when starting Vim.

'compatible' is NOT reset, and |defaults.vim| is not loaded:
- when Vim was started with the |-u| command line argument, especially with "-u NONE", or
- when started with the |-C| command line argument, or
- when the name of the executable ends in "ex". (This has been done to make Vim behave like "ex", when it is started as "ex")

But there is a side effect of setting or resetting 'compatible' at the moment a .vimrc file is found: Mappings are interpreted the moment they are encountered. This makes a difference when using things like "<CR>". If the mappings depend on a certain value of 'compatible', set or reset it before giving the mapping.

Defaults without a .vimrc file ~

defaults.vim

If Vim is started normally and no user vimrc file is found, the
$VIMRUTIME/defaults.vim script is loaded. This will set 'compatible' off,
switch on syntax highlighting and a few more things. See the script for
details. NOTE: this is done since Vim 8.0, not in Vim 7.4. (it was added in
patch 7.4.2111 to be exact).

This should work well for new Vim users. If you create your own .vimrc, it is
recommended to add this line somewhere near the top: >
	unlet! skip_defaults_vim
	source $VIMRUNTIME/defaults.vim
Then Vim works like before you had a .vimrc. Copying $VIMRUNTIME/vimrc_example
is way to do this. Alternatively, you can copy defaults.vim to your .vimrc
and modify it (but then you won't get updates when it changes).

If you don't like some of the defaults, you can still source defaults.vim and
revert individual settings. See the defaults.vim file for hints on how to
revert each item.
skip_defaults_vim
If you use a system-wide vimrc and don't want defaults.vim to change settings,
set the "skip_defaults_vim" variable. If this was set and you want to load
defaults.vim from your .vimrc, first unlet skip_defaults_vim, as in the
example above.

Avoiding trojan horses ~
trojan-horse
While reading the "vimrc" or the "exrc" file in the current directory, some
commands can be disabled for security reasons by setting the 'secure' option.
This is always done when executing the command from a tags file. Otherwise it
would be possible that you accidentally use a vimrc or tags file that somebody
else created and contains nasty commands. The disabled commands are the ones
that start a shell, the ones that write to a file, and ":autocmd". The ":map"
commands are echoed, so you can see which keys are being mapped.
	If you want Vim to execute all commands in a local vimrc file, you
can reset the 'secure' option in the EXINIT or VIMINIT environment variable or
in the global "exrc" or "vimrc" file. This is not possible in "vimrc" or
"exrc" in the current directory, for obvious reasons.
	On Unix systems, this only happens if you are not the owner of the
vimrc file. Warning: If you unpack an archive that contains a vimrc or exrc
file, it will be owned by you. You won't have the security protection. Check
the vimrc file before you start Vim in that directory, or reset the 'exrc'
option. Some Unix systems allow a user to do "chown" on a file. This makes
it possible for another user to create a nasty vimrc and make you the owner.
Be careful!
	When using tag search commands, executing the search command (the last
part of the line in the tags file) is always done in secure mode. This works
just like executing a command from a vimrc/exrc in the current directory.

If Vim startup is slow ~
slow-start
If Vim takes a long time to start up, use the |--startuptime| argument to find
out what happens. There are a few common causes:
- If the Unix version was compiled with the GUI and/or X11 (check the output
  of ":version" for "+GUI" and "+X11"), it may need to load shared libraries
  and connect to the X11 server. Try compiling a version with GUI and X11
  disabled. This also should make the executable smaller.
  Use the |-X| command line argument to avoid connecting to the X server when
  running in a terminal.
- If you have "viminfo" enabled, the loading of the viminfo file may take a
  while. You can find out if this is the problem by disabling viminfo for a
  moment (use the Vim argument "-i NONE", |-i|). Try reducing the number of

lines stored in a register with ":set viminfo='20,<50,s10". |viminfo-file|.

Intro message ~

:intro

When Vim starts without a file name, an introductory message is displayed (for those who don't know what Vim is). It is removed as soon as the display is redrawn in any way. To see the message again, use the ":intro" command (if there is not enough room, you will see only part of it).
To avoid the intro message on startup, add the 'I' flag to 'shortmess'.

info-message

The |--help| and |--version| arguments cause Vim to print a message and then exit. Normally the message is sent to stdout, thus can be redirected to a file with: >

```
vim --help >file
```

From inside Vim: >

```
:read !vim --help
```

When using gvim, it detects that it might have been started from the desktop, without a terminal to show messages on. This is detected when both stdout and stderr are not a tty. This breaks the ":read" command, as used in the example above. To make it work again, set 'shellredir' to ">" instead of the default ">&": >

```
:set shellredir=>
:read !gvim --help
```

This still won't work for systems where gvim does not use stdout at all though.

==========

5. $VIM and $VIMRUNTIME

$VIM

The environment variable "$VIM" is used to locate various user files for Vim, such as the user startup script ".vimrc". This depends on the system, see |startup|.

To avoid the need for every user to set the $VIM environment variable, Vim will try to get the value for $VIM in this order:
1. The value defined by the $VIM environment variable. You can use this to make Vim look in a specific directory for its support files. Example: >
   setenv VIM /home/paul/vim
2. The path from 'helpfile' is used, unless it contains some environment variable too (the default is "$VIMRUNTIME/doc/help.txt": chicken-egg problem). The file name ("help.txt" or any other) is removed. Then trailing directory names are removed, in this order: "doc", "runtime" and "vim{version}" (e.g., "vim54").
3. For MSDOS, Win32 and OS/2 Vim tries to use the directory name of the executable. If it ends in "/src", this is removed. This is useful if you unpacked the .zip file in some directory, and adjusted the search path to find the vim executable. Trailing directory names are removed, in this order: "runtime" and "vim{version}" (e.g., "vim54").
4. For Unix the compile-time defined installation directory is used (see the output of ":version").

Once Vim has done this once, it will set the $VIM environment variable. To change it later, use a ":let" command like this: >

```
:let $VIM = "/home/paul/vim/"
```

<

$VIMRUNTIME
The environment variable "$VIMRUNTIME" is used to locate various support
files, such as the on-line documentation and files used for syntax
highlighting. For example, the main help file is normally
"$VIMRUNTIME/doc/help.txt".
You don't normally set $VIMRUNTIME yourself, but let Vim figure it out. This
is the order used to find the value of $VIMRUNTIME:
1. If the environment variable $VIMRUNTIME is set, it is used. You can use
   this when the runtime files are in an unusual location.
2. If "$VIM/vim{version}" exists, it is used. {version} is the version
   number of Vim, without any '-' or '.'. For example: "$VIM/vim54". This is
   the normal value for $VIMRUNTIME.
3. If "$VIM/runtime" exists, it is used.
4. The value of $VIM is used. This is for backwards compatibility with older
   versions.
5. When the 'helpfile' option is set and doesn't contain a '$', its value is
   used, with "doc/help.txt" removed from the end.

For Unix, when there is a compiled-in default for $VIMRUNTIME (check the
output of ":version"), steps 2, 3 and 4 are skipped, and the compiled-in
default is used after step 5. This means that the compiled-in default
overrules the value of $VIM. This is useful if $VIM is "/etc" and the runtime
files are in "/usr/share/vim/vim54".

Once Vim has done this once, it will set the $VIMRUNTIME environment variable.
To change it later, use a ":let" command like this: >
        :let $VIMRUNTIME = "/home/piet/vim/vim54"

In case you need the value of $VIMRUNTIME in a shell (e.g., for a script that
greps in the help files) you might be able to use this: >

        VIMRUNTIME=`vim -e -T dumb --cmd 'exe "set t_cm=\<C-M>"|echo $VIMRUNTIME|
quit' | tr -d '\015' `

==
6. Suspending   *suspend*

                                        iconize *iconise* *CTRL-Z* *v_CTRL-Z*
CTRL-Z   Suspend Vim, like ":stop".
                        Works in Normal and in Visual mode. In Insert and
                        Command-line mode, the CTRL-Z is inserted as a normal
                        character. In Visual mode Vim goes back to Normal
                        mode.
                        Note: if CTRL-Z undoes a change see |mswin.vim|.

:sus[pend][!]   or   *:sus* *:suspend* *:st* *:stop*
:st[op][!]   Suspend Vim.
                        If the '!' is not given and 'autowrite' is set, every
                        buffer with changes and a file name is written out.
                        If the '!' is given or 'autowrite' is not set, changed
                        buffers are not written, don't forget to bring Vim
                        back to the foreground later!

In the GUI, suspending is implemented as iconising gvim. In Windows 95/NT,
gvim is minimized.

On many Unix systems, it is possible to suspend Vim with CTRL-Z. This is only
possible in Normal and Visual mode (see next chapter, |vim-modes|). Vim will
continue if you make it the foreground job again. On other systems, CTRL-Z
will start a new shell. This is the same as the ":sh" command. Vim will

continue if you exit from the shell.

In X-windows the selection is disowned when Vim suspends. this means you can't paste it in another application (since Vim is going to sleep an attempt to get the selection would make the program hang).

==
7. Exiting *exiting*

There are several ways to exit Vim:
- Close the last window with `:quit`. Only when there are no changes.
- Close the last window with `:quit!`. Also when there are changes.
- Close all windows with `:qall`. Only when there are no changes.
- Close all windows with `:qall!`. Also when there are changes.
- Use `:cquit`. Also when there are changes.

When using `:cquit` or when there was an error message Vim exits with exit code 1. Errors can be avoided by using `:silent!` or with `:catch`.

==
8. Saving settings *save-settings*

Mostly you will edit your vimrc files manually. This gives you the greatest flexibility. There are a few commands to generate a vimrc file automatically. You can use these files as they are, or copy/paste lines to include in another vimrc file.

:mk *:mkexrc*
:mk[exrc] [file] Write current key mappings and changed options to [file] (default ".exrc" in the current directory), unless it already exists. {not in Vi}

:mk[exrc]! [file] Always write current key mappings and changed options to [file] (default ".exrc" in the current directory). {not in Vi}

:mkv *:mkvimrc*
:mkv[imrc][!] [file] Like ":mkexrc", but the default is ".vimrc" in the current directory. The ":version" command is also written to the file. {not in Vi}

These commands will write ":map" and ":set" commands to a file, in such a way that when these commands are executed, the current key mappings and options will be set to the same values. The options 'columns', 'endofline', 'fileformat', 'key', 'lines', 'modified', 'scroll', 'term', 'textmode', 'ttyfast' and 'ttymouse' are not included, because these are terminal or file dependent. Note that the options 'binary', 'paste' and 'readonly' are included, this might not always be what you want.

When special keys are used in mappings, The 'cpoptions' option will be temporarily set to its Vim default, to avoid the mappings to be misinterpreted. This makes the file incompatible with Vi, but makes sure it can be used with different terminals.

Only global mappings are stored, not mappings local to a buffer.

A common method is to use a default ".vimrc" file, make some modifications with ":map" and ":set" commands and write the modified file. First read the default ".vimrc" in with a command like ":source ~piet/.vimrc.Cprogs", change the settings and then save them in the current directory with ":mkvimrc!". If you want to make this file your default .vimrc, move it to your home directory (on Unix), s: (Amiga) or $VIM directory (MS-DOS). You could also use

```
autocommands |autocommand| and/or modelines |modeline|.

                                                    *vimrc-option-example*
If you only want to add a single option setting to your vimrc, you can use
these steps:
1. Edit your vimrc file with Vim.
2. Play with the option until it's right.  E.g., try out different values for
   'guifont'.
3. Append a line to set the value of the option, using the expression register
   '=' to enter the value.  E.g., for the 'guifont' option: >
   o:set guifont=<C-R>=&guifont<CR><Esc>
<  [<C-R> is a CTRL-R, <CR> is a return, <Esc> is the escape key]
   You need to escape special characters, esp. spaces.

Note that when you create a .vimrc file, this can influence the 'compatible'
option, which has several side effects.  See |'compatible'|.
":mkvimrc", ":mkexrc" and ":mksession" write the command to set or reset the
'compatible' option to the output file first, because of these side effects.

==============================================================================
9. Views and Sessions                                   *views-sessions*

This is introduced in sections |21.4| and |21.5| of the user manual.

                                                    *View* *view-file*
A View is a collection of settings that apply to one window.  You can save a
View and when you restore it later, the text is displayed in the same way.
The options and mappings in this window will also be restored, so that you can
continue editing like when the View was saved.

                                                    *Session* *session-file*
A Session keeps the Views for all windows, plus the global settings.  You can
save a Session and when you restore it later the window layout looks the same.
You can use a Session to quickly switch between different projects,
automatically loading the files you were last working on in that project.

Views and Sessions are a nice addition to viminfo-files, which are used to
remember information for all Views and Sessions together |viminfo-file|.

You can quickly start editing with a previously saved View or Session with the
|-S| argument: >
        vim -S Session.vim
<
All this is {not in Vi} and {not available when compiled without the
|+mksession| feature}.

                                                        *:mks* *:mksession*
:mks[ession][!] [file]  Write a Vim script that restores the current editing
                        session.
                        When [!] is included an existing file is overwritten.
                        When [file] is omitted "Session.vim" is used.

The output of ":mksession" is like ":mkvimrc", but additional commands are
added to the file.  Which ones depends on the 'sessionoptions' option.  The
resulting file, when executed with a ":source" command:
1. Restores global mappings and options, if 'sessionoptions' contains
   "options".  Script-local mappings will not be written.
2. Restores global variables that start with an uppercase letter and contain
   at least one lowercase letter, if 'sessionoptions' contains "globals".
3. Unloads all currently loaded buffers.
4. Restores the current directory if 'sessionoptions' contains "curdir", or
   sets the current directory to where the Session file is if 'sessionoptions'
```

```
   contains "sesdir".
5. Restores GUI Vim window position, if 'sessionoptions' contains "winpos".
6. Restores screen size, if 'sessionoptions' contains "resize".
7. Reloads the buffer list, with the last cursor positions.  If
   'sessionoptions' contains "buffers" then all buffers are restored,
   including hidden and unloaded buffers.  Otherwise only buffers in windows
   are restored.
8. Restores all windows with the same layout.  If 'sessionoptions' contains
   "help", help windows are restored.  If 'sessionoptions' contains "blank",
   windows editing a buffer without a name will be restored.
   If 'sessionoptions' contains "winsize" and no (help/blank) windows were
   left out, the window sizes are restored (relative to the screen size).
   Otherwise, the windows are just given sensible sizes.
9. Restores the Views for all the windows, as with |:mkview|.  But
   'sessionoptions' is used instead of 'viewoptions'.
10. If a file exists with the same name as the Session file, but ending in
   "x.vim" (for eXtra), executes that as well.  You can use *x.vim files to
   specify additional settings and actions associated with a given Session,
   such as creating menu items in the GUI version.

After restoring the Session, the full filename of your current Session is
available in the internal variable "v:this_session" |this_session-variable|.
An example mapping: >
  :nmap <F2> :wa<Bar>exe "mksession! " . v:this_session<CR>:so ~/sessions/
This saves the current Session, and starts off the command to load another.

A session includes all tab pages, unless "tabpages" was removed from
'sessionoptions'. |tab-page|

The |SessionLoadPost| autocmd event is triggered after a session file is
loaded/sourced.
                                                    *SessionLoad-variable*
While the session file is loading the SessionLoad global variable is set to 1.
Plugins can use this to postpone some work until the SessionLoadPost event is
triggered.

                                                          *:mkvie* *:mkview*
:mkvie[w][!] [file]     Write a Vim script that restores the contents of the
                        current window.
                        When [!] is included an existing file is overwritten.
                        When [file] is omitted or is a number from 1 to 9, a
                        name is generated and 'viewdir' prepended.  When the
                        last path part of 'viewdir' does not exist, this
                        directory is created.  E.g., when 'viewdir' is
                        "$VIM/vimfiles/view" then "view" is created in
                        "$VIM/vimfiles".
                        An existing file is always overwritten then.  Use
                        |:loadview| to load this view again.
                        When [file] is the name of a file ('viewdir' is not
                        used), a command to edit the file is added to the
                        generated file.

The output of ":mkview" contains these items:
1. The argument list used in the window.  When the global argument list is
   used it is reset to the global list.
   The index in the argument list is also restored.
2. The file being edited in the window.  If there is no file, the window is
   made empty.
3. Restore mappings, abbreviations and options local to the window if
   'viewoptions' contains "options" or "localoptions".  For the options it
   restores only values that are local to the current buffer and values local
   to the window.
```

```
   When storing the view as part of a session and "options" is in
   'sessionoptions', global values for local options will be stored too.
4. Restore folds when using manual folding and 'viewoptions' contains
   "folds".  Restore manually opened and closed folds.
5. The scroll position and the cursor position in the file.  Doesn't work very
   well when there are closed folds.
6. The local current directory, if it is different from the global current
   directory.

Note that Views and Sessions are not perfect:
- They don't restore everything.  For example, defined functions, autocommands
  and ":syntax on" are not included.  Things like register contents and
  command line history are in viminfo, not in Sessions or Views.
- Global option values are only set when they differ from the default value.
  When the current value is not the default value, loading a Session will not
  set it back to the default value.  Local options will be set back to the
  default value though.
- Existing mappings will be overwritten without warning.  An existing mapping
  may cause an error for ambiguity.
- When storing manual folds and when storing manually opened/closed folds,
  changes in the file between saving and loading the view will mess it up.
- The Vim script is not very efficient.  But still faster than typing the
  commands yourself!

                                                        *:lo* *:loadview*
:lo[adview] [nr]        Load the view for the current file.  When [nr] is
                        omitted, the view stored with ":mkview" is loaded.
                        When [nr] is specified, the view stored with ":mkview
                        [nr]" is loaded.

The combination of ":mkview" and ":loadview" can be used to store up to ten
different views of a file.  These are remembered in the directory specified
with the 'viewdir' option.  The views are stored using the file name.  If a
file is renamed or accessed through a (symbolic) link the view will not be
found.

You might want to clean up your 'viewdir' directory now and then.

To automatically save and restore views for *.c files: >
        au BufWinLeave *.c mkview
        au BufWinEnter *.c silent loadview

==============================================================================
10. The viminfo file                            *viminfo* *viminfo-file* *E136*
                                                *E575* *E576* *E577*
If you exit Vim and later start it again, you would normally lose a lot of
information.  The viminfo file can be used to remember that information, which
enables you to continue where you left off.

This is introduced in section |21.3| of the user manual.

The viminfo file is used to store:
- The command line history.
- The search string history.
- The input-line history.
- Contents of non-empty registers.
- Marks for several files.
- File marks, pointing to locations in files.
- Last search/substitute pattern (for 'n' and '&').
- The buffer list.
- Global variables.
```

The viminfo file is not supported when the |+viminfo| feature has been
disabled at compile time.

You could also use a Session file. The difference is that the viminfo file
does not depend on what you are working on. There normally is only one
viminfo file. Session files are used to save the state of a specific editing
Session. You could have several Session files, one for each project you are
working on. Viminfo and Session files together can be used to effectively
enter Vim and directly start working in your desired setup. |session-file|

viminfo-read
When Vim is started and the 'viminfo' option is non-empty, the contents of
the viminfo file are read and the info can be used in the appropriate places.
The |v:oldfiles| variable is filled. The marks are not read in at startup
(but file marks are). See |initialization| for how to set the 'viminfo'
option upon startup.

viminfo-write
When Vim exits and 'viminfo' is non-empty, the info is stored in the viminfo
file (it's actually merged with the existing one, if one exists). The
'viminfo' option is a string containing information about what info should be
stored, and contains limits on how much should be stored (see 'viminfo').

Merging happens in two ways. Most items that have been changed or set in the
current Vim session are stored, and what was not changed is filled from what
is currently in the viminfo file. For example:
- Vim session A reads the viminfo, which contains variable START.
- Vim session B does the same
- Vim session A sets the variables AAA and BOTH and exits
- Vim session B sets the variables BBB and BOTH and exits
Now the viminfo will have:
   START - it was in the viminfo and wasn't changed in session A or B
   AAA   - value from session A, session B kept it
   BBB   - value from session B
   BOTH - value from session B, value from session A is lost

viminfo-timestamp
For some items a timestamp is used to keep the last changed version. Here it
doesn't matter in which sequence Vim sessions exit, the newest item(s) are
always kept. This is used for:
- The command line history.
- The search string history.
- The input-line history.
- Contents of non-empty registers.
- The jump list
- File marks
The timestamp feature was added before Vim 8.0. Older versions of Vim,
starting with 7.4.1131, will keep the items with timestamp, but not use them.
Thus when using both an older and a newer version of Vim the most recent data
will be kept.

Notes for Unix:
- The file protection for the viminfo file will be set to prevent other users
  from being able to read it, because it may contain any text or commands that
  you have worked with.
- If you want to share the viminfo file with other users (e.g. when you "su"
  to another user), you can make the file writable for the group or everybody.
  Vim will preserve this when writing new viminfo files. Be careful, don't
  allow just anybody to read and write your viminfo file!
- Vim will not overwrite a viminfo file that is not writable by the current
  "real" user. This helps for when you did "su" to become root, but your
  $HOME is still set to a normal user's home directory. Otherwise Vim would

  create a viminfo file owned by root that nobody else can read.
- The viminfo file cannot be a symbolic link. This is to avoid security
  issues.

Marks are stored for each file separately. When a file is read and 'viminfo'
is non-empty, the marks for that file are read from the viminfo file. NOTE:
The marks are only written when exiting Vim, which is fine because marks are
remembered for all the files you have opened in the current editing session,
unless ":bdel" is used. If you want to save the marks for a file that you are
about to abandon with ":bdel", use ":wv". The '[' and ']' marks are not
stored, but the '"' mark is. The '"' mark is very useful for jumping to the
cursor position when the file was last exited. No marks are saved for files
that start with any string given with the "r" flag in 'viminfo'. This can be
used to avoid saving marks for files on removable media (for MS-DOS you would
use "ra:,rb:", for Amiga "rdf0:,rdf1:,rdf2:").
The |v:oldfiles| variable is filled with the file names that the viminfo file
has marks for.

                                                        viminfo-file-marks
Uppercase marks ('A to 'Z) are stored when writing the viminfo file. The
numbered marks ('0 to '9) are a bit special. When the viminfo file is written
(when exiting or with the ":wviminfo" command), '0 is set to the current cursor
position and file. The old '0 is moved to '1, '1 to '2, etc. This
resembles what happens with the "1 to "9 delete registers. If the current
cursor position is already present in '0 to '9, it is moved to '0, to avoid
having the same position twice. The result is that with "'0", you can jump
back to the file and line where you exited Vim. To do that right away, try
using this command: >

        vim -c "normal '0"

In a csh compatible shell you could make an alias for it: >

        alias lvim vim -c '"'normal "'"0'"'

For a bash-like shell: >

        alias lvim='vim -c "normal '\''0"'

Use the "r" flag in 'viminfo' to specify for which files no marks should be
remembered.

VIMINFO FILE NAME   *viminfo-file-name*

- The default name of the viminfo file is "$HOME/.viminfo" for Unix and OS/2,
  "s:.viminfo" for Amiga, "$HOME_viminfo" for MS-DOS and Win32. For the last
  two, when $HOME is not set, "$VIM_viminfo" is used. When $VIM is also not
  set, "c:_viminfo" is used. For OS/2 "$VIM/.viminfo" is used when $HOME is
  not set and $VIM is set.
- The 'n' flag in the 'viminfo' option can be used to specify another viminfo
  file name |'viminfo'|.
- The "-i" Vim argument can be used to set another file name, |-i|. When the
  file name given is "NONE" (all uppercase), no viminfo file is ever read or
  written. Also not for the commands below!
- For the commands below, another file name can be given, overriding the
  default and the name given with 'viminfo' or "-i" (unless it's NONE).

CHARACTER ENCODING   *viminfo-encoding*

The text in the viminfo file is encoded as specified with the 'encoding'

option. Normally you will always work with the same 'encoding' value, and this works just fine. However, if you read the viminfo file with another value for 'encoding' than what it was written with, some of the text (non-ASCII characters) may be invalid. If this is unacceptable, add the 'c' flag to the 'viminfo' option: >
	:set viminfo+=c
Vim will then attempt to convert the text in the viminfo file from the 'encoding' value it was written with to the current 'encoding' value. This requires Vim to be compiled with the |+iconv| feature. Filenames are not converted.

MANUALLY READING AND WRITING *viminfo-read-write*

Two commands can be used to read and write the viminfo file manually. This can be used to exchange registers between two running Vim programs: First type ":wv" in one and then ":rv" in the other. Note that if the register already contained something, then ":rv!" would be required. Also note however that this means everything will be overwritten with information from the first Vim, including the command line history, etc.

The viminfo file itself can be edited by hand too, although we suggest you start with an existing one to get the format right. It is reasonably self-explanatory once you're in there. This can be useful in order to create a second file, say "~/.my_viminfo" which could contain certain settings that you always want when you first start Vim. For example, you can preload registers with particular data, or put certain commands in the command line history. A line in your .vimrc file like >
	:rviminfo! ~/.my_viminfo
can be used to load this information. You could even have different viminfos for different types of files (e.g., C code) and load them based on the file name, using the ":autocmd" command (see |:autocmd|).

viminfo-errors
When Vim detects an error while reading a viminfo file, it will not overwrite that file. If there are more than 10 errors, Vim stops reading the viminfo file. This was done to avoid accidentally destroying a file when the file name of the viminfo file is wrong. This could happen when accidentally typing "vim -i file" when you wanted "vim -R file" (yes, somebody accidentally did that!). If you want to overwrite a viminfo file with an error in it, you will either have to fix the error, or delete the file (while Vim is running, so most of the information will be restored).

:rv *:rviminfo* *E195*
:rv[iminfo][!] [file] Read from viminfo file [file] (default: see above). If [!] is given, then any information that is already set (registers, marks, |v:oldfiles|, etc.) will be overwritten {not in Vi}

:wv *:wviminfo* *E137* *E138* *E574* *E886* *E929*
:wv[iminfo][!] [file] Write to viminfo file [file] (default: see above). The information in the file is first read in to make a merge between old and new info. When [!] is used, the old information is not read first, only the internal info is written. If 'viminfo' is empty, marks for up to 100 files will be written.
When you get error "E929: Too many viminfo temp files" check that no old temp files were left behind (e.g. ~/.viminf*) and that you can write in the directory of the .viminfo file.
{not in Vi}

```
                                                *:ol* *:oldfiles*
:ol[dfiles]             List the files that have marks stored in the viminfo
                        file.  This list is read on startup and only changes
                        afterwards with `:rviminfo!`.  Also see |v:oldfiles|.
                        The number can be used with |c_#<|.
                        The output can be filtered with |:filter|, e.g.: >
                                filter /\.vim/ oldfiles
<                       The filtering happens on the file name.
                        {not in Vi, only when compiled with the |+eval|
                        feature}

:bro[wse] ol[dfiles][!]
                        List file names as with |:oldfiles|, and then prompt
                        for a number.  When the number is valid that file from
                        the list is edited.
                        If you get the |press-enter| prompt you can press "q"
                        and still get the prompt to enter a file number.
                        Use ! to abandon a modified buffer. |abandon|
                        {not when compiled with tiny or small features}

 vim:tw=78:ts=8:ft=help:norl:
*editing.txt*   For Vim version 8.0.  Last change: 2017 Aug 21


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Editing files                                           *edit-files*

1.  Introduction               |edit-intro|
2.  Editing a file             |edit-a-file|
3.  The argument list          |argument-list|
4.  Writing                    |writing|
5.  Writing and quitting       |write-quit|
6.  Dialogs                    |edit-dialogs|
7.  The current directory      |current-directory|
8.  Editing binary files       |edit-binary|
9.  Encryption                 |encryption|
10. Timestamps                 |timestamps|
11. File Searching             |file-searching|

==============================================================================
1. Introduction                                         *edit-intro*

Editing a file with Vim means:

1. reading the file into a buffer
2. changing the buffer with editor commands
3. writing the buffer into a file

                                                        *current-file*
As long as you don't write the buffer, the original file remains unchanged.
If you start editing a file (read a file into the buffer), the file name is
remembered as the "current file name".  This is also known as the name of the
current buffer.  It can be used with "%" on the command line |:_%|.

                                                        *alternate-file*
If there already was a current file name, then that one becomes the alternate
file name.  It can be used with "#" on the command line |:_#| and you can use
the |CTRL-^| command to toggle between the current and the alternate file.
However, the alternate file name is not changed when |:keepalt| is used.
An alternate file name is remembered for each window.
```

:keepalt *:keepa*
:keepalt {cmd} Execute {cmd} while keeping the current alternate file name. Note that commands invoked indirectly (e.g., with a function) may still set the alternate file name. {not in Vi}

All file names are remembered in the buffer list. When you enter a file name, for editing (e.g., with ":e filename") or writing (e.g., with ":w filename"), the file name is added to the list. You can use the buffer list to remember which files you edited and to quickly switch from one file to another (e.g., to copy text) with the |CTRL-^| command. First type the number of the file and then hit CTRL-^. {Vi: only one alternate file name is remembered}

CTRL-G or *CTRL-G* *:f* *:fi* *:file*
:f[ile] Prints the current file name (as typed, unless ":cd" was used), the cursor position (unless the 'ruler' option is set), and the file status (readonly, modified, read errors, new file). See the 'shortmess' option about how to make this message shorter. {Vi does not include column number}

:f[ile]! like |:file|, but don't truncate the name even when 'shortmess' indicates this.

{count}CTRL-G Like CTRL-G, but prints the current file name with full path. If the count is higher than 1 the current buffer number is also given. {not in Vi}

g_CTRL-G *word-count* *byte-count*
g CTRL-G Prints the current position of the cursor in five ways: Column, Line, Word, Character and Byte. If the number of Characters and Bytes is the same then the Character position is omitted.
If there are characters in the line that take more than one position on the screen (<Tab> or special character), both the "real" column and the screen column are shown, separated with a dash.
Also see the 'ruler' option and the |wordcount()| function.
{not in Vi}

v_g_CTRL-G
{Visual}g CTRL-G Similar to "g CTRL-G", but Word, Character, Line, and Byte counts for the visually selected region are displayed.
In Blockwise mode, Column count is also shown. (For {Visual} see |Visual-mode|.)
{not in VI}

:file_f
:f[ile][!] {name} Sets the current file name to {name}. The optional ! avoids truncating the message, as with |:file|.
If the buffer did have a name, that name becomes the |alternate-file| name. An unlisted buffer is created to hold the old name.

:0file
:0f[ile][!] Remove the name of the current buffer. The optional ! avoids truncating the message, as with |:file|. {not in Vi}

```
:buffers
:files
:ls                     List all the currently known file names.  See
                        'windows.txt' |:files| |:buffers| |:ls|.  {not in
                        Vi}
```

Vim will remember the full path name of a file name that you enter. In most cases when the file name is displayed only the name you typed is shown, but the full path name is being used if you used the ":cd" command |:cd|.

home-replace

If the environment variable $HOME is set, and the file name starts with that string, it is often displayed with HOME replaced with "~". This was done to keep file names short. When reading or writing files the full name is still used, the "~" is only used when displaying file names. When replacing the file name would result in just "~", "~/" is used instead (to avoid confusion between options set to $HOME with 'backupext' set to "~").

When writing the buffer, the default is to use the current file name. Thus when you give the "ZZ" or ":wq" command, the original file will be overwritten. If you do not want this, the buffer can be written into another file by giving a file name argument to the ":write" command. For example: >

```
vim testfile
[change the buffer with editor commands]
:w newfile
:q
```

This will create a file "newfile", that is a modified copy of "testfile". The file "testfile" will remain unchanged. Anyway, if the 'backup' option is set, Vim renames or copies the original file before it will be overwritten. You can use this file if you discover that you need the original file. See also the 'patchmode' option. The name of the backup file is normally the same as the original file with 'backupext' appended. The default "~" is a bit strange to avoid accidentally overwriting existing files. If you prefer ".bak" change the 'backupext' option. Extra dots are replaced with '_' on MS-DOS machines, when Vim has detected that an MS-DOS-like filesystem is being used (e.g., messydos or crossdos) or when the 'shortname' option is on. The backup file can be placed in another directory by setting 'backupdir'.

auto-shortname

Technical: On the Amiga you can use 30 characters for a file name. But on an MS-DOS-compatible filesystem only 8 plus 3 characters are available. Vim tries to detect the type of filesystem when it is creating the .swp file. If an MS-DOS-like filesystem is suspected, a flag is set that has the same effect as setting the 'shortname' option. This flag will be reset as soon as you start editing a new file. The flag will be used when making the file name for the ".swp" and ".~" files for the current file. But when you are editing a file in a normal filesystem and write to an MS-DOS-like filesystem the flag will not have been set. In that case the creation of the ".~" file may fail and you will get an error message. Use the 'shortname' option in this case.

When you started editing without giving a file name, "No File" is displayed in messages. If the ":write" command is used with a file name argument, the file name for the current file is set to that file name. This only happens when the 'F' flag is included in 'cpoptions' (by default it is included) |cpo-F|. This is useful when entering text in an empty buffer and then writing it to a file. If 'cpoptions' contains the 'f' flag (by default it is NOT included) |cpo-f| the file name is set for the ":read file" command. This is useful when starting Vim without an argument and then doing ":read file" to start

editing a file.
When the file name was set and 'filetype' is empty the filetype detection
autocommands will be triggered.

not-edited
Because the file name was set without really starting to edit that file, you
are protected from overwriting that file. This is done by setting the
"notedited" flag. You can see if this flag is set with the CTRL-G or ":file"
command. It will include "[Not edited]" when the "notedited" flag is set.
When writing the buffer to the current file name (with ":w!"), the "notedited"
flag is reset.

abandon
Vim remembers whether you have changed the buffer. You are protected from
losing the changes you made. If you try to quit without writing, or want to
start editing another file, Vim will refuse this. In order to overrule this
protection, add a '!' to the command. The changes will then be lost. For
example: ":q" will not work if the buffer was changed, but ":q!" will. To see
whether the buffer was changed use the "CTRL-G" command. The message includes
the string "[Modified]" if the buffer has been changed, or "+" if the 'm' flag
is in 'shortmess'.

If you want to automatically save the changes without asking, switch on the
'autowriteall' option. 'autowrite' is the associated Vi-compatible option
that does not work for all commands.

If you want to keep the changed buffer without saving it, switch on the
'hidden' option. See |hidden-buffer|. Some commands work like this even when
'hidden' is not set, check the help for the command.

==
2. Editing a file *edit-a-file*

:e *:edit* *reload*
:e[dit] [++opt] [+cmd] Edit the current file. This is useful to re-edit the
current file, when it has been changed outside of Vim.
This fails when changes have been made to the current
buffer and 'autowriteall' isn't set or the file can't
be written.
Also see |++opt| and |+cmd|.
{Vi: no ++opt}

:edit! *discard*
:e[dit]! [++opt] [+cmd]
Edit the current file always. Discard any changes to
the current buffer. This is useful if you want to
start all over again.
Also see |++opt| and |+cmd|.
{Vi: no ++opt}

:edit_f
:e[dit] [++opt] [+cmd] {file}
Edit {file}.
This fails when changes have been made to the current
buffer, unless 'hidden' is set or 'autowriteall' is
set and the file can be written.
Also see |++opt| and |+cmd|.
{Vi: no ++opt}

:edit!_f
:e[dit]! [++opt] [+cmd] {file}
Edit {file} always. Discard any changes to the
current buffer.

```
                        Also see |++opt| and |+cmd|.
                        {Vi: no ++opt}

:e[dit] [++opt] [+cmd] #[count]
                        Edit the [count]th buffer (as shown by |:files|).
                        This command does the same as [count] CTRL-^.  But ":e
                        #" doesn't work if the alternate buffer doesn't have a
                        file name, while CTRL-^ still works then.
                        Also see |++opt| and |+cmd|.
                        {Vi: no ++opt}

                                                        *:ene* *:enew*
:ene[w]                 Edit a new, unnamed buffer.  This fails when changes
                        have been made to the current buffer, unless 'hidden'
                        is set or 'autowriteall' is set and the file can be
                        written.
                        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.
                        {not in Vi}

                                                        *:ene!* *:enew!*
:ene[w]!                Edit a new, unnamed buffer.  Discard any changes to
                        the current buffer.
                        Set 'fileformat' like |:enew|.
                        {not in Vi}

                                                        *:fin* *:find*
:fin[d][!] [++opt] [+cmd] {file}
                        Find {file} in 'path' and then |:edit| it.
                        {not in Vi} {not available when the |+file_in_path|
                        feature was disabled at compile time}

:{count}fin[d][!] [++opt] [+cmd] {file}
                        Just like ":find", but use the {count} match in
                        'path'.  Thus ":2find file" will find the second
                        "file" found in 'path'.  When there are fewer matches
                        for the file in 'path' than asked for, you get an
                        error message.

                                                        *:ex*
:ex [++opt] [+cmd] [file]
                        Same as |:edit|.

                                                        *:vi* *:visual*
:vi[sual][!] [++opt] [+cmd] [file]
                        When used in Ex mode: Leave |Ex-mode|, go back to
                        Normal mode.  Otherwise same as |:edit|.

                                                        *:vie* *:view*
:vie[w][!] [++opt] [+cmd] file
                        When used in Ex mode: Leave |Ex-mode|, go back to
                        Normal mode.  Otherwise same as |:edit|, but set
                        'readonly' option for this buffer.  {not in Vi}

                                                        *CTRL-^* *CTRL-6*
CTRL-^                  Edit the alternate file.  Mostly the alternate file is
                        the previously edited file.  This is a quick way to
                        toggle between two files.  It is equivalent to ":e #",
                        except that it also works when there is no file name.

                        If the 'autowrite' or 'autowriteall' option is on and
```

```
                        the buffer was changed, write it.
                        Mostly the ^ character is positioned on the 6 key,
                        pressing CTRL and 6 then gets you what we call CTRL-^.
                        But on some non-US keyboards CTRL-^ is produced in
                        another way.

{count}CTRL-^           Edit [count]th file in the buffer list (equivalent to
                        ":e #[count]").  This is a quick way to switch between
                        files.
                        See |CTRL-^| above for further details.
                        {not in Vi}

[count]]f                                               *]f* *[f*
[count][f               Same as "gf".  Deprecated.

                                                        *gf* *E446* *E447*
[count]gf               Edit the file whose name is under or after the cursor.
                        Mnemonic: "goto file".
                        Uses the 'isfname' option to find out which characters
                        are supposed to be in a file name.  Trailing
                        punctuation characters ".,:;!" are ignored. Escaped
                        spaces "\ " are reduced to a single space.
                        Uses the 'path' option as a list of directory names to
                        look for the file.  See the 'path' option for details
                        about relative directories and wildcards.
                        Uses the 'suffixesadd' option to check for file names
                        with a suffix added.
                        If the file can't be found, 'includeexpr' is used to
                        modify the name and another attempt is done.
                        If a [count] is given, the count'th file that is found
                        in the 'path' is edited.
                        This command fails if Vim refuses to |abandon| the
                        current file.
                        If you want to edit the file in a new window use
                        |CTRL-W_CTRL-F|.
                        If you do want to edit a new file, use: >
                                :e <cfile>
<                       To make gf always work like that: >
                                :map gf :e <cfile><CR>
<                       If the name is a hypertext link, that looks like
                        "type://machine/path", you need the |netrw| plugin.
                        For Unix the '~' character is expanded, like in
                        "~user/file".  Environment variables are expanded too
                        |expand-env|.
                        {not in Vi}
                        {not available when the |+file_in_path| feature was
                        disabled at compile time}

                                                        *v_gf*
{Visual}[count]gf       Same as "gf", but the highlighted text is used as the
                        name of the file to edit.  'isfname' is ignored.
                        Leading blanks are skipped, otherwise all blanks and
                        special characters are included in the file name.
                        (For {Visual} see |Visual-mode|.)
                        {not in VI}

                                                        *gF*
[count]gF               Same as "gf", except if a number follows the file
                        name, then the cursor is positioned on that line in
                        the file. The file name and the number must be
                        separated by a non-filename (see 'isfname') and
                        non-numeric character. White space between the
```

filename, the separator and the number are ignored.
Examples:
	eval.c:10 ~
	eval.c @ 20 ~
	eval.c (30) ~
	eval.c 40 ~

v_gF
{Visual}[count]gF	Same as "v_gf".

These commands are used to start editing a single file. This means that the file is read into the buffer and the current file name is set. The file that is opened depends on the current directory, see |:cd|.

See |read-messages| for an explanation of the message that is given after the file has been read.

You can use the ":e!" command if you messed up the buffer and want to start all over again. The ":e" command is only useful if you have changed the current file name.

:filename *{file}*
Besides the things mentioned here, more special items for where a filename is expected are mentioned at |cmdline-special|.

Note for systems other than Unix: When using a command that accepts a single file name (like ":edit file") spaces in the file name are allowed, but trailing spaces are ignored. This is useful on systems that regularly embed spaces in file names (like MS-Windows and the Amiga). Example: The command ":e Long File Name " will edit the file "Long File Name". When using a command that accepts more than one file name (like ":next file1 file2") embedded spaces must be escaped with a backslash.

wildcard *wildcards*
Wildcards in {file} are expanded, but as with file completion, 'wildignore' and 'suffixes' apply. Which wildcards are supported depends on the system. These are the common ones:
	?	matches one character
	*	matches anything, including nothing
	**	matches anything, including nothing, recurses into directories
	[abc]	match 'a', 'b' or 'c'

To avoid the special meaning of the wildcards prepend a backslash. However, on MS-Windows the backslash is a path separator and "path\[abc]" is still seen as a wildcard when "[" is in the 'isfname' option. A simple way to avoid this is to use "path\[[]abc]", this matches the file "path\[abc]".

starstar-wildcard
Expanding "**" is possible on Unix, Win32, Mac OS/X and a few other systems. This allows searching a directory tree. This goes up to 100 directories deep. Note there are some commands where this works slightly differently, see |file-searching|.
Example: >
	:n **/*.txt
Finds files:
	aaa.txt ~
	subdir/bbb.txt ~
	a/b/c/d/ccc.txt ~
When non-wildcard characters are used right before or after "**" these are only matched in the top directory. They are not used for directories further down in the tree. For example: >
	:n /usr/inc**/types.h

```
Finds files:
        /usr/include/types.h ~
        /usr/include/sys/types.h ~
        /usr/inc/old/types.h ~
Note that the path with "/sys" is included because it does not need to match
"/inc".  Thus it's like matching "/usr/inc*/*/*...", not
"/usr/inc*/inc*/inc*".

                                          *backtick-expansion* *`-expansion*
On Unix and a few other systems you can also use backticks for the file name
argument, for example: >
        :next `find . -name ver\\*.c -print`
        :view `ls -t *.patch  \| head -n1`
The backslashes before the star are required to prevent the shell from
expanding "ver*.c" prior to execution of the find program.  The backslash
before the shell pipe symbol "|" prevents Vim from parsing it as command
termination.
This also works for most other systems, with the restriction that the
backticks must be around the whole item.  It is not possible to have text
directly before the first or just after the last backtick.

                                                              *`=*
You can have the backticks expanded as a Vim expression, instead of as an
external command, by putting an equal sign right after the first backtick,
e.g.: >
        :e `=tempname()`
The expression can contain just about anything, thus this can also be used to
avoid the special meaning of '"', '|', '%' and '#'.  However, 'wildignore'
does apply like to other wildcards.

Environment variables in the expression are expanded when evaluating the
expression, thus this works: >
        :e `=$HOME . '/.vimrc'`
This does not work, $HOME is inside a string and used literally: >
        :e `='$HOME' . '/.vimrc'`

If the expression returns a string then names are to be separated with line
breaks.  When the result is a |List| then each item is used as a name.  Line
breaks also separate names.
Note that such expressions are only supported in places where a filename is
expected as an argument to an Ex-command.

                                                              *++opt* *[++opt]*
The [++opt] argument can be used to force the value of 'fileformat',
'fileencoding' or 'binary' to a value for one command, and to specify the
behavior for bad characters.  The form is: >
        ++{optname}
Or: >
        ++{optname}={value}

Where {optname} is one of:          *++ff* *++enc* *++bin* *++nobin* *++edit*
    ff     or  fileformat   overrides 'fileformat'
    enc    or  encoding     overrides 'fileencoding'
    bin    or  binary       sets 'binary'
    nobin  or  nobinary     resets 'binary'
    bad                     specifies behavior for bad characters
    edit                    for |:read| only: keep option values as if editing
                            a file

{value} cannot contain white space.  It can be any valid value for these
options.  Examples: >
        :e ++ff=unix
```

```
This edits the same file again with 'fileformat' set to "unix". >

        :w ++enc=latin1 newfile
This writes the current buffer to "newfile" in latin1 format.

There may be several ++opt arguments, separated by white space.  They must all
appear before any |+cmd| argument.

                                                                    *++bad*
The argument of "++bad=" specifies what happens with characters that can't be
converted and illegal bytes.  It can be one of three things:
    ++bad=X      A single-byte character that replaces each bad character.
    ++bad=keep   Keep bad characters without conversion.  Note that this may
                 result in illegal bytes in your text!
    ++bad=drop   Remove the bad characters.

The default is like "++bad=?": Replace each bad character with a question
mark.  In some places an inverted question mark is used (0xBF).

Note that not all commands use the ++bad argument, even though they do not
give an error when you add it.  E.g. |:write|.

Note that when reading, the 'fileformat' and 'fileencoding' options will be
set to the used format.  When writing this doesn't happen, thus a next write
will use the old value of the option.  Same for the 'binary' option.


                                                          *+cmd* *[+cmd]*
The [+cmd] argument can be used to position the cursor in the newly opened
file, or execute any other command:
        +               Start at the last line.
        +{num}          Start at line {num}.
        +/{pat}         Start at first line containing {pat}.
        +{command}      Execute {command} after opening the new file.
                        {command} is any Ex command.
To include a white space in the {pat} or {command}, precede it with a
backslash.  Double the number of backslashes. >
        :edit  +/The\ book           file
        :edit  +/dir\ dirname\\      file
        :edit  +set\ dir=c:\\\\temp  file
Note that in the last example the number of backslashes is halved twice: Once
for the "+cmd" argument and once for the ":set" command.

                                                          *file-formats*
The 'fileformat' option sets the <EOL> style for a file:
'fileformat'    characters         name                         ~
  "dos"         <CR><NL> or <NL>   DOS format           *DOS-format*
  "unix"        <NL>               Unix format          *Unix-format*
  "mac"         <CR>               Mac format           *Mac-format*
Previously 'textmode' was used.  It is obsolete now.

When reading a file, the mentioned characters are interpreted as the <EOL>.
In DOS format (default for MS-DOS, OS/2 and Win32), <CR><NL> and <NL> are both
interpreted as the <EOL>.  Note that when writing the file in DOS format,
<CR> characters will be added for each single <NL>.  Also see |file-read|.

When writing a file, the mentioned characters are used for <EOL>.  For DOS
format <CR><NL> is used.  Also see |DOS-format-write|.

You can read a file in DOS format and write it in Unix format.  This will
replace all <CR><NL> pairs by <NL> (assuming 'fileformats' includes "dos"): >
        :e file
```

```
        :set fileformat=unix
        :w
If you read a file in Unix format and write with DOS format, all <NL>
characters will be replaced with <CR><NL> (assuming 'fileformats' includes
"unix"): >
        :e file
        :set fileformat=dos
        :w

If you start editing a new file and the 'fileformats' option is not empty
(which is the default), Vim will try to detect whether the lines in the file
are separated by the specified formats.  When set to "unix,dos", Vim will
check for lines with a single <NL> (as used on Unix and Amiga) or by a <CR>
<NL> pair (MS-DOS).  Only when ALL lines end in <CR><NL>, 'fileformat' is set
to "dos", otherwise it is set to "unix".  When 'fileformats' includes "mac",
and no <NL> characters are found in the file, 'fileformat' is set to "mac".

If the 'fileformat' option is set to "dos" on non-MS-DOS systems the message
"[dos format]" is shown to remind you that something unusual is happening.  On
MS-DOS systems you get the message "[unix format]" if 'fileformat' is set to
"unix".  On all systems but the Macintosh you get the message "[mac format]"
if 'fileformat' is set to "mac".

If the 'fileformats' option is empty and DOS format is used, but while reading
a file some lines did not end in <CR><NL>, "[CR missing]" will be included in
the file message.
If the 'fileformats' option is empty and Mac format is used, but while reading
a file a <NL> was found, "[NL missing]" will be included in the file message.

If the new file does not exist, the 'fileformat' of the current buffer is used
when 'fileformats' is empty.  Otherwise the first format from 'fileformats' is
used for the new file.

Before editing binary, executable or Vim script files you should set the
'binary' option.  A simple way to do this is by starting Vim with the "-b"
option.  This will avoid the use of 'fileformat'.  Without this you risk that
single <NL> characters are unexpectedly replaced with <CR><NL>.

You can encrypt files that are written by setting the 'key' option.  This
provides some security against others reading your files. |encryption|


==============================================================================
3. The argument list                                *argument-list* *arglist*

If you give more than one file name when starting Vim, this list is remembered
as the argument list.  You can jump to each file in this list.

Do not confuse this with the buffer list, which you can see with the
|:buffers| command.  The argument list was already present in Vi, the buffer
list is new in Vim.  Every file name in the argument list will also be present
in the buffer list (unless it was deleted with |:bdel| or |:bwipe|).  But it's
common that names in the buffer list are not in the argument list.

This subject is introduced in section |07.2| of the user manual.

There is one global argument list, which is used for all windows by default.
It is possible to create a new argument list local to a window, see
|:arglocal|.

You can use the argument list with the following commands, and with the
expression functions |argc()| and |argv()|.  These all work on the argument
```

```
list of the current window.

                                                        *:ar* *:args*
:ar[gs]                 Print the argument list, with the current file in
                        square brackets.

:ar[gs] [++opt] [+cmd] {arglist}                        *:args_f*
                        Define {arglist} as the new argument list and edit
                        the first one.  This fails when changes have been made
                        and Vim does not want to |abandon| the current buffer.
                        Also see |++opt| and |+cmd|.
                        {Vi: no ++opt}

:ar[gs]! [++opt] [+cmd] {arglist}                       *:args_f!*
                        Define {arglist} as the new argument list and edit
                        the first one.  Discard any changes to the current
                        buffer.
                        Also see |++opt| and |+cmd|.
                        {Vi: no ++opt}

:[count]arge[dit][!] [++opt] [+cmd] {name} ..           *:arge* *:argedit*
                        Add {name}s to the argument list and edit it.
                        When {name} already exists in the argument list, this
                        entry is edited.
                        This is like using |:argadd| and then |:edit|.
                        Spaces in filenames have to be escaped with "\".
                        [count] is used like with |:argadd|.
                        If the current file cannot be |abandon|ed {name}s will
                        still be added to the argument list, but won't be
                        edited. No check for duplicates is done.
                        Also see |++opt| and |+cmd|.
                        {not in Vi}

:[count]arga[dd] {name} ..                      *:arga* *:argadd* *E479*
:[count]arga[dd]
                        Add the {name}s to the argument list.  When {name} is
                        omitted add the current buffer name to the argument
                        list.
                        If [count] is omitted, the {name}s are added just
                        after the current entry in the argument list.
                        Otherwise they are added after the [count]'th file.
                        If the argument list is "a b c", and "b" is the
                        current argument, then these commands result in:
                                command         new argument list ~
                                :argadd x       a b x c
                                :0argadd x      x a b c
                                :1argadd x      a x b c
                                :$argadd x      a b c x
                        And after the last one:
                                :+2argadd y     a b c x y
                        There is no check for duplicates, it is possible to
                        add a file to the argument list twice.
                        The currently edited file is not changed.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}
                        Note: you can also use this method: >
                                :args ## x
<                       This will add the "x" item and sort the new list.

:argd[elete] {pattern} ..                       *:argd* *:argdelete* *E480*
                        Delete files from the argument list that match the
                        {pattern}s.  {pattern} is used like a file pattern,
```

```
                        see |file-pattern|.  "%" can be used to delete the
                        current entry.
                        This command keeps the currently edited file, also
                        when it's deleted from the argument list.
                        Example: >
                                :argdel *.obj
<                       {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

:[range]argd[elete]     Delete the {range} files from the argument list.
                        Example: >
                                :10,$argdel
<                       Deletes arguments 10 and further, keeping 1-9. >
                                :$argd
<                       Deletes just the last one. >
                                :argd
                                :.argd
<                       Deletes the current argument. >
                                :%argd
<                       Removes all the files from the arglist.
                        When the last number in the range is too high, up to
                        the last argument is deleted.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

                                                        *:argu* *:argument*
:[count]argu[ment] [count] [++opt] [+cmd]
                        Edit file [count] in the argument list.  When [count]
                        is omitted the current entry is used.  This fails
                        when changes have been made and Vim does not want to
                        |abandon| the current buffer.
                        Also see |++opt| and |+cmd|.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

:[count]argu[ment]! [count] [++opt] [+cmd]
                        Edit file [count] in the argument list, discard any
                        changes to the current buffer.  When [count] is
                        omitted the current entry is used.
                        Also see |++opt| and |+cmd|.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}

:[count]n[ext] [++opt] [+cmd]                   *:n* *:ne* *:next* *E165* *E163*
                        Edit [count] next file.  This fails when changes have
                        been made and Vim does not want to |abandon| the
                        current buffer.  Also see |++opt| and |+cmd|.  {Vi: no
                        count or ++opt}.

:[count]n[ext]! [++opt] [+cmd]
                        Edit [count] next file, discard any changes to the
                        buffer.  Also see |++opt| and |+cmd|.  {Vi: no count
                        or ++opt}.

:n[ext] [++opt] [+cmd] {arglist}                        *:next_f*
                        Same as |:args_f|.

:n[ext]! [++opt] [+cmd] {arglist}
                        Same as |:args_f!|.

:[count]N[ext] [count] [++opt] [+cmd]                   *:Next* *:N* *E164*
                        Edit [count] previous file in argument list.  This
```

```
                        fails when changes have been made and Vim does not
                        want to |abandon| the current buffer.
                        Also see |++opt| and |+cmd|.  {Vi: no count or ++opt}.

:[count]N[ext]! [count] [++opt] [+cmd]
                        Edit [count] previous file in argument list.  Discard
                        any changes to the buffer.  Also see |++opt| and
                        |+cmd|.  {Vi: no count or ++opt}.

:[count]prev[ious] [count] [++opt] [+cmd]               *:prev* *:previous*
                        Same as :Next.  Also see |++opt| and |+cmd|.  {Vi:
                        only in some versions}

                                                        *:rew* *:rewind*
:rew[ind] [++opt] [+cmd]
                        Start editing the first file in the argument list.
                        This fails when changes have been made and Vim does
                        not want to |abandon| the current buffer.
                        Also see |++opt| and |+cmd|. {Vi: no ++opt}

:rew[ind]! [++opt] [+cmd]
                        Start editing the first file in the argument list.
                        Discard any changes to the buffer.  Also see |++opt|
                        and |+cmd|. {Vi: no ++opt}

                                                        *:fir* *:first*
:fir[st][!] [++opt] [+cmd]
                        Other name for ":rewind". {not in Vi}

                                                        *:la* *:last*
:la[st] [++opt] [+cmd]
                        Start editing the last file in the argument list.
                        This fails when changes have been made and Vim does
                        not want to |abandon| the current buffer.
                        Also see |++opt| and |+cmd|.  {not in Vi}

:la[st]! [++opt] [+cmd]
                        Start editing the last file in the argument list.
                        Discard any changes to the buffer.  Also see |++opt|
                        and |+cmd|.  {not in Vi}

                                                        *:wn* *:wnext*
:[count]wn[ext] [++opt]
                        Write current file and start editing the [count]
                        next file.  Also see |++opt| and |+cmd|.  {not in Vi}

:[count]wn[ext] [++opt] {file}
                        Write current file to {file} and start editing the
                        [count] next file, unless {file} already exists and
                        the 'writeany' option is off.  Also see |++opt| and
                        |+cmd|.  {not in Vi}

:[count]wn[ext]! [++opt] {file}
                        Write current file to {file} and start editing the
                        [count] next file.  Also see |++opt| and |+cmd|.  {not
                        in Vi}

:[count]wN[ext][!] [++opt] [file]               *:wN* *:wNext*
:[count]wp[revious][!] [++opt] [file]           *:wp* *:wprevious*
                        Same as :wnext, but go to previous file instead of
                        next.  {not in Vi}
```

The [count] in the commands above defaults to one. For some commands it is
possible to use two counts. The last one (rightmost one) is used.

If no [+cmd] argument is present, the cursor is positioned at the last known
cursor position for the file. If 'startofline' is set, the cursor will be
positioned at the first non-blank in the line, otherwise the last know column
is used. If there is no last known cursor position the cursor will be in the
first line (the last line in Ex mode).

{arglist}
The wildcards in the argument list are expanded and the file names are sorted.
Thus you can use the command "vim *.c" to edit all the C files. From within
Vim the command ":n *.c" does the same.

White space is used to separate file names. Put a backslash before a space or
tab to include it in a file name. E.g., to edit the single file "foo bar": >
	:next foo\ bar

On Unix and a few other systems you can also use backticks, for example: >
	:next `find . -name \\*.c -print`
The backslashes before the star are required to prevent "*.c" to be expanded
by the shell before executing the find program.

arglist-position
When there is an argument list you can see which file you are editing in the
title of the window (if there is one and 'title' is on) and with the file
message you get with the "CTRL-G" command. You will see something like
	(file 4 of 11)
If 'shortmess' contains 'f' it will be
	(4 of 11)
If you are not really editing the file at the current position in the argument
list it will be
	(file (4) of 11)
This means that you are position 4 in the argument list, but not editing the
fourth file in the argument list. This happens when you do ":e file".

LOCAL ARGUMENT LIST

{not in Vi}
{not available when compiled without the |+windows| or |+listcmds| features}

:arglocal
:argl[ocal]		Make a local copy of the global argument list.
			Doesn't start editing another file.

:argl[ocal][!] [++opt] [+cmd] {arglist}
			Define a new argument list, which is local to the
			current window. Works like |:args_f| otherwise.

:argglobal
:argg[lobal]		Use the global argument list for the current window.
			Doesn't start editing another file.

:argg[lobal][!] [++opt] [+cmd] {arglist}
			Use the global argument list for the current window.
			Define a new global argument list like |:args_f|.
			All windows using the global argument list will see
			this new list.

There can be several argument lists. They can be shared between windows.
When they are shared, changing the argument list in one window will also

```
change it in the other window.

When a window is split the new window inherits the argument list from the
current window.  The two windows then share this list, until one of them uses
|:arglocal| or |:argglobal| to use another argument list.


USING THE ARGUMENT LIST

                                                *:argdo*
:[range]argdo[!] {cmd}  Execute {cmd} for each file in the argument list or
                        if [range] is specified only for arguments in that
                        range.  It works like doing this: >
                                :rewind
                                :{cmd}
                                :next
                                :{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 file, further files
                        in the argument list will not be visited.
                        The last file in the argument list (or where an error
                        occurred) becomes the current file.
                        {cmd} can contain '|' to concatenate several commands.
                        {cmd} must not change the argument list.
                        Note: While this command is executing, the Syntax
                        autocommand event is disabled by adding it to
                        'eventignore'.  This considerably speeds up editing
                        each file.
                        {not in Vi} {not available when compiled without the
                        |+listcmds| feature}
                        Also see |:windo|, |:tabdo|, |:bufdo|, |:cdo|, |:ldo|,
                        |:cfdo| and |:lfdo|

Example: >
        :args *.c
        :argdo set ff=unix | update
This sets the 'fileformat' option to "unix" and writes the file if it is now
changed.  This is done for all *.c files.

Example: >
        :args *.[ch]
        :argdo %s/\<my_foo\>/My_Foo/ge | update
This changes the word "my_foo" to "My_Foo" in all *.c and *.h files.  The "e"
flag is used for the ":substitute" command to avoid an error for files where
"my_foo" isn't used.  ":update" writes the file only if changes were made.

==============================================================================
4. Writing                                      *writing* *save-file*

Note: When the 'write' option is off, you are not able to write any file.

                                                        *:w* *:write*
                                                *E502* *E503* *E504* *E505*
                                                *E512* *E514* *E667* *E796*
:w[rite] [++opt]        Write the whole buffer to the current file.  This is
                        the normal way to save changes to a file.  It fails
                        when the 'readonly' option is set or when there is
                        another reason why the file can't be written.
                        For ++opt see |++opt|, but only ++bin, ++nobin, ++ff
                        and ++enc are effective.
```

```
:w[rite]! [++opt]       Like ":write", but forcefully write when 'readonly' is
                        set or there is another reason why writing was
                        refused.
                        Note: This may change the permission and ownership of
                        the file and break (symbolic) links.  Add the 'W' flag
                        to 'cpoptions' to avoid this.

:[range]w[rite][!] [++opt]
                        Write the specified lines to the current file.  This
                        is unusual, because the file will not contain all
                        lines in the buffer.

                                                        *:w_f* *:write_f*
:[range]w[rite] [++opt] {file}
                        Write the specified lines to {file}, unless it
                        already exists and the 'writeany' option is off.

                                                        *:w!*
:[range]w[rite]! [++opt] {file}
                        Write the specified lines to {file}.  Overwrite an
                        existing file.

                                                *:w_a* *:write_a* *E494*
:[range]w[rite][!] [++opt] >>
                        Append the specified lines to the current file.

:[range]w[rite][!] [++opt] >> {file}
                        Append the specified lines to {file}.  '!' forces the
                        write even if file does not exist.

                                                        *:w_c* *:write_c*
:[range]w[rite] [++opt] !{cmd}
                        Execute {cmd} with [range] lines as standard input
                        (note the space in front of the '!').  {cmd} is
                        executed like with ":!{cmd}", any '!' is replaced with
                        the previous command |:!|.

The default [range] for the ":w" command is the whole buffer (1,$).  If you
write the whole buffer, it is no longer considered changed.  When you
write it to a different file with ":w somefile" it depends on the "+" flag in
'cpoptions'.  When included, the write command will reset the 'modified' flag,
even though the buffer itself may still be different from its file.

If a file name is given with ":w" it becomes the alternate file.  This can be
used, for example, when the write fails and you want to try again later with
":w #".  This can be switched off by removing the 'A' flag from the
'cpoptions' option.

                                                        *:sav* *:saveas*
:sav[eas][!] [++opt] {file}
                        Save the current buffer under the name {file} and set
                        the filename of the current buffer to {file}.  The
                        previous name is used for the alternate file name.
                        The [!] is needed to overwrite an existing file.
                        When 'filetype' is empty filetype detection is done
                        with the new name, before the file is written.
                        When the write was successful 'readonly' is reset.
                        {not in Vi}

                                                        *:up* *:update*
:[range]up[date][!] [++opt] [>>] [file]
```

Like ":write", but only write when the buffer has been
modified. {not in Vi}

WRITING WITH MULTIPLE BUFFERS *buffer-write*

:wa *:wall*

:wa[ll] Write all changed buffers. Buffers without a file name cause an error message. Buffers which are readonly are not written. {not in Vi}

:wa[ll]! Write all changed buffers, even the ones that are readonly. Buffers without a file name are not written and cause an error message. {not in Vi}

Vim will warn you if you try to overwrite a file that has been changed elsewhere. See |timestamp|.

backup *E207* *E506* *E507* *E508* *E509* *E510*

If you write to an existing file (but do not append) while the 'backup', 'writebackup' or 'patchmode' option is on, a backup of the original file is made. The file is either copied or renamed (see 'backupcopy'). After the file has been successfully written and when the 'writebackup' option is on and the 'backup' option is off, the backup file is deleted. When the 'patchmode' option is on the backup file may be renamed.

backup-table

'backup'	'writebackup'	action ~
off	off	no backup made
off	on	backup current file, deleted afterwards (default)
on	off	delete old backup, backup current file
on	on	delete old backup, backup current file

When the 'backupskip' pattern matches with the name of the file which is written, no backup file is made. The values of 'backup' and 'writebackup' are ignored then.

When the 'backup' option is on, an old backup file (with the same name as the new backup file) will be deleted. If 'backup' is not set, but 'writebackup' is set, an existing backup file will not be deleted. The backup file that is made while the file is being written will have a different name.

On some filesystems it's possible that in a crash you lose both the backup and the newly written file (it might be there but contain bogus data). In that case try recovery, because the swap file is synced to disk and might still be there. |:recover|

The directories given with the 'backupdir' option are used to put the backup file in. (default: same directory as the written file).

Whether the backup is a new file, which is a copy of the original file, or the original file renamed depends on the 'backupcopy' option. See there for an explanation of when the copy is made and when the file is renamed.

If the creation of a backup file fails, the write is not done. If you want to write anyway add a '!' to the command.

write-permissions

When writing a new file the permissions are read-write. For unix the mask is 0666 with additionally umask applied. When writing a file that was read Vim will preserve the permissions, but clear the s-bit.

write-readonly
When the 'cpoptions' option contains 'W', Vim will refuse to overwrite a
readonly file. When 'W' is not present, ":w!" will overwrite a readonly file,
if the system allows it (the directory must be writable).

write-fail
If the writing of the new file fails, you have to be careful not to lose
your changes AND the original file. If there is no backup file and writing
the new file failed, you have already lost the original file! DON'T EXIT VIM
UNTIL YOU WRITE OUT THE FILE! If a backup was made, it is put back in place
of the original file (if possible). If you exit Vim, and lose the changes
you made, the original file will mostly still be there. If putting back the
original file fails, there will be an error message telling you that you
lost the original file.

DOS-format-write
If the 'fileformat' is "dos", <CR> <NL> is used for <EOL>. This is default
for MS-DOS, Win32 and OS/2. On other systems the message "[dos format]" is
shown to remind you that an unusual <EOL> was used.
Unix-format-write
If the 'fileformat' is "unix", <NL> is used for <EOL>. On MS-DOS, Win32 and
OS/2 the message "[unix format]" is shown.
Mac-format-write
If the 'fileformat' is "mac", <CR> is used for <EOL>. On non-Mac systems the
message "[mac format]" is shown.

See also |file-formats| and the 'fileformat' and 'fileformats' options.

ACL
ACL stands for Access Control List. It is an advanced way to control access
rights for a file. It is used on new MS-Windows and Unix systems, but only
when the filesystem supports it.
Vim attempts to preserve the ACL info when writing a file. The backup file
will get the ACL info of the original file.
The ACL info is also used to check if a file is read-only (when opening the
file).

read-only-share
When MS-Windows shares a drive on the network it can be marked as read-only.
This means that even if the file read-only attribute is absent, and the ACL
settings on NT network shared drives allow writing to the file, you can still
not write to the file. Vim on Win32 platforms will detect read-only network
drives and will mark the file as read-only. You will not be able to override
it with |:write|.

write-device
When the file name is actually a device name, Vim will not make a backup (that
would be impossible). You need to use "!", since the device already exists.
Example for Unix: >
:w! /dev/lpt0
and for MS-DOS or MS-Windows: >
:w! lpt0
For Unix a device is detected when the name doesn't refer to a normal file or
a directory. A fifo or named pipe also looks like a device to Vim.
For MS-DOS and MS-Windows the device is detected by its name:
AUX
CON
CLOCK$
NUL
PRN
COMn n=1,2,3... etc

```
        LPTn    n=1,2,3... etc
The names can be in upper- or lowercase.

==============================================================================
5. Writing and quitting                                 *write-quit*

                                                        *:q* *:quit*
:q[uit]                 Quit the current window.  Quit Vim if this is the last
                        window.  This fails when changes have been made and
                        Vim refuses to |abandon| the current buffer, and when
                        the last file in the argument list has not been
                        edited.
                        If there are other tab pages and quitting the last
                        window in the current tab page the current tab page is
                        closed |tab-page|.
                        Triggers the |QuitPre| autocommand event.
                        See |CTRL-W_q| for quitting another window.

:conf[irm] q[uit]       Quit, but give prompt when changes have been made, or
                        the last file in the argument list has not been
                        edited.  See |:confirm| and 'confirm'.  {not in Vi}

:q[uit]!                Quit without writing, also when the current buffer has
                        changes.  The buffer is unloaded, also when it has
                        'hidden' set.
                        If this is the last window and there is a modified
                        hidden buffer, the current buffer is abandoned and the
                        first changed hidden buffer becomes the current
                        buffer.
                        Use ":qall!" to exit always.

:cq[uit]                Quit always, without writing, and return an error
                        code.  See |:cq|.  Used for Manx's QuickFix mode (see
                        |quickfix|).  {not in Vi}

                                                        *:wq*
:wq [++opt]             Write the current file and quit.  Writing fails when
                        the file is read-only or the buffer does not have a
                        name.  Quitting fails when the last file in the
                        argument list has not been edited.

:wq! [++opt]            Write the current file and quit.  Writing fails when
                        the current buffer does not have a name.

:wq [++opt] {file}      Write to {file} and quit.  Quitting fails when the
                        last file in the argument list has not been edited.

:wq! [++opt] {file}     Write to {file} and quit.

:[range]wq[!] [++opt] [file]
                        Same as above, but only write the lines in [range].

                                                        *:x* *:xit*
:[range]x[it][!] [++opt] [file]
                        Like ":wq", but write only when changes have been
                        made.
                        When 'hidden' is set and there are more windows, the
                        current buffer becomes hidden, after writing the file.

                                                        *:exi* *:exit*
:[range]exi[t][!] [++opt] [file]
                        Same as :xit.
```

ZZ
ZZ Write current file, if modified, and quit (same as ":x"). (Note: If there are several windows for the current file, the file is written if it was modified and the window is closed).

ZQ
ZQ Quit without checking for changes (same as ":q!"). {not in Vi}

MULTIPLE WINDOWS AND BUFFERS *window-exit*

:qa *:qall*
:qa[ll] Exit Vim, unless there are some buffers which have been changed. (Use ":bmod" to go to the next modified buffer). When 'autowriteall' is set all changed buffers will be written, like |:wqall|. {not in Vi}

:conf[irm] qa[ll]
Exit Vim. Bring up a prompt when some buffers have been changed. See |:confirm|. {not in Vi}

:qa[ll]! Exit Vim. Any changes to buffers are lost. {not in Vi}
Also see |:cquit|, it does the same but exits with a non-zero value.

:quita *:quitall*
:quita[ll][!] Same as ":qall". {not in Vi}

:wqa[ll] [++opt] *:wqa* *:wqall* *:xa* *:xall*
:xa[ll] Write all changed buffers and exit Vim. If there are buffers without a file name, which are readonly or which cannot be written for another reason, Vim will not quit. {not in Vi}

:conf[irm] wqa[ll] [++opt]
:conf[irm] xa[ll]
Write all changed buffers and exit Vim. Bring up a prompt when some buffers are readonly or cannot be written for another reason. See |:confirm|. {not in Vi}

:wqa[ll]! [++opt]
:xa[ll]! Write all changed buffers, even the ones that are readonly, and exit Vim. If there are buffers without a file name or which cannot be written for another reason, Vim will not quit. {not in Vi}

==========
6. Dialogs *edit-dialogs*

:confirm *:conf*
:conf[irm] {command} Execute {command}, and use a dialog when an operation has to be confirmed. Can be used on the |:q|, |:qa| and |:w| commands (the latter to override a read-only setting), and any other command that can fail in such a way, such as |:only|, |:buffer|, |:bdelete|, etc.

Examples: >
:confirm w foo
< Will ask for confirmation when "foo" already exists. >
:confirm q

```
<       Will ask for confirmation when there are changes. >
  :confirm qa
<       If any modified, unsaved buffers exist, you will be prompted to save
        or abandon each one.  There are also choices to "save all" or "abandon
        all".

If you want to always use ":confirm", set the 'confirm' option.

                        *:browse* *:bro* *E338* *E614* *E615* *E616*
:bro[wse] {command}     Open a file selection dialog for an argument to
                        {command}.  At present this works for |:e|, |:w|,
                        |:wall|, |:wq|, |:wqall|, |:x|, |:xall|, |:exit|,
                        |:view|, |:sview|, |:r|, |:saveas|, |:sp|, |:mkexrc|,
                        |:mkvimrc|, |:mksession|, |:mkview|, |:split|,
                        |:vsplit|, |:tabe|, |:tabnew|, |:cfile|, |:cgetfile|,
                        |:caddfile|, |:lfile|, |:lgetfile|, |:laddfile|,
                        |:diffsplit|, |:diffpatch|, |:open|, |:pedit|,
                        |:redir|, |:source|, |:update|, |:visual|, |:vsplit|,
                        and |:qall| if 'confirm' is set.
                        {only in Win32, Athena, Motif, GTK and Mac GUI}
                        When ":browse" is not possible you get an error
                        message.  If the |+browse| feature is missing or the
                        {command} doesn't support browsing, the {command} is
                        executed without a dialog.
                        ":browse set" works like |:options|.
                        See also |:oldfiles| for ":browse oldfiles".

The syntax is best shown via some examples: >
        :browse e $vim/foo
<               Open the browser in the $vim/foo directory, and edit the
                file chosen. >
        :browse e
<               Open the browser in the directory specified with 'browsedir',
                and edit the file chosen. >
        :browse w
<               Open the browser in the directory of the current buffer,
                with the current buffer filename as default, and save the
                buffer under the filename chosen. >
        :browse w C:/bar
<               Open the browser in the C:/bar directory, with the current
                buffer filename as default, and save the buffer under the
                filename chosen.
Also see the |'browsedir'| option.
For versions of Vim where browsing is not supported, the command is executed
unmodified.

                                                        *browsefilter*
For MS Windows and GTK, you can modify the filters that are used in the browse
dialog.  By setting the g:browsefilter or b:browsefilter variables, you can
change the filters globally or locally to the buffer.  The variable is set to
a string in the format "{filter label}\t{pattern};{pattern}\n" where {filter
label} is the text that appears in the "Files of Type" comboBox, and {pattern}
is the pattern which filters the filenames.  Several patterns can be given,
separated by ';'.

For Motif the same format is used, but only the very first pattern is actually
used (Motif only offers one pattern, but you can edit it).

For example, to have only Vim files in the dialog, you could use the following
command: >

     let g:browsefilter = "Vim Scripts\t*.vim\nVim Startup Files\t*vimrc\n"
```

```
You can override the filter setting on a per-buffer basis by setting the
b:browsefilter variable.  You would most likely set b:browsefilter in a
filetype plugin, so that the browse dialog would contain entries related to
the type of file you are currently editing.  Disadvantage: This makes it
difficult to start editing a file of a different type.  To overcome this, you
may want to add "All Files\t*.*\n" as the final filter, so that the user can
still access any desired file.

To avoid setting browsefilter when Vim does not actually support it, you can
use has("browsefilter"): >

        if has("browsefilter")
           let g:browsefilter = "whatever"
        endif

==============================================================================
7. The current directory                                *current-directory*

You may use the |:cd| and |:lcd| commands to change to another directory, so
you will not have to type that directory name in front of the file names.  It
also makes a difference for executing external commands, e.g. ":!ls".

Changing directory fails when the current buffer is modified, the '.' flag is
present in 'cpoptions' and "!" is not used in the command.

                                                        *:cd* *E747* *E472*
:cd[!]                  On non-Unix systems: Print the current directory
                        name.  On Unix systems: Change the current directory
                        to the home directory.  Use |:pwd| to print the
                        current directory on all systems.

:cd[!] {path}           Change the current directory to {path}.
                        If {path} is relative, it is searched for in the
                        directories listed in |'cdpath'|.
                        Does not change the meaning of an already opened file,
                        because its full path name is remembered.  Files from
                        the |arglist| may change though!
                        On MS-DOS this also changes the active drive.
                        To change to the directory of the current file: >
                                :cd %:h
<
                                                        *:cd-* *E186*
:cd[!] -                Change to the previous current directory (before the
                        previous ":cd {path}" command). {not in Vi}

                                                        *:chd* *:chdir*
:chd[ir][!] [path]      Same as |:cd|.

                                                        *:lc* *:lcd*
:lc[d][!] {path}        Like |:cd|, but only set the current directory when
                        the cursor is in the current window.  The current
                        directory for other windows is not changed, switching
                        to another window will stop using {path}.
                        {not in Vi}

                                                        *:lch* *:lchdir*
:lch[dir][!]            Same as |:lcd|. {not in Vi}

                                                        *:pw* *:pwd* *E187*
:pw[d]                  Print the current directory name.  {Vi: no pwd}
                        Also see |getcwd()|.
```

So long as no |:lcd| command has been used, all windows share the same current directory. Using a command to jump to another window doesn't change anything for the current directory.
When a |:lcd| command has been used for a window, the specified directory becomes the current directory for that window. Windows where the |:lcd| command has not been used stick to the global current directory. When jumping to another window the current directory will become the last specified local current directory. If none was specified, the global current directory is used.
When a |:cd| command is used, the current window will lose his local current directory and will use the global current directory from now on.

After using |:cd| the full path name will be used for reading and writing files. On some networked file systems this may cause problems. The result of using the full path name is that the file names currently in use will remain referring to the same file. Example: If you have a file a:test and a directory a:vim the commands ":e test" ":cd vim" ":w" will overwrite the file a:test and not write a:vim/test. But if you do ":w test" the file a:vim/test will be written, because you gave a new file name and did not refer to a filename before the ":cd".

==
8. Editing binary files *edit-binary*

Although Vim was made to edit text files, it is possible to edit binary files. The |-b| Vim argument (b for binary) makes Vim do file I/O in binary mode, and sets some options for editing binary files ('binary' on, 'textwidth' to 0, 'modeline' off, 'expandtab' off). Setting the 'binary' option has the same effect. Don't forget to do this before reading the file.

There are a few things to remember when editing binary files:
- When editing executable files the number of characters must not change. Use only the "R" or "r" command to change text. Do not delete characters with "x" or by backspacing.
- Set the 'textwidth' option to 0. Otherwise lines will unexpectedly be split in two.
- When there are not many <EOL>s, the lines will become very long. If you want to edit a line that does not fit on the screen reset the 'wrap' option. Horizontal scrolling is used then. If a line becomes too long (more than about 32767 characters on the Amiga, much more on 32-bit systems, see |limits|) you cannot edit that line. The line will be split when reading the file. It is also possible that you get an "out of memory" error when reading the file.
- Make sure the 'binary' option is set BEFORE loading the file. Otherwise both <CR> <NL> and <NL> are considered to end a line and when the file is written the <NL> will be replaced with <CR> <NL>.
- <Nul> characters are shown on the screen as ^@. You can enter them with "CTRL-V CTRL-@" or "CTRL-V 000" {Vi cannot handle <Nul> characters in the file}
- To insert a <NL> character in the file split a line. When writing the buffer to a file a <NL> will be written for the <EOL>.
- Vim normally appends an <EOL> at the end of the file if there is none. Setting the 'binary' option prevents this. If you want to add the final <EOL>, set the 'endofline' option. You can also read the value of this option to see if there was an <EOL> for the last line (you cannot see this in the text).

==
9. Encryption *encryption*

Vim is able to write files encrypted, and read them back. The encrypted text

```
cannot be read without the right key.
{only available when compiled with the |+cryptv| feature}  *E833*

The text in the swap file and the undo file is also encrypted.  *E843*
However, this is done block-by-block and may reduce the time needed to crack a
password.  You can disable the swap file, but then a crash will cause you to
lose your work.  The undo file can be disabled without much disadvantage. >
        :set noundofile
        :noswapfile edit secrets

Note: The text in memory is not encrypted.  A system administrator may be able
to see your text while you are editing it.  When filtering text with
":!filter" or using ":w !command" the text is also not encrypted, this may
reveal it to others.  The 'viminfo' file is not encrypted.

You could do this to edit very secret text: >
        :set noundofile viminfo=
        :noswapfile edit secrets.txt
Keep in mind that without a swap file you risk losing your work in the event
of a crash or a power failure.

WARNING: If you make a typo when entering the key and then write the file and
exit, the text will be lost!

The normal way to work with encryption, is to use the ":X" command, which will
ask you to enter a key.  A following write command will use that key to
encrypt the file.  If you later edit the same file, Vim will ask you to enter
a key.  If you type the same key as that was used for writing, the text will
be readable again.  If you use a wrong key, it will be a mess.

                                                        *:X*
:X      Prompt for an encryption key.  The typing is done without showing the
        actual text, so that someone looking at the display won't see it.
        The typed key is stored in the 'key' option, which is used to encrypt
        the file when it is written.  The file will remain unchanged until you
        write it.  See also |-x|.

The value of the 'key' options is used when text is written.  When the option
is not empty, the written file will be encrypted, using the value as the
encryption key.  A magic number is prepended, so that Vim can recognize that
the file is encrypted.

To disable the encryption, reset the 'key' option to an empty value: >
        :set key=

You can use the 'cryptmethod' option to select the type of encryption, use one
of these: >
        :setlocal cm=zip        " weak method, backwards compatible
        :setlocal cm=blowfish   " method with flaws
        :setlocal cm=blowfish2  " medium strong method

Do this before writing the file.  When reading an encrypted file it will be
set automatically to the method used when that file was written.  You can
change 'cryptmethod' before writing that file to change the method.

To set the default method, used for new files, use this in your |vimrc|
file: >
        set cm=blowfish2
Using "blowfish2" is highly recommended.  Only use another method if you
must use an older Vim version that does not support it.

The message given for reading and writing a file will show "[crypted]" when
```

```
using zip, "[blowfish]" when using blowfish, etc.

When writing an undo file, the same key and method will be used for the text
in the undo file. |persistent-undo|.

To test for blowfish support you can use these conditions: >
        has('crypt-blowfish')
        has('crypt-blowfish2')
This works since Vim 7.4.1099 while blowfish support was added earlier.
Thus the condition failing doesn't mean blowfish is not supported. You can
test for blowfish with: >
        v:version >= 703
And for blowfish2 with: >
        v:version > 704 || (v:version == 704 && has('patch401'))
If you are sure Vim includes patch 7.4.237 a simpler check is: >
        has('patch-7.4.401')
<
                                                    *E817* *E818* *E819* *E820*
When encryption does not work properly, you would be able to write your text
to a file and never be able to read it back.  Therefore a test is performed to
check if the encryption works as expected.  If you get one of these errors
don't write the file encrypted!  You need to rebuild the Vim binary to fix
this.

*E831* This is an internal error, "cannot happen".  If you can reproduce it,
please report to the developers.

When reading a file that has been encrypted and the 'key' option is not empty,
it will be used for decryption.  If the value is empty, you will be prompted
to enter the key.  If you don't enter a key, or you enter the wrong key, the
file is edited without being decrypted.  There is no warning about using the
wrong key (this makes brute force methods to find the key more difficult).

If want to start reading a file that uses a different key, set the 'key'
option to an empty string, so that Vim will prompt for a new one.  Don't use
the ":set" command to enter the value, other people can read the command over
your shoulder.

Since the value of the 'key' option is supposed to be a secret, its value can
never be viewed.  You should not set this option in a vimrc file.

An encrypted file can be recognized by the "file" command, if you add these
lines to "/etc/magic", "/usr/share/misc/magic" or wherever your system has the
"magic" file: >
     0  string  VimCrypt~       Vim encrypted file
     >9 string  01      - "zip" cryptmethod
     >9 string  02      - "blowfish" cryptmethod
     >9 string  03      - "blowfish2" cryptmethod

Notes:
- Encryption is not possible when doing conversion with 'charconvert'.
- Text you copy or delete goes to the numbered registers.  The registers can
  be saved in the .viminfo file, where they could be read.  Change your
  'viminfo' option to be safe.
- Someone can type commands in Vim when you walk away for a moment, he should
  not be able to get the key.
- If you make a typing mistake when entering the key, you might not be able to
  get your text back!
- If you type the key with a ":set key=value" command, it can be kept in the
  history, showing the 'key' value in a viminfo file.
- There is never 100% safety.  The encryption in Vim has not been tested for
  robustness.
```

- The algorithm used for 'cryptmethod' "zip" is breakable. A 4 character key in about one hour, a 6 character key in one day (on a Pentium 133 PC). This requires that you know some text that must appear in the file. An expert can break it for any key. When the text has been decrypted, this also means that the key can be revealed, and other files encrypted with the same key can be decrypted.
- Pkzip uses the same encryption as 'cryptmethod' "zip", and US Govt has no objection to its export. Pkzip's public file APPNOTE.TXT describes this algorithm in detail.
- The implementation of 'cryptmethod' "blowfish" has a flaw. It is possible to crack the first 64 bytes of a file and in some circumstances more of the file. Use of it is not recommended, but it's still the strongest method supported by Vim 7.3 and 7.4. The "zip" method is even weaker.
- Vim originates from the Netherlands. That is where the sources come from. Thus the encryption code is not exported from the USA.

==
10. Timestamps *timestamp* *timestamps*

Vim remembers the modification timestamp, mode and size of a file when you begin editing it. This is used to avoid that you have two different versions of the same file (without you knowing this).

After a shell command is run (|:!cmd| |suspend| |:read!| |K|) timestamps, file modes and file sizes are compared for all buffers in a window. Vim will run any associated |FileChangedShell| autocommands or display a warning for any files that have changed. In the GUI this happens when Vim regains input focus.

E321 *E462*
If you want to automatically reload a file when it has been changed outside of Vim, set the 'autoread' option. This doesn't work at the moment you write the file though, only when the file wasn't changed inside of Vim.

If you do not want to be asked or automatically reload the file, you can use this: >
```
set buftype=nofile
```

Or, when starting gvim from a shell: >
```
gvim file.log -c "set buftype=nofile"
```

Note that if a FileChangedShell autocommand is defined you will not get a warning message or prompt. The autocommand is expected to handle this.

There is no warning for a directory (e.g., with |netrw-browse|). But you do get warned if you started editing a new file and it was created as a directory later.

When Vim notices the timestamp of a file has changed, and the file is being edited in a buffer but has not changed, Vim checks if the contents of the file is equal. This is done by reading the file again (into a hidden buffer, which is immediately deleted again) and comparing the text. If the text is equal, you will get no warning.

If you don't get warned often enough you can use the following command.

:checkt *:checktime*
:checkt[ime] Check if any buffers were changed outside of Vim.
Check and warn you if you would end up with two versions of a file.
If this is called from an autocommand, a ":global" command or is not typed the actual check is postponed

until a moment the side effects (reloading the file)
would be harmless.
Each loaded buffer is checked for its associated file
being changed. If the file was changed Vim will take
action. If there are no changes in the buffer and
'autoread' is set, the buffer is reloaded. Otherwise,
you are offered the choice of reloading the file. If
the file was deleted you get an error message.
If the file previously didn't exist you get a warning
if it exists now.
Once a file has been checked the timestamp is reset,
you will not be warned again.

:[N]checkt[ime] {filename}
:[N]checkt[ime] [N]
Check the timestamp of a specific buffer. The buffer
may be specified by name, number or with a pattern.

E813 *E814*
Vim will reload the buffer if you chose to. If a window is visible that
contains this buffer, the reloading will happen in the context of this window.
Otherwise a special window is used, so that most autocommands will work. You
can't close this window. A few other restrictions apply. Best is to make
sure nothing happens outside of the current buffer. E.g., setting
window-local options may end up in the wrong window. Splitting the window,
doing something there and closing it should be OK (if there are no side
effects from other autocommands). Closing unrelated windows and buffers will
get you into trouble.

Before writing a file the timestamp is checked. If it has changed, Vim will
ask if you really want to overwrite the file:

WARNING: The file has been changed since reading it!!!
Do you really want to write to it (y/n)?

If you hit 'y' Vim will continue writing the file. If you hit 'n' the write is
aborted. If you used ":wq" or "ZZ" Vim will not exit, you will get another
chance to write the file.

The message would normally mean that somebody has written to the file after
the edit session started. This could be another person, in which case you
probably want to check if your changes to the file and the changes from the
other person should be merged. Write the file under another name and check for
differences (the "diff" program can be used for this).

It is also possible that you modified the file yourself, from another edit
session or with another command (e.g., a filter command). Then you will know
which version of the file you want to keep.

There is one situation where you get the message while there is nothing wrong:
On a Win32 system on the day daylight saving time starts. There is something
in the Win32 libraries that confuses Vim about the hour time difference. The
problem goes away the next day.

==
11. File Searching *file-searching*

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

The file searching is currently used for the 'path', 'cdpath' and 'tags'
options, for |finddir()| and |findfile()|. Other commands use |wildcards|

```
which is slightly different.

There are three different types of searching:

1) Downward search:                                        *starstar*
   Downward search uses the wildcards '*', '**' and possibly others
   supported by your operating system.  '*' and '**' are handled inside Vim,
   so they work on all operating systems.  Note that "**" only acts as a
   special wildcard when it is at the start of a name.

   The usage of '*' is quite simple: It matches 0 or more characters.  In a
   search pattern this would be ".*".  Note that the "." is not used for file
   searching.

   '**' is more sophisticated:
      - It ONLY matches directories.
      - It matches up to 30 directories deep by default, so you can use it to
        search an entire directory tree
      - The maximum number of levels matched can be given by appending a number
        to '**'.
        Thus '/usr/**2' can match: >
                /usr
                /usr/include
                /usr/include/sys
                /usr/include/g++
                /usr/lib
                /usr/lib/X11
                ....
<       It does NOT match '/usr/include/g++/std' as this would be three
        levels.
        The allowed number range is 0 ('**0' is removed) to 100
        If the given number is smaller than 0 it defaults to 30, if it's
        bigger than 100 then 100 is used.  The system also has a limit on the
        path length, usually 256 or 1024 bytes.
      - '**' can only be at the end of the path or be followed by a path
        separator or by a number and a path separator.

   You can combine '*' and '**' in any order: >
        /usr/**/sys/*
        /usr/*tory/sys/**
        /usr/**2/sys/*

2) Upward search:
   Here you can give a directory and then search the directory tree upward for
   a file.  You could give stop-directories to limit the upward search.  The
   stop-directories are appended to the path (for the 'path' option) or to
   the filename (for the 'tags' option) with a ';'.  If you want several
   stop-directories separate them with ';'.  If you want no stop-directory
   ("search upward till the root directory) just use ';'. >
        /usr/include/sys;/usr
<   will search in: >
           /usr/include/sys
           /usr/include
           /usr
<
   If you use a relative path the upward search is started in Vim's current
   directory or in the directory of the current file (if the relative path
   starts with './' and 'd' is not included in 'cpoptions').

   If Vim's current path is /u/user_x/work/release and you do >
        :set path=include;/u/user_x
<  and then search for a file with |gf| the file is searched in: >
```

```
        /u/user_x/work/release/include
        /u/user_x/work/include
        /u/user_x/include

3) Combined up/downward search:
   If Vim's current path is /u/user_x/work/release and you do >
        set path=**;/u/user_x
<  and then search for a file with |gf| the file is searched in: >
        /u/user_x/work/release/**
        /u/user_x/work/**
        /u/user_x/**
<
   BE CAREFUL!  This might consume a lot of time, as the search of
   '/u/user_x/**' includes '/u/user_x/work/**' and
   '/u/user_x/work/release/**'.  So '/u/user_x/work/release/**' is searched
   three times and '/u/user_x/work/**' is searched twice.

   In the above example you might want to set path to: >
        :set path=**,/u/user_x/**
<  This searches:
        /u/user_x/work/release/** ~
        /u/user_x/** ~
   This searches the same directories, but in a different order.

   Note that completion for ":find", ":sfind", and ":tabfind" commands do not
   currently work with 'path' items that contain a URL or use the double star
   with depth limiter (/usr/**2) or upward search (;) notations.

 vim:tw=78:ts=8:ft=help:norl:
*motion.txt*    For Vim version 8.0.  Last change: 2017 Mar 12


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Cursor motions                                    *cursor-motions* *navigation*

These commands move the cursor position.  If the new position is off of the
screen, the screen is scrolled to show the cursor (see also 'scrolljump' and
'scrolloff' options).

1. Motions and operators        |operator|
2. Left-right motions           |left-right-motions|
3. Up-down motions              |up-down-motions|
4. Word motions                 |word-motions|
5. Text object motions          |object-motions|
6. Text object selection        |object-select|
7. Marks                        |mark-motions|
8. Jumps                        |jump-motions|
9. Various motions              |various-motions|

General remarks:

If you want to know where you are in the file use the "CTRL-G" command
|CTRL-G| or the "g CTRL-G" command |g_CTRL-G|.  If you set the 'ruler' option,
the cursor position is continuously shown in the status line (which slows down
Vim a little).

Experienced users prefer the hjkl keys because they are always right under
their fingers.  Beginners often prefer the arrow keys, because they do not
know what the hjkl keys do.  The mnemonic value of hjkl is clear from looking
at the keyboard.  Think of j as an arrow pointing downwards.
```

The 'virtualedit' option can be set to make it possible to move the cursor to
positions where there is no character or halfway a character.

==
1. Motions and operators *operator*

The motion commands can be used after an operator command, to have the command
operate on the text that was moved over. That is the text between the cursor
position before and after the motion. Operators are generally used to delete
or change text. The following operators are available:

c|	c	change
d|	d	delete
y|	y	yank into register (does not change the text)
~|	~	swap case (only if 'tildeop' is set)
g~|	g~	swap case
gu|	gu	make lowercase
gU|	gU	make uppercase
!|	!	filter through an external program
=|	=	filter through 'equalprg' or C-indenting if empty
gq|	gq	text formatting
g?|	g?	ROT13 encoding
>|	>	shift right
<|	<	shift left
zf|	zf	define a fold
g@|	g@	call function set with the 'operatorfunc' option

If the motion includes a count and the operator also had a count before it,
the two counts are multiplied. For example: "2d3w" deletes six words.

After applying the operator the cursor is mostly left at the start of the text
that was operated upon. For example, "yfe" doesn't move the cursor, but "yFe"
moves the cursor leftwards to the "e" where the yank started.

linewise *characterwise*
The operator either affects whole lines, or the characters between the start
and end position. Generally, motions that move between lines affect lines
(are linewise), and motions that move within a line affect characters (are
characterwise). However, there are some exceptions.

exclusive *inclusive*
A character motion is either inclusive or exclusive. When inclusive, the
start and end position of the motion are included in the operation. When
exclusive, the last character towards the end of the buffer is not included.
Linewise motions always include the start and end position.

Which motions are linewise, inclusive or exclusive is mentioned with the
command. There are however, two general exceptions:
1. If the motion is exclusive and the end of the motion is in column 1, the
   end of the motion is moved to the end of the previous line and the motion
   becomes inclusive. Example: "}" moves to the first line after a paragraph,
   but "d}" will not include that line.
   exclusive-linewise
2. If the motion is exclusive, the end of the motion is in column 1 and the
   start of the motion was at or before the first non-blank in the line, the
   motion becomes linewise. Example: If a paragraph begins with some blanks
   and you do "d}" while standing on the first non-blank, all the lines of
   the paragraph are deleted, including the blanks. If you do a put now, the
   deleted lines will be inserted below the cursor position.

Note that when the operator is pending (the operator command is typed, but the

```
motion isn't yet), a special set of mappings can be used.  See |:omap|.

Instead of first giving the operator and then a motion you can use Visual
mode: mark the start of the text with "v", move the cursor to the end of the
text that is to be affected and then hit the operator.  The text between the
start and the cursor position is highlighted, so you can see what text will
be operated upon.  This allows much more freedom, but requires more key
strokes and has limited redo functionality.  See the chapter on Visual mode
|Visual-mode|.

You can use a ":" command for a motion.  For example "d:call FindEnd()".
But this can't be repeated with "." if the command is more than one line.
This can be repeated: >
        d:call search("f")<CR>
This cannot be repeated: >
        d:if 1<CR>
           call search("f")<CR>
        endif<CR>
Note that when using ":" any motion becomes characterwise exclusive.


FORCING A MOTION TO BE LINEWISE, CHARACTERWISE OR BLOCKWISE

When a motion is not of the type you would like to use, you can force another
type by using "v", "V" or CTRL-V just after the operator.
Example: >
        dj
deletes two lines >
        dvj
deletes from the cursor position until the character below the cursor >
        d<C-V>j
deletes the character under the cursor and the character below the cursor. >

Be careful with forcing a linewise movement to be used characterwise or
blockwise, the column may not always be defined.

                                                        *o_v*
v               When used after an operator, before the motion command: Force
                the operator to work characterwise, also when the motion is
                linewise.  If the motion was linewise, it will become
                |exclusive|.
                If the motion already was characterwise, toggle
                inclusive/exclusive.  This can be used to make an exclusive
                motion inclusive and an inclusive motion exclusive.

                                                        *o_V*
V               When used after an operator, before the motion command: Force
                the operator to work linewise, also when the motion is
                characterwise.

                                                        *o_CTRL-V*
CTRL-V          When used after an operator, before the motion command: Force
                the operator to work blockwise.  This works like Visual block
                mode selection, with the corners defined by the cursor
                position before and after the motion.

==============================================================================
2. Left-right motions                                   *left-right-motions*

These commands move the cursor to the specified column in the current line.
They stop at the first column and at the end of the line, except "$", which
may move to one of the next lines.  See 'whichwrap' option to make some of the
```

commands move across line boundaries.

h or *h*
<Left> or *<Left>*
CTRL-H or *CTRL-H* *<BS>*
<BS> [count] characters to the left. |exclusive| motion.
Note: If you prefer <BS> to delete a character, use
the mapping:
:map CTRL-V<BS> X
(to enter "CTRL-V<BS>" type the CTRL-V key, followed
by the <BS> key)
See |:fixdel| if the <BS> key does not do what you
want.

l or *l*
<Right> or *<Right>* *<Space>*
<Space> [count] characters to the right. |exclusive| motion.
See the 'whichwrap' option for adjusting the behavior
at end of line

0
0 To the first character of the line. |exclusive|
motion.

<Home> *<kHome>*
<Home> To the first character of the line. |exclusive|
motion. When moving up or down next, stay in same
TEXT column (if possible). Most other commands stay
in the same SCREEN column. <Home> works like "1|",
which differs from "0" when the line starts with a
<Tab>. {not in Vi}

^
^ To the first non-blank character of the line.
|exclusive| motion.

$ *<End>* *<kEnd>*
$ or <End> To the end of the line. When a count is given also go
[count - 1] lines downward. |inclusive| motion.
In Visual mode the cursor goes to just after the last
character in the line.
When 'virtualedit' is active, "$" may move the cursor
back from past the end of the line to the last
character in the line.

*g_*
g_ To the last non-blank character of the line and
[count - 1] lines downward |inclusive|. {not in Vi}

g0 *g<Home>*
g0 or g<Home> When lines wrap ('wrap' on): To the first character of
the screen line. |exclusive| motion. Differs from
"0" when a line is wider than the screen.
When lines don't wrap ('wrap' off): To the leftmost
character of the current line that is on the screen.
Differs from "0" when the first character of the line
is not on the screen. {not in Vi}

*g^*
g^ When lines wrap ('wrap' on): To the first non-blank
character of the screen line. |exclusive| motion.
Differs from "^" when a line is wider than the screen.

When lines don't wrap ('wrap' off): To the leftmost non-blank character of the current line that is on the screen. Differs from "^" when the first non-blank character of the line is not on the screen. {not in Vi}

gm

gm — Like "g0", but half a screenwidth to the right (or as much as possible). {not in Vi}

g$ *g<End>*

g$ or g<End> — When lines wrap ('wrap' on): To the last character of the screen line and [count - 1] screen lines downward |inclusive|. Differs from "$" when a line is wider than the screen.
When lines don't wrap ('wrap' off): To the rightmost character of the current line that is visible on the screen. Differs from "$" when the last character of the line is not on the screen or when a count is used.
Additionally, vertical movements keep the column, instead of going to the end of the line.
When 'virtualedit' is enabled moves to the end of the screen line.
{not in Vi}

bar

| — To screen column [count] in the current line. |exclusive| motion. Ceci n'est pas une pipe.

f

f{char} — To [count]'th occurrence of {char} to the right. The cursor is placed on {char} |inclusive|.
{char} can be entered as a digraph |digraph-arg|.
When 'encoding' is set to Unicode, composing characters may be used, see |utf-8-char-arg|.
|:lmap| mappings apply to {char}. The CTRL-^ command in Insert mode can be used to switch this on/off |i_CTRL-^|.

F

F{char} — To the [count]'th occurrence of {char} to the left. The cursor is placed on {char} |exclusive|.
{char} can be entered like with the |f| command.

t

t{char} — Till before [count]'th occurrence of {char} to the right. The cursor is placed on the character left of {char} |inclusive|.
{char} can be entered like with the |f| command.

T

T{char} — Till after [count]'th occurrence of {char} to the left. The cursor is placed on the character right of {char} |exclusive|.
{char} can be entered like with the |f| command.

;

; — Repeat latest f, t, F or T [count] times. See |cpo-;|

,

, — Repeat latest f, t, F or T in opposite direction [count] times. See also |cpo-;|

```
==============================================================================
3. Up-down motions                                      *up-down-motions*

k               or                                      *k*
<Up>            or                                      *<Up>* *CTRL-P*
CTRL-P                  [count] lines upward |linewise|.

j               or                                      *j*
<Down>          or                                      *<Down>*
CTRL-J          or                                      *CTRL-J*
<NL>            or                                      *<NL>* *CTRL-N*
CTRL-N                  [count] lines downward |linewise|.

gk              or                                      *gk* *g<Up>*
g<Up>                   [count] display lines upward.  |exclusive| motion.
                        Differs from 'k' when lines wrap, and when used with
                        an operator, because it's not linewise.  {not in Vi}

gj              or                                      *gj* *g<Down>*
g<Down>                 [count] display lines downward.  |exclusive| motion.
                        Differs from 'j' when lines wrap, and when used with
                        an operator, because it's not linewise.  {not in Vi}

                                                        *-*
-  <minus>              [count] lines upward, on the first non-blank
                        character |linewise|.

+               or                                      *+*
CTRL-M          or                                      *CTRL-M* *<CR>*
<CR>                    [count] lines downward, on the first non-blank
                        character |linewise|.

                                                        *_*
_  <underscore>         [count] - 1 lines downward, on the first non-blank
                        character |linewise|.

                                                        *G*
G                       Goto line [count], default last line, on the first
                        non-blank character |linewise|.  If 'startofline' not
                        set, keep the same column.
                        G is a one of |jump-motions|.

                                                        *<C-End>*
<C-End>                 Goto line [count], default last line, on the last
                        character |inclusive|. {not in Vi}

<C-Home>        or                                      *gg* *<C-Home>*
gg                      Goto line [count], default first line, on the first
                        non-blank character |linewise|.  If 'startofline' not
                        set, keep the same column.

                                                        *:[range]*
:[range]                Set the cursor on the last line number in [range].
                        [range] can also be just one line number, e.g., ":1"
                        or ":'m".
                        In contrast with |G| this command does not modify the
                        |jumplist|.
                                                        *N%*
{count}%                Go to {count} percentage in the file, on the first
                        non-blank in the line |linewise|.  To compute the new
                        line number this formula is used:
```

```
                            ({count} * number-of-lines + 99) / 100
                        See also 'startofline' option.  {not in Vi}

:[range]go[to] [count]                                  *:go* *:goto* *go*
[count]go               Go to [count] byte in the buffer.  Default [count] is
                        one, start of the file.  When giving [range], the
                        last number in it used as the byte count.  End-of-line
                        characters are counted depending on the current
                        'fileformat' setting.
                        Also see the |line2byte()| function, and the 'o'
                        option in 'statusline'.
                        {not in Vi}
                        {not available when compiled without the
                        |+byte_offset| feature}

These commands move to the specified line.  They stop when reaching the first
or the last line.  The first two commands put the cursor in the same column
(if possible) as it was after the last command that changed the column,
except after the "$" command, then the cursor will be put on the last
character of the line.

If "k", "-" or CTRL-P is used with a [count] and there are less than [count]
lines above the cursor and the 'cpo' option includes the "-" flag it is an
error. |cpo--|.

==============================================================================
4. Word motions                                         *word-motions*

<S-Right>       or                                      *<S-Right>* *w*
w                       [count] words forward.  |exclusive| motion.

<C-Right>       or                                      *<C-Right>* *W*
W                       [count] WORDS forward.  |exclusive| motion.

                                                        *e*
e                       Forward to the end of word [count] |inclusive|.
                        Does not stop in an empty line.

                                                        *E*
E                       Forward to the end of WORD [count] |inclusive|.
                        Does not stop in an empty line.

<S-Left>        or                                      *<S-Left>* *b*
b                       [count] words backward.  |exclusive| motion.

<C-Left>        or                                      *<C-Left>* *B*
B                       [count] WORDS backward.  |exclusive| motion.

                                                        *ge*
ge                      Backward to the end of word [count] |inclusive|.

                                                        *gE*
gE                      Backward to the end of WORD [count] |inclusive|.

These commands move over words or WORDS.
                                                        *word*
A word consists of a sequence of letters, digits and underscores, or a
sequence of other non-blank characters, separated with white space (spaces,
tabs, <EOL>).  This can be changed with the 'iskeyword' option.  An empty line
is also considered to be a word.
                                                        *WORD*
A WORD consists of a sequence of non-blank characters, separated with white
```

space. An empty line is also considered to be a WORD.

A sequence of folded lines is counted for one word of a single character.
"w" and "W", "e" and "E" move to the start/end of the first word or WORD after
a range of folded lines. "b" and "B" move to the start of the first word or
WORD before the fold.

Special case: "cw" and "cW" are treated like "ce" and "cE" if the cursor is
on a non-blank. This is because "cw" is interpreted as change-word, and a
word does not include the following white space. {Vi: "cw" when on a blank
followed by other blanks changes only the first blank; this is probably a
bug, because "dw" deletes all the blanks}

Another special case: When using the "w" motion in combination with an
operator and the last word moved over is at the end of a line, the end of
that word becomes the end of the operated text, not the first word in the
next line.

The original Vi implementation of "e" is buggy. For example, the "e" command
will stop on the first character of a line if the previous line was empty.
But when you use "2e" this does not happen. In Vim "ee" and "2e" are the
same, which is more logical. However, this causes a small incompatibility
between Vi and Vim.

==
5. Text object motions   *object-motions*

                                                        (
([count] sentences backward. |exclusive| motion.

)
)   [count] sentences forward. |exclusive| motion.

                                                        {
{   [count] paragraphs backward. |exclusive| motion.

                                                        }
}   [count] paragraphs forward. |exclusive| motion.

                                                        ]]
]]   [count] sections forward or to the next '{' in the
                        first column. When used after an operator, then also
                        stops below a '}' in the first column. |exclusive|
                        Note that |exclusive-linewise| often applies.

                                                        ][
][[count] sections forward or to the next '}' in the
                        first column. |exclusive|
                        Note that |exclusive-linewise| often applies.

                                                        [[
[[[count] sections backward or to the previous '{' in
                        the first column. |exclusive|
                        Note that |exclusive-linewise| often applies.

                                                        []
[]   [count] sections backward or to the previous '}' in
                        the first column. |exclusive|
                        Note that |exclusive-linewise| often applies.

These commands move over three kinds of text objects.

sentence
A sentence is defined as ending at a '.', '!' or '?' followed by either the end of a line, or by a space or tab. Any number of closing ')', ']', '"' and ''' characters may appear after the '.', '!' or '?' before the spaces, tabs or end of line. A paragraph and section boundary is also a sentence boundary.
If the 'J' flag is present in 'cpoptions', at least two spaces have to follow the punctuation mark; <Tab>s are not recognized as white space.
The definition of a sentence cannot be changed.

paragraph
A paragraph begins after each empty line, and also at each of a set of paragraph macros, specified by the pairs of characters in the 'paragraphs' option. The default is "IPLPPPQPP TPHPLIPpLpItpplpipbp", which corresponds to the macros ".IP", ".LP", etc. (These are nroff macros, so the dot must be in the first column). A section boundary is also a paragraph boundary.
Note that a blank line (only containing white space) is NOT a paragraph boundary.
Also note that this does not include a '{' or '}' in the first column. When the '{' flag is in 'cpoptions' then '{' in the first column is used as a paragraph boundary |posix|.

section
A section begins after a form-feed (<C-L>) in the first column and at each of a set of section macros, specified by the pairs of characters in the 'sections' option. The default is "SHNHH HUnhsh", which defines a section to start at the nroff macros ".SH", ".NH", ".H", ".HU", ".nh" and ".sh".

The "]" and "[" commands stop at the '{' or '}' in the first column. This is useful to find the start or end of a function in a C program. Note that the first character of the command determines the search direction and the second character the type of brace found.

If your '{' or '}' are not in the first column, and you would like to use "[[" and "]]" anyway, try these mappings: >
```
   :map [[ ?{<CR>w99[{
   :map ][ /}<CR>b99]}
   :map ]] j0[[%/{<CR>
   :map [] k$][%?}<CR>
```
[type these literally, see |<>|]

==
6. Text object selection *object-select* *text-objects* *v_a* *v_i*

This is a series of commands that can only be used while in Visual mode or after an operator. The commands that start with "a" select "a"n object including white space, the commands starting with "i" select an "inner" object without white space, or just the white space. Thus the "inner" commands always select less text than the "a" commands.

These commands are {not in Vi}.
These commands are not available when the |+textobjects| feature has been disabled at compile time.
Also see `gn` and `gN`, operating on the last search pattern.

v_aw *aw*
aw "a word", select [count] words (see |word|). Leading or trailing white space is included, but not counted.
When used in Visual linewise mode "aw" switches to Visual characterwise mode.

v_iw *iw*
iw "inner word", select [count] words (see |word|).
White space between words is counted too.
When used in Visual linewise mode "iw" switches to
Visual characterwise mode.

v_aW *aW*
aW "a WORD", select [count] WORDs (see |WORD|).
Leading or trailing white space is included, but not
counted.
When used in Visual linewise mode "aW" switches to
Visual characterwise mode.

v_iW *iW*
iW "inner WORD", select [count] WORDs (see |WORD|).
White space between words is counted too.
When used in Visual linewise mode "iW" switches to
Visual characterwise mode.

v_as *as*
as "a sentence", select [count] sentences (see
|sentence|).
When used in Visual mode it is made characterwise.

v_is *is*
is "inner sentence", select [count] sentences (see
|sentence|).
When used in Visual mode it is made characterwise.

v_ap *ap*
ap "a paragraph", select [count] paragraphs (see
|paragraph|).
Exception: a blank line (only containing white space)
is also a paragraph boundary.
When used in Visual mode it is made linewise.

v_ip *ip*
ip "inner paragraph", select [count] paragraphs (see
|paragraph|).
Exception: a blank line (only containing white space)
is also a paragraph boundary.
When used in Visual mode it is made linewise.

v_a] *v_a[* *a]* *a[*
a] a["a [] block", select [count] '[' ']' blocks. This
goes backwards to the [count] unclosed '[', and finds
the matching ']'. The enclosed text is selected,
including the '[' and ']'.
When used in Visual mode it is made characterwise.

v_i] *v_i[* *i]* *i[*
i] i["inner [] block", select [count] '[' ']' blocks. This
goes backwards to the [count] unclosed '[', and finds
the matching ']'. The enclosed text is selected,
excluding the '[' and ']'.
When used in Visual mode it is made characterwise.

v_a) *a)* *a(*
vab *v_ab* *v_a(* *ab*
a) a(ab "a block", select [count] blocks, from "[count] [(" to
the matching ')', including the '(' and ')' (see

|[(|). Does not include white space outside of the
parenthesis.
When used in Visual mode it is made characterwise.

i) *v_i)* *i)* *i(*
i(*vib* *v_ib* *v_i(* *ib*
ib "inner block", select [count] blocks, from "[count] [(" to the matching ')', excluding the '(' and ')' (see |[(|).
When used in Visual mode it is made characterwise.

a> *v_a>* *v_a<* *a>* *a<*
a< "a <> block", select [count] <> blocks, from the [count]'th unmatched '<' backwards to the matching '>', including the '<' and '>'.
When used in Visual mode it is made characterwise.

i> *v_i>* *v_i<* *i>* *i<*
i< "inner <> block", select [count] <> blocks, from the [count]'th unmatched '<' backwards to the matching '>', excluding the '<' and '>'.
When used in Visual mode it is made characterwise.

v_at *at*
at "a tag block", select [count] tag blocks, from the [count]'th unmatched "<aaa>" backwards to the matching "</aaa>", including the "<aaa>" and "</aaa>".
See |tag-blocks| about the details.
When used in Visual mode it is made characterwise.

v_it *it*
it "inner tag block", select [count] tag blocks, from the [count]'th unmatched "<aaa>" backwards to the matching "</aaa>", excluding the "<aaa>" and "</aaa>".
See |tag-blocks| about the details.
When used in Visual mode it is made characterwise.

a} *v_a}* *a}* *a{*
a{ *v_aB* *v_a{* *aB*
aB "a Block", select [count] Blocks, from "[count] [{" to the matching '}', including the '{' and '}' (see |[{|).
When used in Visual mode it is made characterwise.

i} *v_i}* *i}* *i{*
i{ *v_iB* *v_i{* *iB*
iB "inner Block", select [count] Blocks, from "[count] [{" to the matching '}', excluding the '{' and '}' (see |[{|).
When used in Visual mode it is made characterwise.

a" *v_aquote* *aquote*
a' *v_a'* *a'*
a`  *v_a`* *a`*
"a quoted string". Selects the text from the previous quote until the next quote. The 'quoteescape' option is used to skip escaped quotes.
Only works within one line.
When the cursor starts on a quote, Vim will figure out which quote pairs form a string by searching from the start of the line.
Any trailing white space is included, unless there is

none, then leading white space is included.
When used in Visual mode it is made characterwise.
Repeating this object in Visual mode another string is
included. A count is currently not used.

i" *v_iquote* *iquote*
i' *v_i'* *i'*
i` *v_i`* *i`*
Like a", a' and a`, but exclude the quotes and
repeating won't extend the Visual selection.
Special case: With a count of 2 the quotes are
included, but no extra white space as with a"/a'/a`.

When used after an operator:
For non-block objects:
For the "a" commands: The operator applies to the object and the white space after the object. If there is no white space after the object or when the cursor was in the white space before the object, the white space before the object is included.
For the "inner" commands: If the cursor was on the object, the operator applies to the object. If the cursor was on white space, the operator applies to the white space.
For a block object:
The operator applies to the block where the cursor is in, or the block on which the cursor is on one of the braces. For the "inner" commands the surrounding braces are excluded. For the "a" commands, the braces are included.

When used in Visual mode:
When start and end of the Visual area are the same (just after typing "v"):
One object is selected, the same as for using an operator.
When start and end of the Visual area are not the same:
For non-block objects the area is extended by one object or the white space up to the next object, or both for the "a" objects. The direction in which this happens depends on which side of the Visual area the cursor is. For the block objects the block is extended one level outwards.

For illustration, here is a list of delete commands, grouped from small to big objects. Note that for a single character and a whole line the existing vi movement commands are used.

"dl"	delete character (alias: "x")	|dl|
"diw"	delete inner word	*diw*
"daw"	delete a word	*daw*
"diW"	delete inner WORD (see |WORD|)	*diW*
"daW"	delete a WORD (see |WORD|)	*daW*
"dgn"	delete the next search pattern match	*dgn*
"dd"	delete one line	|dd|
"dis"	delete inner sentence	*dis*
"das"	delete a sentence	*das*
"dib"	delete inner '(' ')' block	*dib*
"dab"	delete a '(' ')' block	*dab*
"dip"	delete inner paragraph	*dip*
"dap"	delete a paragraph	*dap*
"diB"	delete inner '{' '}' block	*diB*
"daB"	delete a '{' '}' block	*daB*

Note the difference between using a movement command and an object. The movement command operates from here (cursor position) to where the movement takes us. When using an object the whole object is operated upon, no matter where on the object the cursor is. For example, compare "dw" and "daw": "dw" deletes from the cursor position to the start of the next word, "daw" deletes

the word under the cursor and the space after or before it.

Tag blocks *tag-blocks*

For the "it" and "at" text objects an attempt is done to select blocks between matching tags for HTML and XML. But since these are not completely compatible there are a few restrictions.

The normal method is to select a <tag> until the matching </tag>. For "at" the tags are included, for "it" they are excluded. But when "it" is repeated the tags will be included (otherwise nothing would change). Also, "it" used on a tag block with no contents will select the leading tag.

"<aaa/>" items are skipped. Case is ignored, also for XML where case does matter.

In HTML it is possible to have a tag like
 or <meta ...> without a matching end tag. These are ignored.

The text objects are tolerant about mistakes. Stray end tags are ignored.

==
7. Marks *mark-motions* *E20* *E78*

Jumping to a mark can be done in two ways:
1. With ` (backtick): The cursor is positioned at the specified location and the motion is |exclusive|.
2. With ' (single quote): The cursor is positioned on the first non-blank character in the line of the specified location and the motion is linewise.

m *mark* *Mark*
m{a-zA-Z} Set mark {a-zA-Z} at cursor position (does not move the cursor, this is not a motion command).

m' *m`*
m' or m` Set the previous context mark. This can be jumped to with the "''" or "``" command (does not move the cursor, this is not a motion command).

m[*m]*
m[or m] Set the |'[| or |']| mark. Useful when an operator is to be simulated by multiple commands. (does not move the cursor, this is not a motion command).

m< *m>*
m< or m> Set the |'<| or |'>| mark. Useful to change what the `gv` command selects. (does not move the cursor, this is not a motion command).
Note that the Visual mode cannot be set, only the start and end position.

:ma *:mark* *E191*
:[range]ma[rk] {a-zA-Z'}
Set mark {a-zA-Z'} at last line number in [range], column 0. Default is cursor line.

:k
:[range]k{a-zA-Z'} Same as :mark, but the space before the mark name can be omitted.

```
                                                *'* *'a* *`* *`a*
'{a-z}  `{a-z}          Jump to the mark {a-z} in the current buffer.

                                                *'A* *'0* *`A* *`0*
'{A-Z0-9}  `{A-Z0-9}    To the mark {A-Z0-9} in the file where it was set (not
                        a motion command when in another file).  {not in Vi}

                                                *g'* *g'a* *g`* *g`a*
g'{mark}  g`{mark}
                        Jump to the {mark}, but don't change the jumplist when
                        jumping within the current buffer.  Example: >
                                g`"
<                       jumps to the last known position in a file.  See
                        $VIMRUNTIME/vimrc_example.vim.
                        Also see |:keepjumps|.
                        {not in Vi}

                                                *:marks*
:marks                  List all the current marks (not a motion command).
                        The |'(|, |')|, |'{| and |'}| marks are not listed.
                        The first column has number zero.
                        {not in Vi}
                                                *E283*
:marks {arg}            List the marks that are mentioned in {arg} (not a
                        motion command).  For example: >
                                :marks aB
<                       to list marks 'a' and 'B'.  {not in Vi}

                                                        *:delm* *:delmarks*
:delm[arks] {marks}     Delete the specified marks.  Marks that can be deleted
                        include A-Z and 0-9.  You cannot delete the ' mark.
                        They can be specified by giving the list of mark
                        names, or with a range, separated with a dash.  Spaces
                        are ignored.  Examples: >
                           :delmarks a        deletes mark a
                           :delmarks a b 1    deletes marks a, b and 1
                           :delmarks Aa       deletes marks A and a
                           :delmarks p-z      deletes marks in the range p to z
                           :delmarks ^.[]     deletes marks ^ . [ ]
                           :delmarks \"       deletes mark "
<                       {not in Vi}

:delm[arks]!            Delete all marks for the current buffer, but not marks
                        A-Z or 0-9.
                        {not in Vi}

A mark is not visible in any way.  It is just a position in the file that is
remembered.  Do not confuse marks with named registers, they are totally
unrelated.

'a - 'z         lowercase marks, valid within one file
'A - 'Z         uppercase marks, also called file marks, valid between files
'0 - '9         numbered marks, set from .viminfo file

Lowercase marks 'a to 'z are remembered as long as the file remains in the
buffer list.  If you remove the file from the buffer list, all its marks are
lost.  If you delete a line that contains a mark, that mark is erased.

Lowercase marks can be used in combination with operators.  For example: "d't"
deletes the lines from the cursor position to mark 't'.  Hint: Use mark 't' for
Top, 'b' for Bottom, etc..  Lowercase marks are restored when using undo and
redo.
```

Uppercase marks 'A to 'Z include the file name. {Vi: no uppercase marks} You can use them to jump from file to file. You can only use an uppercase mark with an operator if the mark is in the current file. The line number of the mark remains correct, even if you insert/delete lines or edit another file for a moment. When the 'viminfo' option is not empty, uppercase marks are kept in the .viminfo file. See |viminfo-file-marks|.

Numbered marks '0 to '9 are quite different. They can not be set directly. They are only present when using a viminfo file |viminfo-file|. Basically '0 is the location of the cursor when you last exited Vim, '1 the last but one time, etc. Use the "r" flag in 'viminfo' to specify files for which no Numbered mark should be stored. See |viminfo-file-marks|.

'[*`[*
'[`[To the first character of the previously changed or yanked text. {not in Vi}

'] *`]*
'] `] To the last character of the previously changed or yanked text. {not in Vi}

After executing an operator the Cursor is put at the beginning of the text that was operated upon. After a put command ("p" or "P") the cursor is sometimes placed at the first inserted line and sometimes on the last inserted character. The four commands above put the cursor at either end. Example: After yanking 10 lines you want to go to the last one of them: "10Y']". After inserting several lines with the "p" command you want to jump to the lowest inserted line: "p']". This also works for text that has been inserted.

Note: After deleting text, the start and end positions are the same, except when using blockwise Visual mode. These commands do not work when no change was made yet in the current file.

'< *`<*
'< `< To the first line or character of the last selected Visual area in the current buffer. For block mode it may also be the last character in the first line (to be able to define the block). {not in Vi}.

'> *`>*
'> `> To the last line or character of the last selected Visual area in the current buffer. For block mode it may also be the first character of the last line (to be able to define the block). Note that 'selection' applies, the position may be just after the Visual area. {not in Vi}.

'' *``*
'' `` To the position before the latest jump, or where the last "m'" or "m`" command was given. Not set when the |:keepjumps| command modifier was used.
Also see |restore-position|.

'quote *`quote*
'" `" To the cursor position when last exiting the current buffer. Defaults to the first character of the first line. See |last-position-jump| for how to use this for each opened file.
Only one position is remembered per buffer, not one for each window. As long as the buffer is visible in

a window the position won't be changed.
{not in Vi}.

*'^* *`^*
'^ `^ To the position where the cursor was the last time
when Insert mode was stopped. This is used by the
|gi| command. Not set when the |:keepjumps| command
modifier was used. {not in Vi}

'. *`.*
'. `. To the position where the last change was made. The
position is at or near where the change started.
Sometimes a command is executed as several changes,
then the position can be near the end of what the
command changed. For example when inserting a word,
the position will be on the last character.
To jump to older changes use |g;|.
{not in Vi}

'(*`(*
'(`(To the start of the current sentence, like the |(|
command. {not in Vi}

') *`)*
') `) To the end of the current sentence, like the |)|
command. {not in Vi}

'{ *`{*
'{ `{ To the start of the current paragraph, like the |{|
command. {not in Vi}

'} *`}*
'} `} To the end of the current paragraph, like the |}|
command. {not in Vi}

These commands are not marks themselves, but jump to a mark:

]'
]' [count] times to next line with a lowercase mark below
the cursor, on the first non-blank character in the
line. {not in Vi}

]`
]` [count] times to lowercase mark after the cursor. {not
in Vi}

['
[' [count] times to previous line with a lowercase mark
before the cursor, on the first non-blank character in
the line. {not in Vi}

[`
[` [count] times to lowercase mark before the cursor.
{not in Vi}

:loc[kmarks] {command} *:loc* *:lockmarks*
Execute {command} without adjusting marks. This is
useful when changing text in a way that the line count
will be the same when the change has completed.
WARNING: When the line count does change, marks below
the change will keep their line number, thus move to

```
                        another text line.
                        These items will not be adjusted for deleted/inserted
                        lines:
                        - lower case letter marks 'a - 'z
                        - upper case letter marks 'A - 'Z
                        - numbered marks '0 - '9
                        - last insert position '^
                        - last change position '.
                        - the Visual area '< and '>
                        - line numbers in placed signs
                        - line numbers in quickfix positions
                        - positions in the |jumplist|
                        - positions in the |tagstack|
                        These items will still be adjusted:
                        - previous context mark ''
                        - the cursor position
                        - the view of a window on a buffer
                        - folds
                        - diffs

:kee[pmarks] {command}                                  *:kee* *:keepmarks*
                        Currently only has effect for the filter command
                        |:range!|:
                        - When the number of lines after filtering is equal to
                          or larger than before, all marks are kept at the
                          same line number.
                        - When the number of lines decreases, the marks in the
                          lines that disappeared are deleted.
                        In any case the marks below the filtered text have
                        their line numbers adjusted, thus stick to the text,
                        as usual.
                        When the 'R' flag is missing from 'cpoptions' this has
                        the same effect as using ":keepmarks".

                                                        *:keepj* *:keepjumps*
:keepj[umps] {command}
                        Moving around in {command} does not change the |''|,
                        |'.| and |'^| marks, the |jumplist| or the
                        |changelist|.
                        Useful when making a change or inserting text
                        automatically and the user doesn't want to go to this
                        position.  E.g., when updating a "Last change"
                        timestamp in the first line: >

                                :let lnum = line(".")
                                :keepjumps normal gg
                                :call SetLastChange()
                                :keepjumps exe "normal " . lnum . "G"
<
                        Note that ":keepjumps" must be used for every command.
                        When invoking a function the commands in that function
                        can still change the jumplist.  Also, for
                        ":keepjumps exe 'command '" the "command" won't keep
                        jumps.  Instead use: ":exe 'keepjumps command'"

==============================================================================
8. Jumps                                            *jump-motions*

A "jump" is one of the following commands: "'", "`", "G", "/", "?", "n",
"N", "%", "(", ")", "[[", "]]", "{", "}", ":s", ":tag", "L", "M", "H" and
the commands that start editing a new file.  If you make the cursor "jump"
with one of these commands, the position of the cursor before the jump is
```

remembered. You can return to that position with the "''" and "``" command,
unless the line containing that position was changed or deleted.

                                                        CTRL-O
CTRL-O   Go to [count] Older cursor position in jump list
                        (not a motion command).
                        {not in Vi}
                        {not available without the |+jumplist| feature}

<Tab>   or   *CTRL-I* *<Tab>*
CTRL-I   Go to [count] newer cursor position in jump list
                        (not a motion command).
                        {not in Vi}
                        {not available without the |+jumplist| feature}

                                                        :ju *:jumps*
:ju[mps]   Print the jump list (not a motion command).
                        {not in Vi}
                        {not available without the |+jumplist| feature}

                                                        :cle *:clearjumps*
:cle[arjumps]   Clear the jump list of the current window.
                        {not in Vi}
                        {not available without the |+jumplist| feature}

                                                        jumplist
Jumps are remembered in a jump list. With the CTRL-O and CTRL-I command you
can go to cursor positions before older jumps, and back again. Thus you can
move up and down the list. There is a separate jump list for each window.
The maximum number of entries is fixed at 100.
{not available without the |+jumplist| feature}

For example, after three jump commands you have this jump list:

```
  jump line  col file/text ~
    3     1    0 some text ~
    2    70    0 another line ~
    1  1154   23 end. ~
 > ~
```

The "file/text" column shows the file name, or the text at the jump if it is
in the current file (an indent is removed and a long line is truncated to fit
in the window).

You are currently in line 1167. If you then use the CTRL-O command, the
cursor is put in line 1154. This results in:

```
  jump line  col file/text ~
    2     1    0 some text ~
    1    70    0 another line ~
 >  0  1154   23 end. ~
    1  1167    0 foo bar ~
```

The pointer will be set at the last used jump position. The next CTRL-O
command will use the entry above it, the next CTRL-I command will use the
entry below it. If the pointer is below the last entry, this indicates that
you did not use a CTRL-I or CTRL-O before. In this case the CTRL-O command
will cause the cursor position to be added to the jump list, so you can get
back to the position before the CTRL-O. In this case this is line 1167.

With more CTRL-O commands you will go to lines 70 and 1. If you use CTRL-I
you can go back to 1154 and 1167 again. Note that the number in the "jump"

column indicates the count for the CTRL-O or CTRL-I command that takes you to
this position.

If you use a jump command, the current line number is inserted at the end of
the jump list. If the same line was already in the jump list, it is removed.
The result is that when repeating CTRL-O you will get back to old positions
only once.

When the |:keepjumps| command modifier is used, jumps are not stored in the
jumplist. Jumps are also not stored in other cases, e.g., in a |:global|
command. You can explicitly add a jump by setting the ' mark with "m'". Note
that calling setpos() does not do this.

After the CTRL-O command that got you into line 1154 you could give another
jump command (e.g., "G"). The jump list would then become:

```
  jump line  col file/text ~
    4     1    0 some text ~
    3    70    0 another line ~
    2  1167    0 foo bar ~
    1  1154   23 end. ~
 > ~
```

The line numbers will be adjusted for deleted and inserted lines. This fails
if you stop editing a file without writing, like with ":n!".

When you split a window, the jumplist will be copied to the new window.

If you have included the ' item in the 'viminfo' option the jumplist will be
stored in the viminfo file and restored when starting Vim.

CHANGE LIST JUMPS *changelist* *change-list-jumps* *E664*

When making a change the cursor position is remembered. One position is
remembered for every change that can be undone, unless it is close to a
previous change. Two commands can be used to jump to positions of changes,
also those that have been undone:

g; *E662*

g; Go to [count] older position in change list.
If [count] is larger than the number of older change
positions go to the oldest change.
If there is no older change an error message is given.
(not a motion command)
{not in Vi}
{not available without the |+jumplist| feature}

g, *E663*

g, Go to [count] newer cursor position in change list.
Just like |g;| but in the opposite direction.
(not a motion command)
{not in Vi}
{not available without the |+jumplist| feature}

When using a count you jump as far back or forward as possible. Thus you can
use "999g;" to go to the first change for which the position is still
remembered. The number of entries in the change list is fixed and is the same
as for the |jumplist|.

When two undo-able changes are in the same line and at a column position less
than 'textwidth' apart only the last one is remembered. This avoids that a

sequence of small changes in a line, for example "xxxxx", adds many positions to the change list. When 'textwidth' is zero 'wrapmargin' is used. When that also isn't set a fixed number of 79 is used. Detail: For the computations bytes are used, not characters, to avoid a speed penalty (this only matters for multi-byte encodings).

Note that when text has been inserted or deleted the cursor position might be a bit different from the position of the change. Especially when lines have been deleted.

When the |:keepjumps| command modifier is used the position of a change is not remembered.

:changes

:changes Print the change list. A ">" character indicates the current position. Just after a change it is below the newest entry, indicating that "g;" takes you to the newest entry position. The first column indicates the count needed to take you to this position. Example:

```
change line  col text ~
    3     9    8 bla bla bla
    2    11   57 foo is a bar
    1    14   54 the latest changed line
>
```

The "3g;" command takes you to line 9. Then the output of ":changes is:

```
change line  col text ~
>   0     9    8 bla bla bla
    1    11   57 foo is a bar
    2    14   54 the latest changed line
```

Now you can use "g," to go to line 11 and "2g," to go to line 14.

==========

9. Various motions *various-motions*

%

% Find the next item in this line after or under the cursor and jump to its match. |inclusive| motion. Items can be:

```
([{}])          parenthesis or (curly/square) brackets
                (this can be changed with the
                'matchpairs' option)
/* */           start or end of C-style comment
#if, #ifdef, #else, #elif, #endif
                C preprocessor conditionals (when the
                cursor is on the # or no ([{
                following)
```

For other items the matchit plugin can be used, see |matchit-install|. This plugin also helps to skip matches in comments.

When 'cpoptions' contains "M" |cpo-M| backslashes before parens and braces are ignored. Without "M" the number of backslashes matters: an even number doesn't match with an odd number. Thus in "(\))" and "\((\)" the first and last parenthesis match.

When the '%' character is not present in 'cpoptions'
|cpo-%|, parens and braces inside double quotes are
ignored, unless the number of parens/braces in a line
is uneven and this line and the previous one does not
end in a backslash. '(', '{', '[', ']', '}' and ')'
are also ignored (parens and braces inside single
quotes). Note that this works fine for C, but not for
Perl, where single quotes are used for strings.

Nothing special is done for matches in comments. You
can either use the matchit plugin |matchit-install| or
put quotes around matches.

No count is allowed, {count}% jumps to a line {count}
percentage down the file |N%|. Using '%' on
#if/#else/#endif makes the movement linewise.

[(
[(go to [count] previous unmatched '('.
|exclusive| motion. {not in Vi}

[{
[{ go to [count] previous unmatched '{'.
|exclusive| motion. {not in Vi}

])
]) go to [count] next unmatched ')'.
|exclusive| motion. {not in Vi}

]}
]} go to [count] next unmatched '}'.
|exclusive| motion. {not in Vi}

The above four commands can be used to go to the start or end of the current
code block. It is like doing "%" on the '(', ')', '{' or '}' at the other
end of the code block, but you can do this from anywhere in the code block.
Very useful for C programs. Example: When standing on "case x:", "[{" will
bring you back to the switch statement.

]m
]m Go to [count] next start of a method (for Java or
similar structured language). When not before the
start of a method, jump to the start or end of the
class. When no '{' is found after the cursor, this is
an error. |exclusive| motion. {not in Vi}
]M
]M Go to [count] next end of a method (for Java or
similar structured language). When not before the end
of a method, jump to the start or end of the class.
When no '}' is found after the cursor, this is an
error. |exclusive| motion. {not in Vi}
[m
[m Go to [count] previous start of a method (for Java or
similar structured language). When not after the
start of a method, jump to the start or end of the
class. When no '{' is found before the cursor this is
an error. |exclusive| motion. {not in Vi}
[M
[M Go to [count] previous end of a method (for Java or
similar structured language). When not after the
end of a method, jump to the start or end of the
class. When no '}' is found before the cursor this is

```
                        an error. |exclusive| motion. {not in Vi}

The above two commands assume that the file contains a class with methods.
The class definition is surrounded in '{' and '}'.  Each method in the class
is also surrounded with '{' and '}'.  This applies to the Java language.  The
file looks like this: >

        // comment
        class foo {
                int method_one() {
                        body_one();
                }
                int method_two() {
                        body_two();
                }
        }
Starting with the cursor on "body_two()", using "[m" will jump to the '{' at
the start of "method_two()" (obviously this is much more useful when the
method is long!).  Using "2[m" will jump to the start of "method_one()".
Using "3[m" will jump to the start of the class.

                                                *[#*
[#                      go to [count] previous unmatched "#if" or "#else".
                        |exclusive| motion. {not in Vi}

                                                *]#*
]#                      go to [count] next unmatched "#else" or "#endif".
                        |exclusive| motion. {not in Vi}

These two commands work in C programs that contain #if/#else/#endif
constructs.  It brings you to the start or end of the #if/#else/#endif where
the current line is included.  You can then use "%" to go to the matching line.

                                                *[star* *[/*
[*  or  [/              go to [count] previous start of a C comment "/*".
                        |exclusive| motion. {not in Vi}

                                                *]star* *]/*
]*  or  ]/              go to [count] next end of a C comment "*/".
                        |exclusive| motion. {not in Vi}


                                                *H*
H                       To line [count] from top (Home) of window (default:
                        first line on the window) on the first non-blank
                        character |linewise|.  See also 'startofline' option.
                        Cursor is adjusted for 'scrolloff' option.

                                                *M*
M                       To Middle line of window, on the first non-blank
                        character |linewise|.  See also 'startofline' option.

                                                *L*
L                       To line [count] from bottom of window (default: Last
                        line on the window) on the first non-blank character
                        |linewise|.  See also 'startofline' option.
                        Cursor is adjusted for 'scrolloff' option.

<LeftMouse>             Moves to the position on the screen where the mouse
                        click is |exclusive|.  See also |<LeftMouse>|.  If the
                        position is in a status line, that window is made the
                        active window and the cursor is not moved.  {not in Vi}
```

```
 vim:tw=78:ts=8:ft=help:norl:
*scroll.txt*    For Vim version 8.0.  Last change: 2016 Nov 10


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Scrolling                                                *scrolling*

These commands move the contents of the window.  If the cursor position is
moved off of the window, the cursor is moved onto the window (with
'scrolloff' screen lines around it).  A page is the number of lines in the
window minus two.  The mnemonics for these commands may be a bit confusing.
Remember that the commands refer to moving the window (the part of the buffer
that you see) upwards or downwards in the buffer.  When the window moves
upwards in the buffer, the text in the window moves downwards on your screen.

See section |03.7| of the user manual for an introduction.

1. Scrolling downwards          |scroll-down|
2. Scrolling upwards            |scroll-up|
3. Scrolling relative to cursor |scroll-cursor|
4. Scrolling horizontally       |scroll-horizontal|
5. Scrolling synchronously      |scroll-binding|
6. Scrolling with a mouse wheel |scroll-mouse-wheel|

==============================================================================
1. Scrolling downwards                                   *scroll-down*

The following commands move the edit window (the part of the buffer that you
see) downwards (this means that more lines downwards in the text buffer can be
seen):

                                                         *CTRL-E*
CTRL-E                  Scroll window [count] lines downwards in the buffer.
                        Mnemonic: Extra lines.

                                                         *CTRL-D*
CTRL-D                  Scroll window Downwards in the buffer.  The number of
                        lines comes from the 'scroll' option (default: half a
                        screen).  If [count] given, first set 'scroll' option
                        to [count].  The cursor is moved the same number of
                        lines down in the file (if possible; when lines wrap
                        and when hitting the end of the file there may be a
                        difference).  When the cursor is on the last line of
                        the buffer nothing happens and a beep is produced.
                        See also 'startofline' option.
                        {difference from vi: Vim scrolls 'scroll' screen
                        lines, instead of file lines; makes a difference when
                        lines wrap}

<S-Down>        or                               *<S-Down>* *<kPageDown>*
<PageDown>      or                               *<PageDown>* *CTRL-F*
CTRL-F                  Scroll window [count] pages Forwards (downwards) in
                        the buffer.  See also 'startofline' option.
                        When there is only one window the 'window' option
                        might be used.

                                                         *z+*
z+                      Without [count]: Redraw with the line just below the
                        window at the top of the window.  Put the cursor in
```

that line, at the first non-blank in the line.
With [count]: just like "z<CR>".

==
2. Scrolling upwards *scroll-up*

The following commands move the edit window (the part of the buffer that you see) upwards (this means that more lines upwards in the text buffer can be seen):

CTRL-Y
CTRL-Y Scroll window [count] lines upwards in the buffer.
Note: When using the MS-Windows key bindings CTRL-Y is remapped to redo.

CTRL-U
CTRL-U Scroll window Upwards in the buffer. The number of lines comes from the 'scroll' option (default: half a screen). If [count] given, first set the 'scroll' option to [count]. The cursor is moved the same number of lines up in the file (if possible; when lines wrap and when hitting the end of the file there may be a difference). When the cursor is on the first line of the buffer nothing happens and a beep is produced. See also 'startofline' option.
{difference from vi: Vim scrolls 'scroll' screen lines, instead of file lines; makes a difference when lines wrap}

<S-Up> *<kPageUp>*
<S-Up> or
<PageUp> *CTRL-B*
<PageUp> or
CTRL-B Scroll window [count] pages Backwards (upwards) in the buffer. See also 'startofline' option.
When there is only one window the 'window' option might be used.

*z^*
z^ Without [count]: Redraw with the line just above the window at the bottom of the window. Put the cursor in that line, at the first non-blank in the line.
With [count]: First scroll the text to put the [count] line at the bottom of the window, then redraw with the line which is now at the top of the window at the bottom of the window. Put the cursor in that line, at the first non-blank in the line.

==
3. Scrolling relative to cursor *scroll-cursor*

The following commands reposition the edit window (the part of the buffer that you see) while keeping the cursor on the same line. Note that the 'scrolloff' option may cause context lines to show above and below the cursor.

z<CR>
z<CR> Redraw, line [count] at top of window (default cursor line). Put cursor at first non-blank in the line.

zt
zt Like "z<CR>", but leave the cursor in the same column. {not in Vi}

zN<CR>
z{height}<CR> Redraw, make window {height} lines tall. This is useful to make the number of lines small when screen updating is very slow. Cannot make the height more than the physical screen height.

z.
z. Redraw, line [count] at center of window (default cursor line). Put cursor at first non-blank in the line.

zz
zz Like "z.", but leave the cursor in the same column. Careful: If caps-lock is on, this command becomes "ZZ": write buffer and exit! {not in Vi}

z-
z- Redraw, line [count] at bottom of window (default cursor line). Put cursor at first non-blank in the line.

zb
zb Like "z-", but leave the cursor in the same column. {not in Vi}

==
4. Scrolling horizontally *scroll-horizontal*

For the following four commands the cursor follows the screen. If the character that the cursor is on is moved off the screen, the cursor is moved to the closest character that is on the screen. The value of 'sidescroll' is not used.

zl *z<Right>*
z<Right> or
zl Move the view on the text [count] characters to the right, thus scroll the text [count] characters to the left. This only works when 'wrap' is off. {not in Vi}

zh *z<Left>*
z<Left> or
zh Move the view on the text [count] characters to the left, thus scroll the text [count] characters to the right. This only works when 'wrap' is off. {not in Vi}

zL
zL Move the view on the text half a screenwidth to the right, thus scroll the text half a screenwidth to the left. This only works when 'wrap' is off. {not in Vi}

zH
zH Move the view on the text half a screenwidth to the left, thus scroll the text half a screenwidth to the right. This only works when 'wrap' is off. {not in Vi}

For the following two commands the cursor is not moved in the text, only the text scrolls on the screen.

zs
zs Scroll the text horizontally to position the cursor

at the start (left side) of the screen. This only works when 'wrap' is off. {not in Vi}

ze

ze Scroll the text horizontally to position the cursor at the end (right side) of the screen. This only works when 'wrap' is off. {not in Vi}

==========

5. Scrolling synchronously *scroll-binding*

Occasionally, it is desirable to bind two or more windows together such that when one window is scrolled, the other windows are also scrolled. In Vim, windows can be given this behavior by setting the (window-specific) 'scrollbind' option. When a window that has 'scrollbind' set is scrolled, all other 'scrollbind' windows are scrolled the same amount, if possible. The behavior of 'scrollbind' can be modified by the 'scrollopt' option.

When using the scrollbars, the binding only happens when scrolling the window with focus (where the cursor is). You can use this to avoid scroll-binding for a moment without resetting options.

When a window also has the 'diff' option set, the scroll-binding uses the differences between the two buffers to synchronize the position precisely. Otherwise the following method is used.

scrollbind-relative

Each 'scrollbind' window keeps track of its "relative offset," which can be thought of as the difference between the current window's vertical scroll position and the other window's vertical scroll position. When one of the 'scrollbind' windows is asked to vertically scroll past the beginning or end limit of its text, the window no longer scrolls, but remembers how far past the limit it wishes to be. The window keeps this information so that it can maintain the same relative offset, regardless of its being asked to scroll past its buffer's limits.

However, if a 'scrollbind' window that has a relative offset that is past its buffer's limits is given the cursor focus, the other 'scrollbind' windows must jump to a location where the current window's relative offset is valid. This behavior can be changed by clearing the "jump" flag from the 'scrollopt' option.

syncbind *:syncbind* *:sync*

:syncbind Force all 'scrollbind' windows to have the same relative offset. I.e., when any of the 'scrollbind' windows is scrolled to the top of its buffer, all of the 'scrollbind' windows will also be at the top of their buffers.

scrollbind-quickadj

The 'scrollbind' flag is meaningful when using keyboard commands to vertically scroll a window, and also meaningful when using the vertical scrollbar of the window which has the cursor focus. However, when using the vertical scrollbar of a window which doesn't have the cursor focus, 'scrollbind' is ignored. This allows quick adjustment of the relative offset of 'scrollbind' windows.

==========

6. Scrolling with a mouse wheel *scroll-mouse-wheel*

When your mouse has a scroll wheel, it should work with Vim in the GUI. How it works depends on your system. It might also work in an xterm |xterm-mouse-wheel|. By default only vertical scroll wheels are supported,

but some GUIs also support horizontal scroll wheels.

For the Win32 GUI the scroll action is hard coded. It works just like dragging the scrollbar of the current window. How many lines are scrolled depends on your mouse driver. If the scroll action causes input focus problems, see |intellimouse-wheel-problems|.

For the X11 GUIs (Motif, Athena and GTK) scrolling the wheel generates key presses <ScrollWheelUp>, <ScrollWheelDown>, <ScrollWheelLeft> and <ScrollWheelRight>. For example, if you push the scroll wheel upwards a <ScrollWheelUp> key press is generated causing the window to scroll upwards (while the text is actually moving downwards). The default action for these keys are:

<ScrollWheelUp>	scroll three lines up	*<ScrollWheelUp>*
<S-ScrollWheelUp>	scroll one page up	*<S-ScrollWheelUp>*
<C-ScrollWheelUp>	scroll one page up	*<C-ScrollWheelUp>*
<ScrollWheelDown>	scroll three lines down	*<ScrollWheelDown>*
<S-ScrollWheelDown>	scroll one page down	*<S-ScrollWheelDown>*
<C-ScrollWheelDown>	scroll one page down	*<C-ScrollWheelDown>*
<ScrollWheelLeft>	scroll six columns left	*<ScrollWheelLeft>*
<S-ScrollWheelLeft>	scroll one page left	*<S-ScrollWheelLeft>*
<C-ScrollWheelLeft>	scroll one page left	*<C-ScrollWheelLeft>*
<ScrollWheelRight>	scroll six columns right	*<ScrollWheelRight>*
<S-ScrollWheelRight>	scroll one page right	*<S-ScrollWheelRight>*
<C-ScrollWheelRight>	scroll one page right	*<C-ScrollWheelRight>*

This should work in all modes, except when editing the command line.

Note that horizontal scrolling only works if 'nowrap' is set. Also, unless the "h" flag in 'guioptions' is set, the cursor moves to the longest visible line if the cursor line is about to be scrolled off the screen (similarly to how the horizontal scrollbar works).

You can modify the default behavior by mapping the keys. For example, to make the scroll wheel move one line or half a page in Normal mode: >

```
:map <ScrollWheelUp> <C-Y>
:map <S-ScrollWheelUp> <C-U>
:map <ScrollWheelDown> <C-E>
:map <S-ScrollWheelDown> <C-D>
```

You can also use Alt and Ctrl modifiers.

This only works when Vim gets the scroll wheel events, of course. You can check if this works with the "xev" program.

When using XFree86, the /etc/XF86Config file should have the correct entry for your mouse. For FreeBSD, this entry works for a Logitech scrollmouse: >

```
Protocol     "MouseMan"
Device       "/dev/psm0"
ZAxisMapping 4 5
```

See the XFree86 documentation for information.

<MouseDown> *<MouseUp>*

The keys <MouseDown> and <MouseUp> have been deprecated. Use <ScrollWheelUp> instead of <MouseDown> and use <ScrollWheelDown> instead of <MouseUp>.

xterm-mouse-wheel

To use the mouse wheel in a new xterm you only have to make the scroll wheel work in your Xserver, as mentioned above.

To use the mouse wheel in an older xterm you must do this:

1. Make it work in your Xserver, as mentioned above.
2. Add translations for the xterm, so that the xterm will pass a scroll event to Vim as an escape sequence.

```
3. Add mappings in Vim, to interpret the escape sequences as <ScrollWheelDown>
   or <ScrollWheelUp> keys.

You can do the translations by adding this to your ~.Xdefaults file (or other
file where your X resources are kept): >

  XTerm*VT100.Translations:             #override \n\
                s<Btn4Down>: string("0x9b") string("[64~") \n\
                s<Btn5Down>: string("0x9b") string("[65~") \n\
                <Btn4Down>: string("0x9b") string("[62~") \n\
                <Btn5Down>: string("0x9b") string("[63~") \n\
                <Btn4Up>: \n\
                <Btn5Up>:

Add these mappings to your vimrc file: >
        :map <M-Esc>[62~ <ScrollWheelUp>
        :map! <M-Esc>[62~ <ScrollWheelUp>
        :map <M-Esc>[63~ <ScrollWheelDown>
        :map! <M-Esc>[63~ <ScrollWheelDown>
        :map <M-Esc>[64~ <S-ScrollWheelUp>
        :map! <M-Esc>[64~ <S-ScrollWheelUp>
        :map <M-Esc>[65~ <S-ScrollWheelDown>
        :map! <M-Esc>[65~ <S-ScrollWheelDown>
<
 vim:tw=78:ts=8:ft=help:norl:
*insert.txt*    For Vim version 8.0.  Last change: 2017 May 30


                  VIM REFERENCE MANUAL    by Bram Moolenaar


                                                *Insert* *Insert-mode*
Inserting and replacing text                            *mode-ins-repl*

Most of this file is about Insert and Replace mode.  At the end are a few
commands for inserting text in other ways.

An overview of the most often used commands can be found in chapter 24 of the
user manual |usr_24.txt|.

1. Special keys                                         |ins-special-keys|
2. Special special keys                                 |ins-special-special|
3. 'textwidth' and 'wrapmargin' options                 |ins-textwidth|
4. 'expandtab', 'smarttab' and 'softtabstop'  options   |ins-expandtab|
5. Replace mode                                         |Replace-mode|
6. Virtual Replace mode                                 |Virtual-Replace-mode|
7. Insert mode completion                               |ins-completion|
8. Insert mode commands                                 |inserting|
9. Ex insert commands                                   |inserting-ex|
10. Inserting a file                                    |inserting-file|

Also see 'virtualedit', for moving the cursor to positions where there is no
character.  Useful for editing a table.

==============================================================================
1. Special keys                                         *ins-special-keys*

In Insert and Replace mode, the following characters have a special meaning;
other characters are inserted directly.  To insert one of these special
characters into the buffer, precede it with CTRL-V.  To insert a <Nul>
character use "CTRL-V CTRL-@" or "CTRL-V 000".  On some systems, you have to
use "CTRL-V 003" to insert a CTRL-C.  Note: When CTRL-V is mapped you can
```

```
often use CTRL-Q instead |i_CTRL-Q|.

If you are working in a special language mode when inserting text, see the
'langmap' option, |'langmap'|, on how to avoid switching this mode on and off
all the time.

If you have 'insertmode' set, <Esc> and a few other keys get another meaning.
See |'insertmode'|.

char            action  ~
-----------------------------------------------------------------------
                                                *i_CTRL-[* *i_<Esc>*
<Esc> or CTRL-[ End insert or Replace mode, go back to Normal mode.  Finish
                abbreviation.
                Note: If your <Esc> key is hard to hit on your keyboard, train
                yourself to use CTRL-[.
                If Esc doesn't work and you are using a Mac, try CTRL-Esc.
                Or disable Listening under Accessibility preferences.
                                                *i_CTRL-C*
CTRL-C          Quit insert mode, go back to Normal mode.  Do not check for
                abbreviations.  Does not trigger the |InsertLeave| autocommand
                event.

                                                *i_CTRL-@*
CTRL-@          Insert previously inserted text and stop insert.  {Vi: only
                when typed as first char, only up to 128 chars}
                                                *i_CTRL-A*
CTRL-A          Insert previously inserted text.  {not in Vi}

                                                *i_CTRL-H* *i_<BS>* *i_BS*
<BS> or CTRL-H  Delete the character before the cursor (see |i_backspacing|
                about joining lines).
                See |:fixdel| if your <BS> key does not do what you want.
                {Vi: does not delete autoindents}
                                                *i_<Del>* *i_DEL*
<Del>           Delete the character under the cursor.  If the cursor is at
                the end of the line, and the 'backspace' option includes
                "eol", delete the <EOL>; the next line is appended after the
                current one.
                See |:fixdel| if your <Del> key does not do what you want.
                {not in Vi}
                                                *i_CTRL-W*
CTRL-W          Delete the word before the cursor (see |i_backspacing| about
                joining lines).  See the section "word motions",
                |word-motions|, for the definition of a word.
                                                *i_CTRL-U*
CTRL-U          Delete all entered characters before the cursor in the current
                line.  If there are no newly entered characters and
                'backspace' is not empty, delete all characters before the
                cursor in the current line.
                See |i_backspacing| about joining lines.
                                                *i_CTRL-I* *i_<Tab>* *i_Tab*
<Tab> or CTRL-I Insert a tab.  If the 'expandtab' option is on, the
                equivalent number of spaces is inserted (use CTRL-V <Tab> to
                avoid the expansion; use CTRL-Q <Tab> if CTRL-V is mapped
                |i_CTRL-Q|).  See also the 'smarttab' option and
                |ins-expandtab|.
                                                *i_CTRL-J* *i_<NL>*
<NL> or CTRL-J  Begin new line.
                                                *i_CTRL-M* *i_<CR>*
<CR> or CTRL-M  Begin new line.
                                                *i_CTRL-K*
```

```
CTRL-K {char1} [char2]
               Enter digraph (see |digraphs|).  When {char1} is a special
               key, the code for that key is inserted in <> form.  For
               example, the string "<S-Space>" can be entered by typing
               <C-K><S-Space> (two keys).  Neither char is considered for
               mapping.  {not in Vi}

CTRL-N         Find next keyword (see |i_CTRL-N|).  {not in Vi}
CTRL-P         Find previous keyword (see |i_CTRL-P|).  {not in Vi}

CTRL-R {0-9a-z"%#*+:.-=}                                        *i_CTRL-R*
               Insert the contents of a 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.  If you have options like
               'textwidth', 'formatoptions', or 'autoindent' set, this will
               influence what will be inserted.  This is different from what
               happens with the "p" command and pasting with the mouse.
               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 inserted text
                       '-'     the last small (less than a line) delete
                                                       *i_CTRL-R_=*
                       '='     the expression register: you are prompted to
                               enter an expression (see |expression|)
                               Note that 0x80 (128 decimal) is used for
                               special keys.  E.g., you can use this to move
                               the cursor up:
                                       CTRL-R ="\<Up>"
                               Use CTRL-R CTRL-R to insert text literally.
                               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.
                               When append() or setline() is invoked the undo
                               sequence will be broken.
               See |registers| about registers.  {not in Vi}

CTRL-R CTRL-R {0-9a-z"%#*+/:.-=}                        *i_CTRL-R_CTRL-R*
               Insert the contents of a register.  Works like using a single
               CTRL-R, but the text is inserted literally, not as if typed.
               This differs when the register contains characters like <BS>.
               Example, where register a contains "ab^Hc": >
        CTRL-R a               results in "ac".
        CTRL-R CTRL-R a        results in "ab^Hc".
<              Options 'textwidth', 'formatoptions', etc. still apply.  If
               you also want to avoid these, use CTRL-R CTRL-O, see below.
               The '.' register (last inserted text) is still inserted as
               typed.  {not in Vi}

CTRL-R CTRL-O {0-9a-z"%#*+/:.-=}                        *i_CTRL-R_CTRL-O*
               Insert the contents of a register literally and don't
               auto-indent.  Does the same as pasting with the mouse
```

```
                |<MiddleMouse>|. When the register is linewise this will
                insert the text above the current line, like with `P`.
                Does not replace characters!
                The '.' register (last inserted text) is still inserted as
                typed.  {not in Vi}

CTRL-R CTRL-P {0-9a-z"%#*+/:.-=}                        *i_CTRL-R_CTRL-P*
                Insert the contents of a register literally and fix the
                indent, like |[<MiddleMouse>|.
                Does not replace characters!
                The '.' register (last inserted text) is still inserted as
                typed.  {not in Vi}

                                                        *i_CTRL-T*
CTRL-T          Insert one shiftwidth of indent at the start of the current
                line.  The indent is always rounded to a 'shiftwidth' (this is
                vi compatible).  {Vi: only when in indent}
                                                        *i_CTRL-D*
CTRL-D          Delete one shiftwidth of indent at the start of the current
                line.  The indent is always rounded to a 'shiftwidth' (this is
                vi compatible).  {Vi: CTRL-D works only when used after
                autoindent}
                                                        *i_0_CTRL-D*
0 CTRL-D        Delete all indent in the current line.  {Vi: CTRL-D works
                only when used after autoindent}
                                                        *i_^_CTRL-D*
^ CTRL-D        Delete all indent in the current line.  The indent is
                restored in the next line.  This is useful when inserting a
                label.  {Vi: CTRL-D works only when used after autoindent}

                                                        *i_CTRL-V*
CTRL-V          Insert next non-digit literally.  For special keys, the
                terminal code is inserted.  It's also possible to enter the
                decimal, octal or hexadecimal value of a character
                |i_CTRL-V_digit|.
                The characters typed right after CTRL-V are not considered for
                mapping.  {Vi: no decimal byte entry}
                Note: When CTRL-V is mapped (e.g., to paste text) you can
                often use CTRL-Q instead |i_CTRL-Q|.

                                                        *i_CTRL-Q*
CTRL-Q          Same as CTRL-V.
                Note: Some terminal connections may eat CTRL-Q, it doesn't
                work then.  It does work in the GUI.

CTRL-X          Enter CTRL-X mode.  This is a sub-mode where commands can
                be given to complete words or scroll the window.  See
                |i_CTRL-X| and |ins-completion|. {not in Vi}

                                                        *i_CTRL-E*
CTRL-E          Insert the character which is below the cursor.  {not in Vi}
                                                        *i_CTRL-Y*
CTRL-Y          Insert the character which is above the cursor.  {not in Vi}
                Note that for CTRL-E and CTRL-Y 'textwidth' is not used, to be
                able to copy characters from a long line.

                                                        *i_CTRL-_*
CTRL-_          Switch between languages, as follows:
                -  When in a rightleft window, revins and nohkmap are toggled,
                   since English will likely be inserted in this case.
                -  When in a norightleft window, revins and hkmap are toggled,
                   since Hebrew will likely be inserted in this case.
```

```
                CTRL-_ moves the cursor to the end of the typed text.

                This command is only available when the 'allowrevins' option
                is set.
                Please refer to |rileft.txt| for more information about
                right-to-left mode.
                {not in Vi}
                Only if compiled with the |+rightleft| feature.

                                                        *i_CTRL-^*
CTRL-^          Toggle the use of typing language characters.
                When language |:lmap| mappings are defined:
                - If 'iminsert' is 1 (langmap mappings used) it becomes 0 (no
                  langmap mappings used).
                - If 'iminsert' has another value it becomes 1, thus langmap
                  mappings are enabled.
                When no language mappings are defined:
                - If 'iminsert' is 2 (Input Method used) it becomes 0 (no
                  Input Method used).
                - If 'iminsert' has another value it becomes 2, thus the Input
                  Method is enabled.
                When set to 1, the value of the "b:keymap_name" variable, the
                'keymap' option or "<lang>" appears in the status line.
                The 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.
                {not in Vi}

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

                                                        *i_<Insert>*
<Insert>        Toggle between Insert and Replace mode.  {not in Vi}
------------------------------------------------------------------------------

                                                        *i_backspacing*
The effect of the <BS>, CTRL-W, and CTRL-U depend on the 'backspace' option
(unless 'revins' is set).  This is a comma separated list of items:

item        action ~
indent      allow backspacing over autoindent
eol         allow backspacing over end-of-line (join lines)
start       allow backspacing over the start position of insert; CTRL-W and
            CTRL-U stop once at the start position

When 'backspace' is empty, Vi compatible backspacing is used.  You cannot
backspace over autoindent, before column 1 or before where insert started.

For backwards compatibility the values "0", "1" and "2" are also allowed, see
|'backspace'|.

If the 'backspace' option does contain "eol" and the cursor is in column 1
when one of the three keys is used, the current line is joined with the
previous line.  This effectively deletes the <EOL> in front of the cursor.
{Vi: does not cross lines, does not delete past start position of insert}

                                                        *i_CTRL-V_digit*
With CTRL-V the decimal, octal or hexadecimal value of a character can be
entered directly.  This way you can enter any character, except a line break
(<NL>, value 10).  There are five ways to enter the character value:
```

```
first char      mode         max nr of chars   max value ~
(none)          decimal            3            255
o or O          octal              3            377      (255)
x or X          hexadecimal        2            ff       (255)
u               hexadecimal        4            ffff     (65535)
U               hexadecimal        8            7fffffff (2147483647)
```

Normally you would type the maximum number of characters. Thus to enter a space (value 32) you would type <C-V>032. You can omit the leading zero, in which case the character typed after the number must be a non-digit. This happens for the other modes as well: As soon as you type a character that is invalid for the mode, the value before it will be used and the "invalid" character is dealt with in the normal way.

If you enter a value of 10, it will end up in the file as a 0. The 10 is a <NL>, which is used internally to represent the <Nul> character. When writing the buffer to a file, the <NL> character is translated into <Nul>. The <NL> character is written at the end of each line. Thus if you want to insert a <NL> character in a file you will have to make a line break.

i_CTRL-X *insert_expand*

CTRL-X enters a sub-mode where several commands can be used. Most of these commands do keyword completion; see |ins-completion|. These are not available when Vim was compiled without the |+insert_expand| feature.

Two commands can be used to scroll the window up or down, without exiting insert mode:

i_CTRL-X_CTRL-E

CTRL-X CTRL-E scroll window one line up.
When doing completion look here: |complete_CTRL-E|

i_CTRL-X_CTRL-Y

CTRL-X CTRL-Y scroll window one line down.
When doing completion look here: |complete_CTRL-Y|

After CTRL-X is pressed, each CTRL-E (CTRL-Y) scrolls the window up (down) by one line unless that would cause the cursor to move from its current position in the file. As soon as another key is pressed, CTRL-X mode is exited and that key is interpreted as in Insert mode.

==

2. Special special keys *ins-special-special*

The following keys are special. They stop the current insert, do something, and then restart insertion. This means you can do something without getting out of Insert mode. This is very handy if you prefer to use the Insert mode all the time, just like editors that don't have a separate Normal mode. You may also want to set the 'backspace' option to "indent,eol,start" and set the 'insertmode' option. You can use CTRL-O if you want to map a function key to a command.

The changes (inserted or deleted characters) before and after these keys can be undone separately. Only the last change can be redone and always behaves like an "i" command.

```
char            action  ~
-------------------------------------------------------------------
<Up>            cursor one line up                          *i_<Up>*
<Down>          cursor one line down                        *i_<Down>*
```

```
CTRL-G <Up>     cursor one line up, insert start column      *i_CTRL-G_<Up>*
CTRL-G k        cursor one line up, insert start column      *i_CTRL-G_k*
CTRL-G CTRL-K   cursor one line up, insert start column      *i_CTRL-G_CTRL-K*
CTRL-G <Down>   cursor one line down, insert start column    *i_CTRL-G_<Down>*
CTRL-G j        cursor one line down, insert start column    *i_CTRL-G_j*
CTRL-G CTRL-J   cursor one line down, insert start column    *i_CTRL-G_CTRL-J*
<Left>          cursor one character left                    *i_<Left>*
<Right>         cursor one character right                   *i_<Right>*
<S-Left>        cursor one word back (like "b" command)      *i_<S-Left>*
<C-Left>        cursor one word back (like "b" command)      *i_<C-Left>*
<S-Right>       cursor one word forward (like "w" command)   *i_<S-Right>*
<C-Right>       cursor one word forward (like "w" command)   *i_<C-Right>*
<Home>          cursor to first char in the line             *i_<Home>*
<End>           cursor to after last char in the line        *i_<End>*
<C-Home>        cursor to first char in the file             *i_<C-Home>*
<C-End>         cursor to after last char in the file        *i_<C-End>*
<LeftMouse>     cursor to position of mouse click            *i_<LeftMouse>*
<S-Up>          move window one page up                      *i_<S-Up>*
<PageUp>        move window one page up                      *i_<PageUp>*
<S-Down>        move window one page down                    *i_<S-Down>*
<PageDown>      move window one page down                    *i_<PageDown>*
<ScrollWheelDown>    move window three lines down       *i_<ScrollWheelDown>*
<S-ScrollWheelDown>  move window one page down          *i_<S-ScrollWheelDown>*
<ScrollWheelUp>      move window three lines up         *i_<ScrollWheelUp>*
<S-ScrollWheelUp>    move window one page up            *i_<S-ScrollWheelUp>*
<ScrollWheelLeft>    move window six columns left       *i_<ScrollWheelLeft>*
<S-ScrollWheelLeft>  move window one page left          *i_<S-ScrollWheelLeft>*
<ScrollWheelRight>   move window six columns right      *i_<ScrollWheelRight>*
<S-ScrollWheelRight> move window one page right         *i_<S-ScrollWheelRight>*
CTRL-O          execute one command, return to Insert mode   *i_CTRL-O*
CTRL-\ CTRL-O   like CTRL-O but don't move the cursor        *i_CTRL-\_CTRL-O*
CTRL-L          when 'insertmode' is set: go to Normal mode  *i_CTRL-L*
CTRL-G u        break undo sequence, start new change        *i_CTRL-G_u*
CTRL-G U        don't break undo with next left/right cursor *i_CTRL-G_U*
                movement (but only if the cursor stays
                within same the line)
-------------------------------------------------------------------------
```

Note: If the cursor keys take you out of Insert mode, check the 'noesckeys' option.

The CTRL-O command sometimes has a side effect: If the cursor was beyond the end of the line, it will be put on the last character in the line. In mappings it's often better to use <Esc> (first put an "x" in the text, <Esc> will then always put the cursor on it). Or use CTRL-\ CTRL-O, but then beware of the cursor possibly being beyond the end of the line. Note that the command following CTRL-\ CTRL-O can still move the cursor, it is not restored to its original position.

The CTRL-O command takes you to Normal mode. If you then use a command enter Insert mode again it normally doesn't nest. Thus when typing "a<C-O>a" and then <Esc> takes you back to Normal mode, you do not need to type <Esc> twice. An exception is when not typing the command, e.g. when executing a mapping or sourcing a script. This makes mappings work that briefly switch to Insert mode.

The shifted cursor keys are not available on all terminals.

Another side effect is that a count specified before the "i" or "a" command is ignored. That is because repeating the effect of the command after CTRL-O is too complicated.

```
An example for using CTRL-G u: >

        :inoremap <C-H> <C-G>u<C-H>

This redefines the backspace key to start a new undo sequence.  You can now
undo the effect of the backspace key, without changing what you typed before
that, with CTRL-O u.  Another example: >

        :inoremap <CR> <C-]><C-G>u<CR>

This breaks undo at each line break.  It also expands abbreviations before
this.

An example for using CTRL-G U: >

        inoremap <Left>  <C-G>U<Left>
        inoremap <Right> <C-G>U<Right>
        inoremap <expr> <Home> col('.') == match(getline('.'), '\S') + 1 ?
         \ repeat('<C-G>U<Left>', col('.') - 1) :
         \ (col('.') < match(getline('.'), '\S') ?
         \     repeat('<C-G>U<Right>', match(getline('.'), '\S') + 0) :
         \     repeat('<C-G>U<Left>', col('.') - 1 - match(getline('.'), '\S')))
        inoremap <expr> <End> repeat('<C-G>U<Right>', col('$') - col('.'))
        inoremap ( ()<C-G>U<Left>

This makes it possible to use the cursor keys in Insert mode, without breaking
the undo sequence and therefore using |.| (redo) will work as expected.
Also entering a text like (with the "(" mapping from above): >

   Lorem ipsum (dolor

will be repeatable by the |.|to the expected

   Lorem ipsum (dolor)

Using CTRL-O splits undo: the text typed before and after it is undone
separately.  If you want to avoid this (e.g., in a mapping) you might be able
to use CTRL-R = |i_CTRL-R|.  E.g., to call a function: >
        :imap <F2> <C-R>=MyFunc()<CR>

When the 'whichwrap' option is set appropriately, the <Left> and <Right>
keys on the first/last character in the line make the cursor wrap to the
previous/next line.

The CTRL-G j and CTRL-G k commands can be used to insert text in front of a
column.  Example: >
   int i;
   int j;
Position the cursor on the first "int", type "istatic <C-G>j       ".  The
result is: >
   static int i;
          int j;
When inserting the same text in front of the column in every line, use the
Visual blockwise command "I" |v_b_I|.

==============================================================================
3. 'textwidth' and 'wrapmargin' options                 *ins-textwidth*

The 'textwidth' option can be used to automatically break a line before it
gets too long.  Set the 'textwidth' option to the desired maximum line
length.  If you then type more characters (not spaces or tabs), the
last word will be put on a new line (unless it is the only word on the
```

line). If you set 'textwidth' to 0, this feature is disabled.

The 'wrapmargin' option does almost the same. The difference is that
'textwidth' has a fixed width while 'wrapmargin' depends on the width of the
screen. When using 'wrapmargin' this is equal to using 'textwidth' with a
value equal to (columns - 'wrapmargin'), where columns is the width of the
screen.

When 'textwidth' and 'wrapmargin' are both set, 'textwidth' is used.

If you don't really want to break the line, but view the line wrapped at a
convenient place, see the 'linebreak' option.

The line is only broken automatically when using Insert mode, or when
appending to a line. When in replace mode and the line length is not
changed, the line will not be broken.

Long lines are broken if you enter a non-white character after the margin.
The situations where a line will be broken can be restricted by adding
characters to the 'formatoptions' option:
"l" Only break a line if it was not longer than 'textwidth' when the insert
     started.
"v" Only break at a white character that has been entered during the
     current insert command. This is mostly Vi-compatible.
"lv" Only break if the line was not longer than 'textwidth' when the insert
     started and only at a white character that has been entered during the
     current insert command. Only differs from "l" when entering non-white
     characters while crossing the 'textwidth' boundary.

Normally an internal function will be used to decide where to break the line.
If you want to do it in a different way set the 'formatexpr' option to an
expression that will take care of the line break.

If you want to format a block of text, you can use the "gq" operator. Type
"gq" and a movement command to move the cursor to the end of the block. In
many cases, the command "gq}" will do what you want (format until the end of
paragraph). Alternatively, you can use "gqap", which will format the whole
paragraph, no matter where the cursor currently is. Or you can use Visual
mode: hit "v", move to the end of the block, and type "gq". See also |gq|.

==
4. 'expandtab', 'smarttab' and 'softtabstop' options   *ins-expandtab*

If the 'expandtab' option is on, spaces will be used to fill the amount of
whitespace of the tab. If you want to enter a real <Tab>, type CTRL-V first
(use CTRL-Q when CTRL-V is mapped |i_CTRL-Q|).
The 'expandtab' option is off by default. Note that in Replace mode, a single
character is replaced with several spaces. The result of this is that the
number of characters in the line increases. Backspacing will delete one
space at a time. The original character will be put back for only one space
that you backspace over (the last one). {Vi does not have the 'expandtab'
option}

                                                        ins-smarttab
When the 'smarttab' option is on, a <Tab> inserts 'shiftwidth' positions at
the beginning of a line and 'tabstop' positions in other places. This means
that often spaces instead of a <Tab> character are inserted. When 'smarttab'
is off, a <Tab> always inserts 'tabstop' positions, and 'shiftwidth' is only
used for ">>" and the like. {not in Vi}

                                                        ins-softtabstop
When the 'softtabstop' option is non-zero, a <Tab> inserts 'softtabstop'

positions, and a <BS> used to delete white space, will delete 'softtabstop'
positions. This feels like 'tabstop' was set to 'softtabstop', but a real
<Tab> character still takes 'tabstop' positions, so your file will still look
correct when used by other applications.

If 'softtabstop' is non-zero, a <BS> will try to delete as much white space to
move to the previous 'softtabstop' position, except when the previously
inserted character is a space, then it will only delete the character before
the cursor. Otherwise you cannot always delete a single character before the
cursor. You will have to delete 'softtabstop' characters first, and then type
extra spaces to get where you want to be.

==
5. Replace mode   *Replace* *Replace-mode* *mode-replace*

Enter Replace mode with the "R" command in normal mode.

In Replace mode, one character in the line is deleted for every character you
type. If there is no character to delete (at the end of the line), the
typed character is appended (as in Insert mode). Thus the number of
characters in a line stays the same until you get to the end of the line.
If a <NL> is typed, a line break is inserted and no character is deleted.

Be careful with <Tab> characters. If you type a normal printing character in
its place, the number of characters is still the same, but the number of
columns will become smaller.

If you delete characters in Replace mode (with <BS>, CTRL-W, or CTRL-U), what
happens is that you delete the changes. The characters that were replaced
are restored. If you had typed past the existing text, the characters you
added are deleted. This is effectively a character-at-a-time undo.

If the 'expandtab' option is on, a <Tab> will replace one character with
several spaces. The result of this is that the number of characters in the
line increases. Backspacing will delete one space at a time. The original
character will be put back for only one space that you backspace over (the
last one). {Vi does not have the 'expandtab' option}

==
6. Virtual Replace mode   *vreplace-mode* *Virtual-Replace-mode*

Enter Virtual Replace mode with the "gR" command in normal mode.
{not available when compiled without the |+vreplace| feature}
{Vi does not have Virtual Replace mode}

Virtual Replace mode is similar to Replace mode, but instead of replacing
actual characters in the file, you are replacing screen real estate, so that
characters further on in the file never appear to move.

So if you type a <Tab> it may replace several normal characters, and if you
type a letter on top of a <Tab> it may not replace anything at all, since the
<Tab> will still line up to the same place as before.

Typing a <NL> still doesn't cause characters later in the file to appear to
move. The rest of the current line will be replaced by the <NL> (that is,
they are deleted), and replacing continues on the next line. A new line is
NOT inserted unless you go past the end of the file.

Interesting effects are seen when using CTRL-T and CTRL-D. The characters
before the cursor are shifted sideways as normal, but characters later in the
line still remain still. CTRL-T will hide some of the old line under the
shifted characters, but CTRL-D will reveal them again.

As with Replace mode, using <BS> etc will bring back the characters that were replaced. This still works in conjunction with 'smartindent', CTRL-T and CTRL-D, 'expandtab', 'smarttab', 'softtabstop', etc.

In 'list' mode, Virtual Replace mode acts as if it was not in 'list' mode, unless "L" is in 'cpoptions'.

Note that the only situations for which characters beyond the cursor should appear to move are in List mode |'list'|, and occasionally when 'wrap' is set (and the line changes length to become shorter or wider than the width of the screen). In other cases spaces may be inserted to avoid following characters to move.

This mode is very useful for editing <Tab> separated columns in tables, for entering new data while keeping all the columns aligned.

==
7. Insert mode completion *ins-completion*

In Insert and Replace mode, there are several commands to complete part of a keyword or line that has been typed. This is useful if you are using complicated keywords (e.g., function names with capitals and underscores).

These commands are not available when the |+insert_expand| feature was disabled at compile time.

Completion can be done for:

1. Whole lines |i_CTRL-X_CTRL-L|
2. keywords in the current file |i_CTRL-X_CTRL-N|
3. keywords in 'dictionary' |i_CTRL-X_CTRL-K|
4. keywords in 'thesaurus', thesaurus-style |i_CTRL-X_CTRL-T|
5. keywords in the current and included files |i_CTRL-X_CTRL-I|
6. tags |i_CTRL-X_CTRL-]|
7. file names |i_CTRL-X_CTRL-F|
8. definitions or macros |i_CTRL-X_CTRL-D|
9. Vim command-line |i_CTRL-X_CTRL-V|
10. User defined completion |i_CTRL-X_CTRL-U|
11. omni completion |i_CTRL-X_CTRL-O|
12. Spelling suggestions |i_CTRL-X_s|
13. keywords in 'complete' |i_CTRL-N| |i_CTRL-P|

All these, except CTRL-N and CTRL-P, are done in CTRL-X mode. This is a sub-mode of Insert and Replace modes. You enter CTRL-X mode by typing CTRL-X and one of the CTRL-X commands. You exit CTRL-X mode by typing a key that is not a valid CTRL-X mode command. Valid keys are the CTRL-X command itself, CTRL-N (next), and CTRL-P (previous).

Also see the 'infercase' option if you want to adjust the case of the match.

complete_CTRL-E
When completion is active you can use CTRL-E to stop it and go back to the originally typed text. The CTRL-E will not be inserted.

complete_CTRL-Y
When the popup menu is displayed you can use CTRL-Y to stop completion and accept the currently selected entry. The CTRL-Y is not inserted. Typing a space, Enter, or some other unprintable character will leave completion mode and insert that typed character.

When the popup menu is displayed there are a few more special keys, see

|popupmenu-keys|.

Note: The keys that are valid in CTRL-X mode are not mapped. This allows for ":map ^F ^X^F" to work (where ^F is CTRL-F and ^X is CTRL-X). The key that ends CTRL-X mode (any key that is not a valid CTRL-X mode command) is mapped. Also, when doing completion with 'complete' mappings apply as usual.

Note: While completion is active Insert mode can't be used recursively. Mappings that somehow invoke ":normal i.." will generate an E523 error.

The following mappings are suggested to make typing the completion commands a bit easier (although they will hide other commands): >

```
    :inoremap ^] ^X^]
    :inoremap ^F ^X^F
    :inoremap ^D ^X^D
    :inoremap ^L ^X^L
```

As a special case, typing CTRL-R to perform register insertion (see |i_CTRL-R|) will not exit CTRL-X mode. This is primarily to allow the use of the '=' register to call some function to determine the next operation. If the contents of the register (or result of the '=' register evaluation) are not valid CTRL-X mode keys, then CTRL-X mode will be exited as if those keys had been typed.

For example, the following will map <Tab> to either actually insert a <Tab> if the current line is currently only whitespace, or start/continue a CTRL-N completion operation: >

```
        function! CleverTab()
           if strpart( getline('.'), 0, col('.')-1 ) =~ '^\s*$'
              return "\<Tab>"
           else
              return "\<C-N>"
           endif
        endfunction
        inoremap <Tab> <C-R>=CleverTab()<CR>
```

Completing whole lines *compl-whole-line*

i_CTRL-X_CTRL-L

CTRL-X CTRL-L Search backwards for a line that starts with the same characters as those in the current line before the cursor. Indent is ignored. The matching line is inserted in front of the cursor.
The 'complete' option is used to decide which buffers are searched for a match. Both loaded and unloaded buffers are used.

CTRL-L or
CTRL-P Search backwards for next matching line. This line replaces the previous matching line.

CTRL-N Search forward for next matching line. This line replaces the previous matching line.

CTRL-X CTRL-L After expanding a line you can additionally get the line next to it by typing CTRL-X CTRL-L again, unless a double CTRL-X is used. Only works for loaded buffers.

Completing keywords in current file *compl-current*

i_CTRL-X_CTRL-P
i_CTRL-X_CTRL-N

CTRL-X CTRL-N Search forwards for words that start with the keyword in front of the cursor. The found keyword is inserted in front of the cursor.

CTRL-X CTRL-P Search backwards for words that start with the keyword in front of the cursor. The found keyword is inserted in front of the cursor.

CTRL-N Search forward for next matching keyword. This keyword replaces the previous matching keyword.

CTRL-P Search backwards for next matching keyword. This keyword replaces the previous matching keyword.

CTRL-X CTRL-N or
CTRL-X CTRL-P Further use of CTRL-X CTRL-N or CTRL-X CTRL-P will copy the words following the previous expansion in other contexts unless a double CTRL-X is used.

If there is a keyword in front of the cursor (a name made out of alphabetic characters and characters in 'iskeyword'), it is used as the search pattern, with "\<" prepended (meaning: start of a word). Otherwise "\<\k\k" is used as search pattern (start of any keyword of at least two characters).

In Replace mode, the number of characters that are replaced depends on the length of the matched string. This works like typing the characters of the matched string in Replace mode.

If there is not a valid keyword character before the cursor, any keyword of at least two characters is matched.

```
e.g., to get:
    printf("(%g, %g, %g)", vector[0], vector[1], vector[2]);
just type:
    printf("(%g, %g, %g)", vector[0], ^P[1], ^P[2]);
```

The search wraps around the end of the file, the value of 'wrapscan' is not used here.

Multiple repeats of the same completion are skipped; thus a different match will be inserted at each CTRL-N and CTRL-P (unless there is only one matching keyword).

Single character matches are never included, as they usually just get in the way of what you were really after.

```
e.g., to get:
        printf("name = %s\n", name);
just type:
        printf("name = %s\n", n^P);
or even:
        printf("name = %s\n", ^P);
```

The 'n' in '\n' is skipped.

After expanding a word, you can use CTRL-X CTRL-P or CTRL-X CTRL-N to get the word following the expansion in other contexts. These sequences search for the text just expanded and further expand by getting an extra word. This is useful if you need to repeat a sequence of complicated words. Although CTRL-P and CTRL-N look just for strings of at least two characters, CTRL-X CTRL-P and CTRL-X CTRL-N can be used to expand words of just one character.

```
e.g., to get:
```

```
                M&eacute;xico
        you can type:
                M^N^P^X^P^X^P
CTRL-N starts the expansion and then CTRL-P takes back the single character
"M", the next two CTRL-X CTRL-P's get the words "&eacute" and ";xico".

If the previous expansion was split, because it got longer than 'textwidth',
then just the text in the current line will be used.

If the match found is at the end of a line, then the first word in the next
line will be inserted and the message "word from next line" displayed, if
this word is accepted the next CTRL-X CTRL-P or CTRL-X CTRL-N will search
for those lines starting with this word.


Completing keywords in 'dictionary'                     *compl-dictionary*

                                                        *i_CTRL-X_CTRL-K*
CTRL-X CTRL-K           Search the files given with the 'dictionary' option
                        for words that start with the keyword in front of the
                        cursor.  This is like CTRL-N, but only the dictionary
                        files are searched, not the current file.  The found
                        keyword is inserted in front of the cursor.  This
                        could potentially be pretty slow, since all matches
                        are found before the first match is used.  By default,
                        the 'dictionary' option is empty.
                        For suggestions where to find a list of words, see the
                        'dictionary' option.

        CTRL-K  or
        CTRL-N          Search forward for next matching keyword.  This
                        keyword replaces the previous matching keyword.

        CTRL-P          Search backwards for next matching keyword.  This
                        keyword replaces the previous matching keyword.

                                                        *i_CTRL-X_CTRL-T*
CTRL-X CTRL-T           Works as CTRL-X CTRL-K, but in a special way.  It uses
                        the 'thesaurus' option instead of 'dictionary'.  If a
                        match is found in the thesaurus file, all the
                        remaining words on the same line are included as
                        matches, even though they don't complete the word.
                        Thus a word can be completely replaced.

                        For an example, imagine the 'thesaurus' file has a
                        line like this: >
                                angry furious mad enraged
<                       Placing the cursor after the letters "ang" and typing
                        CTRL-X CTRL-T would complete the word "angry";
                        subsequent presses would change the word to "furious",
                        "mad" etc.
                        Other uses include translation between two languages,
                        or grouping API functions by keyword.

        CTRL-T  or
        CTRL-N          Search forward for next matching keyword.  This
                        keyword replaces the previous matching keyword.

        CTRL-P          Search backwards for next matching keyword.  This
                        keyword replaces the previous matching keyword.
```

```
Completing keywords in the current and included files   *compl-keyword*

The 'include' option is used to specify a line that contains an include file
name.  The 'path' option is used to search for include files.

                                                        *i_CTRL-X_CTRL-I*
CTRL-X CTRL-I           Search for the first keyword in the current and
                        included files that starts with the same characters
                        as those before the cursor.  The matched keyword is
                        inserted in front of the cursor.

        CTRL-N          Search forwards for next matching keyword.  This
                        keyword replaces the previous matching keyword.
                        Note: CTRL-I is the same as <Tab>, which is likely to
                        be typed after a successful completion, therefore
                        CTRL-I is not used for searching for the next match.

        CTRL-P          Search backward for previous matching keyword.  This
                        keyword replaces the previous matching keyword.

        CTRL-X CTRL-I   Further use of CTRL-X CTRL-I will copy the words
                        following the previous expansion in other contexts
                        unless a double CTRL-X is used.

Completing tags                                         *compl-tag*
                                                        *i_CTRL-X_CTRL-]*
CTRL-X CTRL-]           Search for the first tag that starts with the same
                        characters as before the cursor.  The matching tag is
                        inserted in front of the cursor.  Alphabetic
                        characters and characters in 'iskeyword' are used
                        to decide which characters are included in the tag
                        name (same as for a keyword).  See also |CTRL-]|.
                        The 'showfulltag' option can be used to add context
                        from around the tag definition.
        CTRL-]  or
        CTRL-N          Search forwards for next matching tag.  This tag
                        replaces the previous matching tag.

        CTRL-P          Search backward for previous matching tag.  This tag
                        replaces the previous matching tag.


Completing file names                                   *compl-filename*
                                                        *i_CTRL-X_CTRL-F*
CTRL-X CTRL-F           Search for the first file name that starts with the
                        same characters as before the cursor.  The matching
                        file name is inserted in front of the cursor.
                        Alphabetic characters and characters in 'isfname'
                        are used to decide which characters are included in
                        the file name.  Note: the 'path' option is not used
                        here (yet).
        CTRL-F  or
        CTRL-N          Search forwards for next matching file name.  This
                        file name replaces the previous matching file name.

        CTRL-P          Search backward for previous matching file name.
                        This file name replaces the previous matching file
                        name.


Completing definitions or macros                        *compl-define*
```

The 'define' option is used to specify a line that contains a definition.
The 'include' option is used to specify a line that contains an include file
name. The 'path' option is used to search for include files.

i_CTRL-X_CTRL-D

CTRL-X CTRL-D Search in the current and included files for the first definition (or macro) name that starts with the same characters as before the cursor. The found definition name is inserted in front of the cursor.

CTRL-D or
CTRL-N Search forwards for next matching macro name. This macro name replaces the previous matching macro name.

CTRL-P Search backward for previous matching macro name. This macro name replaces the previous matching macro name.

CTRL-X CTRL-D Further use of CTRL-X CTRL-D will copy the words following the previous expansion in other contexts unless a double CTRL-X is used.

Completing Vim commands *compl-vim*

Completion is context-sensitive. It works like on the Command-line. It completes an Ex command as well as its arguments. This is useful when writing a Vim script.

i_CTRL-X_CTRL-V

CTRL-X CTRL-V Guess what kind of item is in front of the cursor and find the first match for it.
Note: When CTRL-V is mapped you can often use CTRL-Q instead of |i_CTRL-Q|.

CTRL-V or
CTRL-N Search forwards for next match. This match replaces the previous one.

CTRL-P Search backwards for previous match. This match replaces the previous one.

CTRL-X CTRL-V Further use of CTRL-X CTRL-V will do the same as CTRL-V. This allows mapping a key to do Vim command completion, for example: >

```
:imap <Tab> <C-X><C-V>
```

User defined completion *compl-function*

Completion is done by a function that can be defined by the user with the 'completefunc' option. See below for how the function is called and an example |complete-functions|.

i_CTRL-X_CTRL-U

CTRL-X CTRL-U Guess what kind of item is in front of the cursor and find the first match for it.

CTRL-U or
CTRL-N Use the next match. This match replaces the previous one.

CTRL-P Use the previous match. This match replaces the previous one.

Omni completion *compl-omni*

Completion is done by a function that can be defined by the user with the 'omnifunc' option. This is to be used for filetype-specific completion.

See below for how the function is called and an example |complete-functions|. For remarks about specific filetypes see |compl-omni-filetypes|. More completion scripts will appear, check www.vim.org. Currently there is a first version for C++.

i_CTRL-X_CTRL-O

CTRL-X CTRL-O Guess what kind of item is in front of the cursor and find the first match for it.

CTRL-O or
CTRL-N Use the next match. This match replaces the previous one.

CTRL-P Use the previous match. This match replaces the previous one.

Spelling suggestions *compl-spelling*

A word before or at the cursor is located and correctly spelled words are suggested to replace it. If there is a badly spelled word in the line, before or under the cursor, the cursor is moved to after it. Otherwise the word just before the cursor is used for suggestions, even though it isn't badly spelled.

NOTE: CTRL-S suspends display in many Unix terminals. Use 's' instead. Type CTRL-Q to resume displaying.

i_CTRL-X_CTRL-S *i_CTRL-X_s*

CTRL-X CTRL-S or
CTRL-X s Locate the word in front of the cursor and find the first spell suggestion for it.

CTRL-S or
CTRL-N Use the next suggestion. This replaces the previous one. Note that you can't use 's' here.

CTRL-P Use the previous suggestion. This replaces the previous one.

Completing keywords from different sources *compl-generic*

i_CTRL-N

CTRL-N Find next match for words that start with the keyword in front of the cursor, looking in places specified with the 'complete' option. The found keyword is inserted in front of the cursor.

i_CTRL-P

CTRL-P Find previous match for words that start with the keyword in front of the cursor, looking in places specified with the 'complete' option. The found keyword is inserted in front of the cursor.

CTRL-N Search forward for next matching keyword. This keyword replaces the previous matching keyword.

CTRL-P Search backwards for next matching keyword. This

keyword replaces the previous matching keyword.

CTRL-X CTRL-N or
CTRL-X CTRL-P Further use of CTRL-X CTRL-N or CTRL-X CTRL-P will copy the words following the previous expansion in other contexts unless a double CTRL-X is used.

FUNCTIONS FOR FINDING COMPLETIONS *complete-functions*

This applies to 'completefunc' and 'omnifunc'.

The function is called in two different ways:
- First the function is called to find the start of the text to be completed.
- Later the function is called to actually find the matches.

On the first invocation the arguments are:
a:findstart 1
a:base empty

The function must return the column where the completion starts. It must be a number between zero and the cursor column "col('.')". This involves looking at the characters just before the cursor and including those characters that could be part of the completed item. The text between this column and the cursor column will be replaced with the matches.

Special return values:
-1 If no completion can be done, the completion will be cancelled with an error message.
-2 To cancel silently and stay in completion mode.
-3 To cancel silently and leave completion mode.

On the second invocation the arguments are:
a:findstart 0
a:base the text with which matches should match; the text that was located in the first call (can be empty)

The function must return a List with the matching words. These matches usually include the "a:base" text. When there are no matches return an empty List.

In order to return more information than the matching words, return a Dict that contains the List. The Dict can have these items:
words The List of matching words (mandatory).
refresh A string to control re-invocation of the function (optional).
The only value currently recognized is "always", the effect is that the function is called whenever the leading text is changed.
Other items are ignored.

For acting upon end of completion, see the |CompleteDone| autocommand event.

For example, the function can contain this: >
let matches = ... list of words ...
return {'words': matches, 'refresh': 'always'}
<
complete-items
Each list item can either be a string or a Dictionary. When it is a string it is used as the completion. When it is a Dictionary it can contain these items:
word the text that will be inserted, mandatory

```
        abbr            abbreviation of "word"; when not empty it is used in
                        the menu instead of "word"
        menu            extra text for the popup menu, displayed after "word"
                        or "abbr"
        info            more information about the item, can be displayed in a
                        preview window
        kind            single letter indicating the type of completion
        icase           when non-zero case is to be ignored when comparing
                        items to be equal; when omitted zero is used, thus
                        items that only differ in case are added
        dup             when non-zero this match will be added even when an
                        item with the same word is already present.
        empty           when non-zero this match will be added even when it is
                        an empty string

All of these except "icase", "dup" and "empty" must be a string.  If an item
does not meet these requirements then an error message is given and further
items in the list are not used.  You can mix string and Dictionary items in
the returned list.

The "menu" item is used in the popup menu and may be truncated, thus it should
be relatively short.  The "info" item can be longer, it will  be displayed in
the preview window when "preview" appears in 'completeopt'.  The "info" item
will also remain displayed after the popup menu has been removed.  This is
useful for function arguments.  Use a single space for "info" to remove
existing text in the preview window.  The size of the preview window is three
lines, but 'previewheight' is used when it has a value of 1 or 2.

The "kind" item uses a single letter to indicate the kind of completion.  This
may be used to show the completion differently (different color or icon).
Currently these types can be used:
        v       variable
        f       function or method
        m       member of a struct or class
        t       typedef
        d       #define or macro

When searching for matches takes some time call |complete_add()| to add each
match to the total list.  These matches should then not appear in the returned
list!  Call |complete_check()| now and then to allow the user to press a key
while still searching for matches.  Stop searching when it returns non-zero.

                                                        *E839* *E840*
The function is allowed to move the cursor, it is restored afterwards.
The function is not allowed to move to another window or delete text.

An example that completes the names of the months: >
        fun! CompleteMonths(findstart, base)
          if a:findstart
            " locate the start of the word
            let line = getline('.')
            let start = col('.') - 1
            while start > 0 && line[start - 1] =~ '\a'
              let start -= 1
            endwhile
            return start
          else
            " find months matching with "a:base"
            let res = []
            for m in split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec")
              if m =~ '^' . a:base
                call add(res, m)
```

```
              endif
            endfor
            return res
          endif
        endfun
        set completefunc=CompleteMonths
<
The same, but now pretending searching for matches is slow: >
        fun! CompleteMonths(findstart, base)
          if a:findstart
            " locate the start of the word
            let line = getline('.')
            let start = col('.') - 1
            while start > 0 && line[start - 1] =~ '\a'
              let start -= 1
            endwhile
            return start
          else
            " find months matching with "a:base"
            for m in split("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec")
              if m =~ '^' . a:base
                call complete_add(m)
              endif
              sleep 300m        " simulate searching for next match
              if complete_check()
                break
              endif
            endfor
            return []
          endif
        endfun
        set completefunc=CompleteMonths
<
```

INSERT COMPLETION POPUP MENU *ins-completion-menu* *popupmenu-completion*

Vim can display the matches in a simplistic popup menu.

The menu is used when:
- The 'completeopt' option contains "menu" or "menuone".
- The terminal supports at least 8 colors.
- There are at least two matches. One if "menuone" is used.

The 'pumheight' option can be used to set a maximum height. The default is to use all space available.

There are three states:
1. A complete match has been inserted, e.g., after using CTRL-N or CTRL-P.
2. A cursor key has been used to select another match. The match was not inserted then, only the entry in the popup menu is highlighted.
3. Only part of a match has been inserted and characters were typed or the backspace key was used. The list of matches was then adjusted for what is in front of the cursor.

You normally start in the first state, with the first match being inserted. When "longest" is in 'completeopt' and there is more than one match you start in the third state.

If you select another match, e.g., with CTRL-N or CTRL-P, you go to the first state. This doesn't change the list of matches.

When you are back at the original text then you are in the third state. To

```
get there right away you can use a mapping that uses CTRL-P right after
starting the completion: >
        :imap <F7> <C-N><C-P>
<
                                                    *popupmenu-keys*
In the first state these keys have a special meaning:
<BS> and CTRL-H   Delete one character, find the matches for the word before
                  the cursor.  This reduces the list of matches, often to one
                  entry, and switches to the second state.
Any non-special character:
                  Stop completion without changing the match and insert the
                  typed character.

In the second and third state these keys have a special meaning:
<BS> and CTRL-H   Delete one character, find the matches for the shorter word
                  before the cursor.  This may find more matches.
CTRL-L            Add one character from the current match, may reduce the
                  number of matches.
any printable, non-white character:
                  Add this character and reduce the number of matches.

In all three states these can be used:
CTRL-Y            Yes: Accept the currently selected match and stop completion.
CTRL-E            End completion, go back to what was there before selecting a
                  match (what was typed or longest common string).
<PageUp>          Select a match several entries back, but don't insert it.
<PageDown>        Select a match several entries further, but don't insert it.
<Up>              Select the previous match, as if CTRL-P was used, but don't
                  insert it.
<Down>            Select the next match, as if CTRL-N was used, but don't
                  insert it.
<Space> or <Tab>  Stop completion without changing the match and insert the
                  typed character.

The behavior of the <Enter> key depends on the state you are in:
first state:      Use the text as it is and insert a line break.
second state:     Insert the currently selected match.
third state:      Use the text as it is and insert a line break.

In other words: If you used the cursor keys to select another entry in the
list of matches then the <Enter> key inserts that match.  If you typed
something else then <Enter> inserts a line break.


The colors of the menu can be changed with these highlight groups:
Pmenu           normal item  |hl-Pmenu|
PmenuSel        selected item  |hl-PmenuSel|
PmenuSbar       scrollbar  |hl-PmenuSbar|
PmenuThumb      thumb of the scrollbar  |hl-PmenuThumb|

There are no special mappings for when the popup menu is visible.  However,
you can use an Insert mode mapping that checks the |pumvisible()| function to
do something different.  Example: >
        :inoremap <Down> <C-R>=pumvisible() ? "\<lt>C-N>" : "\<lt>Down>"<CR>

You can use of <expr> in mapping to have the popup menu used when typing a
character and some condition is met.  For example, for typing a dot: >
        inoremap <expr> . MayComplete()
        func MayComplete()
            if (can complete)
              return ".\<C-X>\<C-O>"
            endif
```

```
            return '.'
        endfunc

See |:map-<expr>| for more info.


FILETYPE-SPECIFIC REMARKS FOR OMNI COMPLETION       *compl-omni-filetypes*

The file used for {filetype} should be autoload/{filetype}complete.vim
in 'runtimepath'.  Thus for "java" it is autoload/javacomplete.vim.


C                                                           *ft-c-omni*

Completion of C code requires a tags file.  You should use Exuberant ctags,
because it adds extra information that is needed for completion.  You can find
it here: http://ctags.sourceforge.net/  Version 5.6 or later is recommended.

For version 5.5.4 you should add a patch that adds the "typename:" field:
        ftp://ftp.vim.org/pub/vim/unstable/patches/ctags-5.5.4.patch
A compiled .exe for MS-Windows can be found at:
        http://georgevreilly.com/vim/ctags.html

If you want to complete system functions you can do something like this.  Use
ctags to generate a tags file for all the system header files: >
        % ctags -R -f ~/.vim/systags /usr/include /usr/local/include
In your vimrc file add this tags file to the 'tags' option: >
        set tags+=~/.vim/systags

When using CTRL-X CTRL-O after a name without any "." or "->" it is completed
from the tags file directly.  This works for any identifier, also function
names.  If you want to complete a local variable name, which does not appear
in the tags file, use CTRL-P instead.

When using CTRL-X CTRL-O after something that has "." or "->" Vim will attempt
to recognize the type of the variable and figure out what members it has.
This means only members valid for the variable will be listed.

When a member name already was complete, CTRL-X CTRL-O will add a "." or
"->" for composite types.

Vim doesn't include a C compiler, only the most obviously formatted
declarations are recognized.  Preprocessor stuff may cause confusion.
When the same structure name appears in multiple places all possible members
are included.


CSS                                                         *ft-css-omni*

Complete properties and their appropriate values according to CSS 2.1
specification.


HTML                                                        *ft-html-omni*
XHTML                                                       *ft-xhtml-omni*

CTRL-X CTRL-O provides completion of various elements of (X)HTML files.  It is
designed to support writing of XHTML 1.0 Strict files but will also work for
other versions of HTML. Features:

- after "<" complete tag name depending on context (no div suggestion inside
  of an a tag); '/>' indicates empty tags
```

- inside of tag complete proper attributes (no width attribute for an a tag);
  show also type of attribute; '*' indicates required attributes
- when attribute has limited number of possible values help to complete them
- complete names of entities
- complete values of "class" and "id" attributes with data obtained from
  <style> tag and included CSS files
- when completing value of "style" attribute or working inside of "style" tag
  switch to |ft-css-omni| completion
- when completing values of events attributes or working inside of "script"
  tag switch to |ft-javascript-omni| completion
- when used after "</" CTRL-X CTRL-O will close the last opened tag

Note: When used first time completion menu will be shown with little delay
- this is time needed for loading of data file.
Note: Completion may fail in badly formatted documents. In such case try to
run |:make| command to detect formatting problems.

HTML flavor *html-flavor*

The default HTML completion depends on the filetype. For HTML files it is
HTML 4.01 Transitional ('filetype' is "html"), for XHTML it is XHTML 1.0
Strict ('filetype' is "xhtml").

When doing completion outside of any other tag you will have possibility to
choose DOCTYPE and the appropriate data file will be loaded and used for all
next completions.

More about format of data file in |xml-omni-datafile|. Some of the data files
may be found on the Vim website (|www|).

Note that b:html_omni_flavor may point to a file with any XML data. This
makes possible to mix PHP (|ft-php-omni|) completion with any XML dialect
(assuming you have data file for it). Without setting that variable XHTML 1.0
Strict will be used.

JAVASCRIPT *ft-javascript-omni*

Completion of most elements of JavaScript language and DOM elements.

Complete:

- variables
- function name; show function arguments
- function arguments
- properties of variables trying to detect type of variable
- complete DOM objects and properties depending on context
- keywords of language

Completion works in separate JavaScript files (&ft==javascript), inside of
<script> tag of (X)HTML and in values of event attributes (including scanning
of external files).

DOM compatibility

At the moment (beginning of 2006) there are two main browsers - MS Internet
Explorer and Mozilla Firefox. These two applications are covering over 90% of
market. Theoretically standards are created by W3C organisation
(http://www.w3c.org) but they are not always followed/implemented.

IE FF W3C Omni completion ~

```
               +/-     +/-     +    +              ~
               +       +       -    +              ~
               +       -       -    -              ~
               -       +       -    -              ~
```

Regardless from state of implementation in browsers but if element is defined
in standards, completion plugin will place element in suggestion list. When
both major engines implemented element, even if this is not in standards it
will be suggested. All other elements are not placed in suggestion list.

PHP *ft-php-omni*

Completion of PHP code requires a tags file for completion of data from
external files and for class aware completion. You should use Exuberant ctags
version 5.5.4 or newer. You can find it here: http://ctags.sourceforge.net/

Script completes:

- after $ variables name
  - if variable was declared as object add "->", if tags file is available show
    name of class
  - after "->" complete only function and variable names specific for given
    class. To find class location and contents tags file is required. Because
    PHP isn't strongly typed language user can use @var tag to declare class: >

```
        /* @var $myVar myClass */
        $myVar->
<
```

    Still, to find myClass contents tags file is required.

- function names with additional info:
  - in case of built-in functions list of possible arguments and after | type
    data returned by function
  - in case of user function arguments and name of file where function was
    defined (if it is not current file)

- constants names
- class names after "new" declaration

Note: when doing completion first time Vim will load all necessary data into
memory. It may take several seconds. After next use of completion delay
should not be noticeable.

Script detects if cursor is inside <?php ?> tags. If it is outside it will
automatically switch to HTML/CSS/JavaScript completion. Note: contrary to
original HTML files completion of tags (and only tags) isn't context aware.

RUBY *ft-ruby-omni*

Completion of Ruby code requires that vim be built with |+ruby|.

Ruby completion will parse your buffer on demand in order to provide a list of
completions. These completions will be drawn from modules loaded by 'require'
and modules defined in the current buffer.

The completions provided by CTRL-X CTRL-O are sensitive to the context:

```
          CONTEXT                         COMPLETIONS PROVIDED ~
```

```
 1. Not inside a class definition    Classes, constants and globals

 2. Inside a class definition        Methods or constants defined in the class

 3. After '.', '::' or ':'           Methods applicable to the object being
                                       dereferenced

 4. After ':' or ':foo'              Symbol name (beginning with 'foo')

Notes:
 - Vim will load/evaluate code in order to provide completions.  This may
   cause some code execution, which may be a concern. This is no longer
   enabled by default, to enable this feature add >
     let g:rubycomplete_buffer_loading = 1
<- In context 1 above, Vim can parse the entire buffer to add a list of
   classes to the completion results. This feature is turned off by default,
   to enable it add >
     let g:rubycomplete_classes_in_global = 1
<  to your vimrc
 - In context 2 above, anonymous classes are not supported.
 - In context 3 above, Vim will attempt to determine the methods supported by
   the object.
 - Vim can detect and load the Rails environment for files within a rails
   project. The feature is disabled by default, to enable it add >
     let g:rubycomplete_rails = 1
<  to your vimrc


SYNTAX                                                  *ft-syntax-omni*

Vim has the ability to color syntax highlight nearly 500 languages.  Part of
this highlighting includes knowing what keywords are part of a language.  Many
filetypes already have custom completion scripts written for them, the
syntaxcomplete plugin provides basic completion for all other filetypes.  It
does this by populating the omni completion list with the text Vim already
knows how to color highlight.  It can be used for any filetype and provides a
minimal language-sensitive completion.

To enable syntax code completion you can run: >
    setlocal omnifunc=syntaxcomplete#Complete

You can automate this by placing the following in your |.vimrc| (after any
":filetype" command): >
    if has("autocmd") && exists("+omnifunc")
        autocmd Filetype *
                    \   if &omnifunc == "" |
                    \           setlocal omnifunc=syntaxcomplete#Complete |
                    \   endif
    endif

The above will set completion to this script only if a specific plugin does
not already exist for that filetype.

Each filetype can have a wide range of syntax items.  The plugin allows you to
customize which syntax groups to include or exclude from the list.  Let's have
a look at the PHP filetype to see how this works.

If you edit a file called, index.php, run the following command: >
    syntax list

The first thing you will notice is that there are many different syntax groups.
The PHP language can include elements from different languages like HTML,
```

```
JavaScript and many more.  The syntax plugin will only include syntax groups
that begin with the filetype, "php", in this case.  For example these syntax
groups are included by default with the PHP: phpEnvVar, phpIntVar,
phpFunctions.

If you wish non-filetype syntax items to also be included, you can use a
regular expression syntax (added in version 13.0 of autoload\syntaxcomplete.vim)
to add items.  Looking at the output from ":syntax list" while editing a PHP file
I can see some of these entries: >
    htmlArg,htmlTag,htmlTagName,javaScriptStatement,javaScriptGlobalObjects

To pick up any JavaScript and HTML keyword syntax groups while editing a PHP
file, you can use 3 different regexs, one for each language.  Or you can
simply restrict the include groups to a particular value, without using
a regex string: >
    let g:omni_syntax_group_include_php = 'php\w\+,javaScript\w\+,html\w\+'
    let g:omni_syntax_group_include_php = 'phpFunctions,phpMethods'
<
The basic form of this variable is: >
    let g:omni_syntax_group_include_{filetype} = 'regex,comma,separated'

The PHP language has an enormous number of items which it knows how to syntax
highlight.  These items will be available within the omni completion list.

Some people may find this list unwieldy or are only interested in certain
items.  There are two ways to prune this list (if necessary).  If you find
certain syntax groups you do not wish displayed you can use two different
methods to identify these groups.  The first specifically lists the syntax
groups by name.  The second uses a regular expression to identify both
syntax groups.  Simply add one the following to your vimrc: >
    let g:omni_syntax_group_exclude_php = 'phpCoreConstant,phpConstant'
    let g:omni_syntax_group_exclude_php = 'php\w*Constant'

Add as many syntax groups to this list by comma separating them.  The basic
form of this variable is: >
    let g:omni_syntax_group_exclude_{filetype} = 'regex,comma,separated'

You can create as many of these variables as you need, varying only the
filetype at the end of the variable name.

The plugin uses the isKeyword option to determine where word boundaries are
for the syntax items.  For example, in the Scheme language completion should
include the "-", call-with-output-file.  Depending on your filetype, this may
not provide the words you are expecting.  Setting the
g:omni_syntax_use_iskeyword option to 0 will force the syntax plugin to break
on word characters.   This can be controlled adding the following to your
vimrc: >
    let g:omni_syntax_use_iskeyword = 0

For plugin developers, the plugin exposes a public function OmniSyntaxList.
This function can be used to request a List of syntax items.  When editing a
SQL file (:e syntax.sql) you can use the ":syntax list" command to see the
various groups and syntax items.  For example: >
    syntax list

Yields data similar to this: >
    sqlOperator    xxx some prior all like and any escape exists in is not
                       or intersect minus between distinct
                       links to Operator
    sqlType        xxx varbit varchar nvarchar bigint int uniqueidentifier
                       date money long tinyint unsigned xml text smalldate
                       double datetime nchar smallint numeric time bit char
```

```
                        varbinary binary smallmoney
                        image float integer timestamp real decimal

There are two syntax groups listed here: sqlOperator and sqlType.  To retrieve
a List of syntax items you can call OmniSyntaxList a number of different
ways.  To retrieve all syntax items regardless of syntax group:  >
    echo OmniSyntaxList( [] )

To retrieve only the syntax items for the sqlOperator syntax group: >
    echo OmniSyntaxList( ['sqlOperator'] )

To retrieve all syntax items for both the sqlOperator and sqlType groups: >
    echo OmniSyntaxList( ['sqlOperator', 'sqlType'] )

A regular expression can also be used: >
    echo OmniSyntaxList( ['sql\w\+'] )

From within a plugin, you would typically assign the output to a List: >
    let myKeywords = []
    let myKeywords = OmniSyntaxList( ['sqlKeyword'] )



SQL                                                     *ft-sql-omni*

Completion for the SQL language includes statements, functions, keywords.
It will also dynamically complete tables, procedures, views and column lists
with data pulled directly from within a database.  For detailed instructions
and a tutorial see |omni-sql-completion|.

The SQL completion plugin can be used in conjunction with other completion
plugins.  For example, the PHP filetype has its own completion plugin.
Since PHP is often used to generate dynamic website by accessing a database,
the SQL completion plugin can also be enabled.  This allows you to complete
PHP code and SQL code at the same time.


XML                                                     *ft-xml-omni*

Vim 7 provides a mechanism for context aware completion of XML files.  It
depends on a special |xml-omni-datafile| and two commands: |:XMLns| and
|:XMLent|.  Features are:

- after "<" complete the tag name, depending on context
- inside of a tag complete proper attributes
- when an attribute has a limited number of possible values help to complete
  them
- complete names of entities (defined in |xml-omni-datafile| and in the
  current file with "<!ENTITY" declarations)
- when used after "</" CTRL-X CTRL-O will close the last opened tag

Format of XML data file                                 *xml-omni-datafile*

XML data files are stored in the "autoload/xml" directory in 'runtimepath'.
Vim distribution provides examples of data files in the
"$VIMRUNTIME/autoload/xml" directory.  They have a meaningful name which will
be used in commands.  It should be a unique name which will not create
conflicts.  For example, the name xhtml10s.vim means it is the data file for
XHTML 1.0 Strict.

Each file contains a variable with a name like g:xmldata_xhtml10s . It is
a compound from two parts:
```

```
1. "g:xmldata_"  general prefix, constant for all data files
2. "xhtml10s"    the name of the file and the name of the described XML
                 dialect; it will be used as an argument for the |:XMLns|
                 command

Part two must be exactly the same as name of file.

The variable is a |Dictionary|.  Keys are tag names and each value is a two
element |List|.  The first element of the List is also a List with the names
of possible children.  The second element is a |Dictionary| with the names of
attributes as keys and the possible values of attributes as values.  Example: >

    let g:xmldata_crippled = {
    \ "vimxmlentities": ["amp", "lt", "gt", "apos", "quot"],
    \ 'vimxmlroot': ['tag1'],
    \ 'tag1':
    \ [ ['childoftag1a', 'childoftag1b'], {'attroftag1a': [],
    \ 'attroftag1b': ['valueofattr1', 'valueofattr2']}],
    \ 'childoftag1a':
    \ [ [], {'attrofchild': ['attrofchild']}],
    \ 'childoftag1b':
    \ [ ['childoftag1a'], {'attrofchild': []}],
    \ "vimxmltaginfo": {
    \ 'tag1': ['Menu info', 'Long information visible in preview window']},
    \ 'vimxmlattrinfo': {
    \ 'attrofchild': ['Menu info', 'Long information visible in preview window']}}

This example would be put in the "autoload/xml/crippled.vim" file and could
help to write this file: >

    <tag1 attroftag1b="valueofattr1">
        <childoftag1a attrofchild>
                &amp; &lt;
        </childoftag1a>
        <childoftag1b attrofchild="5">
            <childoftag1a>
                &gt; &apos; &quot;
            </childoftag1a>
        </childoftag1b>
    </tag1>

In the example four special elements are visible:

1. "vimxmlentities" - a special key with List containing entities of this XML
   dialect.
2. If the list containing possible values of attributes has one element and
   this element is equal to the name of the attribute this attribute will be
   treated as boolean and inserted as 'attrname' and not as 'attrname="'
3. "vimxmltaginfo" - a special key with a Dictionary containing tag
   names as keys and two element List as values, for additional menu info and
   the long description.
4. "vimxmlattrinfo" - special key with Dictionary containing attribute names
   as keys and two element List as values, for additional menu info and long
   description.

Note: Tag names in the data file MUST not contain a namespace description.
Check xsl.vim for an example.
Note: All data and functions are publicly available as global
variables/functions and can be used for personal editing functions.
```

DTD -> Vim *dtd2vim*

On |www| is the script |dtd2vim| which parses DTD and creates an XML data file
for Vim XML omni completion.

    dtd2vim: http://www.vim.org/scripts/script.php?script_id=1462

Check the beginning of that file for usage details.
The script requires perl and:

    perlSGML: http://savannah.nongnu.org/projects/perlsgml

Commands

:XMLns {name} [{namespace}] *:XMLns*

Vim has to know which data file should be used and with which namespace. For
loading of the data file and connecting data with the proper namespace use
|:XMLns| command. The first (obligatory) argument is the name of the data
(xhtml10s, xsl). The second argument is the code of namespace (h, xsl). When
used without a second argument the dialect will be used as default - without
namespace declaration. For example to use XML completion in .xsl files: >

        :XMLns xhtml10s
        :XMLns xsl xsl

:XMLent {name} *:XMLent*

By default entities will be completed from the data file of the default
namespace. The XMLent command should be used in case when there is no default
namespace: >

        :XMLent xhtml10s

Usage

While used in this situation (after declarations from previous part, | is
cursor position): >

        <|

Will complete to an appropriate XHTML tag, and in this situation: >

        <xsl:|

Will complete to an appropriate XSL tag.

The script xmlcomplete.vim, provided through the |autoload| mechanism,
has the xmlcomplete#GetLastOpenTag() function which can be used in XML files
to get the name of the last open tag (b:unaryTagsStack has to be defined): >

        :echo xmlcomplete#GetLastOpenTag("b:unaryTagsStack")

==
8. Insert mode commands *inserting*

The following commands can be used to insert new text into the buffer. They

can all be undone and repeated with the "." command.

a
a Append text after the cursor [count] times. If the cursor is in the first column of an empty line Insert starts there. But not when 'virtualedit' is set!

A
A Append text at the end of the line [count] times.

<insert> or *i* *insert* *<Insert>*
i Insert text before the cursor [count] times. When using CTRL-O in Insert mode |i_CTRL-O| the count is not supported.

I
I Insert text before the first non-blank in the line [count] times.
When the 'H' flag is present in 'cpoptions' and the line only contains blanks, insert start just before the last blank.

gI
gI Insert text in column 1 [count] times. {not in Vi}

gi
gi Insert text in the same position as where Insert mode was stopped last time in the current buffer.
This uses the |'^| mark. It's different from "`^i" when the mark is past the end of the line.
The position is corrected for inserted/deleted lines, but NOT for inserted/deleted characters.
When the |:keepjumps| command modifier is used the |'^| mark won't be changed.
{not in Vi}

o
o Begin a new line below the cursor and insert text, repeat [count] times. {Vi: blank [count] screen lines}
When the '#' flag is in 'cpoptions' the count is ignored.

O
O Begin a new line above the cursor and insert text, repeat [count] times. {Vi: blank [count] screen lines}
When the '#' flag is in 'cpoptions' the count is ignored.

These commands are used to start inserting text. You can end insert mode with <Esc>. See |mode-ins-repl| for the other special characters in Insert mode. The effect of [count] takes place after Insert mode is exited.

When 'autoindent' is on, the indent for a new line is obtained from the previous line. When 'smartindent' or 'cindent' is on, the indent for a line is automatically adjusted for C programs.

'textwidth' can be set to the maximum width for a line. When a line becomes too long when appending characters a line break is automatically inserted.

```
==============================================================================
9. Ex insert commands                                   *inserting-ex*

                                                        *:a* *:append*
:{range}a[ppend][!]     Insert several lines of text below the specified
                        line.  If the {range} is missing, the text will be
                        inserted after the current line.
                        Adding [!] toggles 'autoindent' for the time this
                        command is executed.

                                                        *:i* *:in* *:insert*
:{range}i[nsert][!]     Insert several lines of text above the specified
                        line.  If the {range} is missing, the text will be
                        inserted before the current line.
                        Adding [!] toggles 'autoindent' for the time this
                        command is executed.

These two commands will keep on asking for lines, until you type a line
containing only a ".".  Watch out for lines starting with a backslash, see
|line-continuation|.

When in Ex mode (see |-e|) a backslash at the end of the line can be used to
insert a NUL character.  To be able to have a line ending in a backslash use
two backslashes.  This means that the number of backslashes is halved, but
only at the end of the line.

NOTE: These commands cannot be used with |:global| or |:vglobal|.
":append" and ":insert" don't work properly in between ":if" and
":endif", ":for" and ":endfor", ":while" and ":endwhile".

                                                        *:start* *:startinsert*
:star[tinsert][!]       Start Insert mode just after executing this command.
                        Works like typing "i" in Normal mode.  When the ! is
                        included it works like "A", append to the line.
                        Otherwise insertion starts at the cursor position.
                        Note that when using this command in a function or
                        script, the insertion only starts after the function
                        or script is finished.
                        This command does not work from |:normal|.
                        {not in Vi}

                                                        *:stopi* *:stopinsert*
:stopi[nsert]           Stop Insert mode as soon as possible.  Works like
                        typing <Esc> in Insert mode.
                        Can be used in an autocommand, example: >
                                :au BufEnter scratch stopinsert
<
                                        *replacing-ex* *:startreplace*
:startr[eplace][!]      Start Replace mode just after executing this command.
                        Works just like typing "R" in Normal mode.  When the
                        ! is included it acts just like "$R" had been typed
                        (ie. begin replace mode at the end-of-line).  Other-
                        wise replacement begins at the cursor position.
                        Note that when using this command in a function or
                        script that the replacement will only start after
                        the function or script is finished.
                        {not in Vi}

                                                        *:startgreplace*
:startg[replace][!]     Just like |:startreplace|, but use Virtual Replace
                        mode, like with |gR|.
                        {not in Vi}
```

```
==============================================================================
10. Inserting a file                                    *inserting-file*

                                                        *:r* *:re* *:read*
:r[ead] [++opt] [name]
                        Insert the file [name] (default: current file) below
                        the cursor.
                        See |++opt| for the possible values of [++opt].

:{range}r[ead] [++opt] [name]
                        Insert the file [name] (default: current file) below
                        the specified line.
                        See |++opt| for the possible values of [++opt].

                                                        *:r!* *:read!*
:[range]r[ead] [++opt] !{cmd}
                        Execute {cmd} and insert its standard output below
                        the cursor or the specified line.  A temporary file is
                        used to store the output of the command which is then
                        read into the buffer.  'shellredir' is used to save
                        the output of the command, which can be set to include
                        stderr or not.  {cmd} is executed like with ":!{cmd}",
                        any '!' is replaced with the previous command |:!|.
                        See |++opt| for the possible values of [++opt].

These commands insert the contents of a file, or the output of a command,
into the buffer.  They can be undone.  They cannot be repeated with the "."
command.  They work on a line basis, insertion starts below the line in which
the cursor is, or below the specified line.  To insert text above the first
line use the command ":0r {name}".

After the ":read" command, the cursor is left on the first non-blank in the
first new line.  Unless in Ex mode, then the cursor is left on the last new
line (sorry, this is Vi compatible).

If a file name is given with ":r", it becomes the alternate file.  This can be
used, for example, when you want to edit that file instead: ":e! #".  This can
be switched off by removing the 'a' flag from the 'cpoptions' option.

Of the [++opt] arguments one is specifically for ":read", the ++edit argument.
This is useful when the ":read" command is actually used to read a file into
the buffer as if editing that file.  Use this command in an empty buffer: >
        :read ++edit filename
The effect is that the 'fileformat', 'fileencoding', 'bomb', etc. options are
set to what has been detected for "filename".  Note that a single empty line
remains, you may want to delete it.

                                                        *file-read*
The 'fileformat' option sets the <EOL> style for a file:
'fileformat'    characters         name                         ~
  "dos"         <CR><NL> or <NL>   DOS format
  "unix"        <NL>               Unix format
  "mac"         <CR>               Mac format
Previously 'textmode' was used.  It is obsolete now.

If 'fileformat' is "dos", a <CR> in front of an <NL> is ignored and a CTRL-Z
at the end of the file is ignored.

If 'fileformat' is "mac", a <NL> in the file is internally represented by a
<CR>.  This is to avoid confusion with a <NL> which is used to represent a
<NUL>.  See |CR-used-for-NL|.
```

```
If the 'fileformats' option is not empty Vim tries to recognize the type of
<EOL> (see |file-formats|).  However, the 'fileformat' option will not be
changed, the detected format is only used while reading the file.
A similar thing happens with 'fileencodings'.

On non-MS-DOS, Win32, and OS/2 systems the message "[dos format]" is shown if
a file is read in DOS format, to remind you that something unusual is done.
On Macintosh, MS-DOS, Win32, and OS/2 the message "[unix format]" is shown if
a file is read in Unix format.
On non-Macintosh systems, the message "[Mac format]" is shown if a file is
read in Mac format.

An example on how to use ":r !": >
        :r !uuencode binfile binfile
This command reads "binfile", uuencodes it and reads it into the current
buffer.  Useful when you are editing e-mail and want to include a binary
file.

                                                        *read-messages*
When reading a file Vim will display a message with information about the read
file.  In the table is an explanation for some of the items.  The others are
self explanatory.  Using the long or the short version depends on the
'shortmess' option.

        long            short           meaning ~
        [readonly]      {RO}            the file is write protected
        [fifo/socket]                   using a stream
        [fifo]                          using a fifo stream
        [socket]                        using a socket stream
        [CR missing]                    reading with "dos" 'fileformat' and a
                                        NL without a preceding CR was found.
        [NL found]                      reading with "mac" 'fileformat' and a
                                        NL was found (could be "unix" format)
        [long lines split]              at least one line was split in two
        [NOT converted]                 conversion from 'fileencoding' to
                                        'encoding' was desired but not
                                        possible
        [converted]                     conversion from 'fileencoding' to
                                        'encoding' done
        [crypted]                       file was decrypted
        [READ ERRORS]                   not all of the file could be read


 vim:tw=78:ts=8:ft=help:norl:
*change.txt*    For Vim version 8.0.  Last change: 2017 Feb 12


                  VIM REFERENCE MANUAL    by Bram Moolenaar


This file describes commands that delete or change text.  In this context,
changing text means deleting the text and replacing it with other text using
one command.  You can undo all of these commands.  You can repeat the non-Ex
commands with the "." command.

1. Deleting text                |deleting|
2. Delete and insert            |delete-insert|
3. Simple changes               |simple-change|         *changing*
4. Complex changes              |complex-change|
   4.1 Filter commands             |filter|
   4.2 Substitute                  |:substitute|
```

For inserting text see |insert.txt|.

==
1. Deleting text   *deleting* *E470*

["x]<Del>   or   *<Del>* *x* *dl*
["x]x   Delete [count] characters under and after the cursor
                        [into register x] (not |linewise|). Does the same as
                        "dl".
                        The <Del> key does not take a [count]. Instead, it
                        deletes the last character of the count.
                        See |:fixdel| if the <Del> key does not do what you
                        want. See |'whichwrap'| for deleting a line break
                        (join lines). {Vi does not support <Del>}

                                                        X *dh*
["x]X   Delete [count] characters before the cursor [into
                        register x] (not |linewise|). Does the same as "dh".
                        Also see |'whichwrap'|.

                                                        d
["x]d{motion}   Delete text that {motion} moves over [into register
                        x]. See below for exceptions.

                                                        dd
["x]dd   Delete [count] lines [into register x] |linewise|.

                                                        D
["x]D   Delete the characters under the cursor until the end
                        of the line and [count]-1 more lines [into register
                        x]; synonym for "d$".
                        (not |linewise|)
                        When the '#' flag is in 'cpoptions' the count is
                        ignored.

{Visual}["x]x   or   *v_x* *v_d* *v_<Del>*
{Visual}["x]d   or
{Visual}["x]<Del>   Delete the highlighted text [into register x] (for
                        {Visual} see |Visual-mode|). {not in Vi}

{Visual}["x]CTRL-H   or   *v_CTRL-H* *v_<BS>*
{Visual}["x]<BS>   When in Select mode: Delete the highlighted text [into
                        register x].

{Visual}["x]X   or   *v_X* *v_D* *v_b_D*
{Visual}["x]D   Delete the highlighted lines [into register x] (for
                        {Visual} see |Visual-mode|). In Visual block mode,
                        "D" deletes the highlighted text plus all text until
                        the end of the line. {not in Vi}

                                        :d *:de* *:del* *:delete* *:dl* *:dp*
:[range]d[elete] [x]   Delete [range] lines (default: current line) [into
                        register x].
                        Note these weird abbreviations:
                           :dl   delete and list
                           :dell   idem

```
                           :delel       idem
                           :deletl      idem
                           :deletel     idem
                           :dp          delete and print
                           :dep         idem
                           :delp        idem
                           :delep       idem
                           :deletp      idem
                           :deletep     idem

:[range]d[elete] [x] {count}
                        Delete {count} lines, starting with [range]
                        (default: current line |cmdline-ranges|) [into
                        register x].

These commands delete text.  You can repeat them with the `.` command
(except `:d`) and undo them.  Use Visual mode to delete blocks of text.  See
|registers| for an explanation of registers.

An exception for the d{motion} command: If the motion is not linewise, the
start and end of the motion are not in the same line, and there are only
blanks before the start and there are no non-blanks after the end of the
motion, the delete becomes linewise.  This means that the delete also removes
the line of blanks that you might expect to remain. Use the |o_v| operator to
force the motion to be characterwise.

Trying to delete an empty region of text (e.g., "d0" in the first column)
is an error when 'cpoptions' includes the 'E' flag.

                                                        *J*
J                       Join [count] lines, with a minimum of two lines.
                        Remove the indent and insert up to two spaces (see
                        below).  Fails when on the last line of the buffer.
                        If [count] is too big it is reduce to the number of
                        lines available.

                                                        *v_J*
{Visual}J               Join the highlighted lines, with a minimum of two
                        lines.  Remove the indent and insert up to two spaces
                        (see below).  {not in Vi}

                                                        *gJ*
gJ                      Join [count] lines, with a minimum of two lines.
                        Don't insert or remove any spaces.  {not in Vi}

                                                        *v_gJ*
{Visual}gJ              Join the highlighted lines, with a minimum of two
                        lines.  Don't insert or remove any spaces.  {not in
                        Vi}

                                                        *:j* *:join*
:[range]j[oin][!] [flags]
                        Join [range] lines.  Same as "J", except with [!]
                        the join does not insert or delete any spaces.
                        If a [range] has equal start and end values, this
                        command does nothing.  The default behavior is to
                        join the current line with the line below it.
                        {not in Vi: !}
                        See |ex-flags| for [flags].

:[range]j[oin][!] {count} [flags]
                        Join {count} lines, starting with [range] (default:
```

current line |cmdline-ranges|). Same as "J", except
with [!] the join does not insert or delete any
spaces.
{not in Vi: !}
See |ex-flags| for [flags].

These commands delete the <EOL> between lines. This has the effect of joining multiple lines into one line. You can repeat these commands (except `:j`) and undo them.

These commands, except "gJ", insert one space in place of the <EOL> unless there is trailing white space or the next line starts with a ')'. These commands, except "gJ", delete any leading white space on the next line. If the 'joinspaces' option is on, these commands insert two spaces after a '.', '!' or '?' (but if 'cpoptions' includes the 'j' flag, they insert two spaces only after a '.').
The 'B' and 'M' flags in 'formatoptions' change the behavior for inserting spaces before and after a multi-byte character |fo-table|.

The '[mark is set at the end of the first line that was joined, '] at the end of the resulting line.

==
2. Delete and insert *delete-insert* *replacing*

R

R Enter Replace mode: Each character you type replaces an existing character, starting with the character under the cursor. Repeat the entered text [count]-1 times. See |Replace-mode| for more details.

gR

gR Enter Virtual Replace mode: Each character you type replaces existing characters in screen space. So a <Tab> may replace several characters at once. Repeat the entered text [count]-1 times. See |Virtual-Replace-mode| for more details.
{not available when compiled without the |+vreplace| feature}

c

["x]c{motion} Delete {motion} text [into register x] and start insert. When 'cpoptions' includes the 'E' flag and there is no text to delete (e.g., with "cTx" when the cursor is just after an 'x'), an error occurs and insert mode does not start (this is Vi compatible). When 'cpoptions' does not include the 'E' flag, the "c" command always starts insert mode, even if there is no text to delete.

cc

["x]cc Delete [count] lines [into register x] and start insert |linewise|. If 'autoindent' is on, preserve the indent of the first line.

C

["x]C Delete from the cursor position to the end of the line and [count]-1 more lines [into register x], and start insert. Synonym for c$ (not |linewise|).

s

```
["x]s                   Delete [count] characters [into register x] and start
                        insert (s stands for Substitute).  Synonym for "cl"
                        (not |linewise|).

                                                        *S*
["x]S                   Delete [count] lines [into register x] and start
                        insert.  Synonym for "cc" |linewise|.

{Visual}["x]c   or                                      *v_c* *v_s*
{Visual}["x]s           Delete the highlighted text [into register x] and
                        start insert (for {Visual} see |Visual-mode|).  {not
                        in Vi}

                                                        *v_r*
{Visual}["x]r{char}     Replace all selected characters by {char}.

                                                        *v_C*
{Visual}["x]C           Delete the highlighted lines [into register x] and
                        start insert.  In Visual block mode it works
                        differently |v_b_C|.  {not in Vi}
                                                        *v_S*
{Visual}["x]S           Delete the highlighted lines [into register x] and
                        start insert (for {Visual} see |Visual-mode|).  {not
                        in Vi}
                                                        *v_R*
{Visual}["x]R           Currently just like {Visual}["x]S.  In a next version
                        it might work differently. {not in Vi}

Notes:
- You can end Insert and Replace mode with <Esc>.
- See the section "Insert and Replace mode" |mode-ins-repl| for the other
  special characters in these modes.
- The effect of [count] takes place after Vim exits Insert or Replace mode.
- When the 'cpoptions' option contains '$' and the change is within one line,
  Vim continues to show the text to be deleted and puts a '$' at the last
  deleted character.

See |registers| for an explanation of registers.

Replace mode is just like Insert mode, except that every character you enter
deletes one character.  If you reach the end of a line, Vim appends any
further characters (just like Insert mode).  In Replace mode, the backspace
key restores the original text (if there was any).  (See section "Insert and
Replace mode" |mode-ins-repl|).

                                                *cw* *cW*
Special case: When the cursor is in a word, "cw" and "cW" do not include the
white space after a word, they only change up to the end of the word.  This is
because Vim interprets "cw" as change-word, and a word does not include the
following white space.
{Vi: "cw" when on a blank followed by other blanks changes only the first
blank; this is probably a bug, because "dw" deletes all the blanks; use the
'w' flag in 'cpoptions' to make it work like Vi anyway}

If you prefer "cw" to include the space after a word, use this mapping: >
        :map cw dwi
Or use "caw" (see |aw|).

                                                        *:c* *:ch* *:change*
:{range}c[hange][!]     Replace lines of text with some different text.
                        Type a line containing only "." to stop replacing.
                        Without {range}, this command changes only the current
```

line.
Adding [!] toggles 'autoindent' for the time this
command is executed.

==
3. Simple changes *simple-change*

r
r{char} Replace the character under the cursor with {char}.
If {char} is a <CR> or <NL>, a line break replaces the
character. To replace with a real <CR>, use CTRL-V
<CR>. CTRL-V <NL> replaces with a <Nul>.
{Vi: CTRL-V <CR> still replaces with a line break,
cannot replace something with a <CR>}

If {char} is CTRL-E or CTRL-Y the character from the
line below or above is used, just like with |i_CTRL-E|
and |i_CTRL-Y|. This also works with a count, thus
`10r<C-E>` copies 10 characters from the line below.

If you give a [count], Vim replaces [count] characters
with [count] {char}s. When {char} is a <CR> or <NL>,
however, Vim inserts only one <CR>: "5r<CR>" replaces
five characters with a single line break.
When {char} is a <CR> or <NL>, Vim performs
autoindenting. This works just like deleting the
characters that are replaced and then doing
"i<CR><Esc>".
{char} can be entered as a digraph |digraph-arg|.
|:lmap| mappings apply to {char}. The CTRL-^ command
in Insert mode can be used to switch this on/off
|i_CTRL-^|. See |utf-8-char-arg| about using
composing characters when 'encoding' is Unicode.

gr
gr{char} Replace the virtual characters under the cursor with
{char}. This replaces in screen space, not file
space. See |gR| and |Virtual-Replace-mode| for more
details. As with |r| a count may be given.
{char} can be entered like with |r|.
{not available when compiled without the |+vreplace|
feature}

digraph-arg
The argument for Normal mode commands like |r| and |t| is a single character.
When 'cpo' doesn't contain the 'D' flag, this character can also be entered
like |digraphs|. First type CTRL-K and then the two digraph characters.
{not available when compiled without the |+digraphs| feature}

case
The following commands change the case of letters. The currently active
|locale| is used. See |:language|. The LC_CTYPE value matters here.

~
~ 'notildeop' option: Switch case of the character
under the cursor and move the cursor to the right.
If a [count] is given, do that many characters. {Vi:
no count}

~{motion} 'tildeop' option: switch case of {motion} text. {Vi:
tilde cannot be used as an operator}

```
                                                        *g~*
g~{motion}              Switch case of {motion} text. {not in Vi}

g~g~                                                    *g~g~* *g~~*
g~~                     Switch case of current line. {not in Vi}.

                                                        *v_~*
{Visual}~               Switch case of highlighted text (for {Visual} see
                        |Visual-mode|). {not in Vi}

                                                        *v_U*
{Visual}U               Make highlighted text uppercase (for {Visual} see
                        |Visual-mode|). {not in Vi}

                                                        *gU* *uppercase*
gU{motion}              Make {motion} text uppercase. {not in Vi}
                        Example: >
                                :map! <C-F> <Esc>gUiw`]a
<                       This works in Insert mode: press CTRL-F to make the
                        word before the cursor uppercase.  Handy to type
                        words in lowercase and then make them uppercase.


gUgU                                                    *gUgU* *gUU*
gUU                     Make current line uppercase. {not in Vi}.

                                                        *v_u*
{Visual}u               Make highlighted text lowercase (for {Visual} see
                        |Visual-mode|).  {not in Vi}

                                                        *gu* *lowercase*
gu{motion}              Make {motion} text lowercase. {not in Vi}

gugu                                                    *gugu* *guu*
guu                     Make current line lowercase. {not in Vi}.

                                                        *g?* *rot13*
g?{motion}              Rot13 encode {motion} text. {not in Vi}

                                                        *v_g?*
{Visual}g?              Rot13 encode the highlighted text (for {Visual} see
                        |Visual-mode|).  {not in Vi}

g?g?                                                    *g?g?* *g??*
g??                     Rot13 encode current line. {not in Vi}.

To turn one line into title caps, make every first letter of a word
uppercase: >
        :s/\v<(.)(\w*)/\u\1\L\2/g


Adding and subtracting ~
                                                        *CTRL-A*
CTRL-A                  Add [count] to the number or alphabetic character at
                        or after the cursor.  {not in Vi}

                                                        *v_CTRL-A*
{Visual}CTRL-A          Add [count] to the number or alphabetic character in
                        the highlighted text.  {not in Vi}

                                                        *v_g_CTRL-A*
{Visual}g CTRL-A        Add [count] to the number or alphabetic character in
```

the highlighted text. If several lines are highlighted, each one will be incremented by an additional [count] (so effectively creating a [count] incrementing sequence). {not in Vi}
For Example, if you have this list of numbers:

```
1. ~
1. ~
1. ~
1. ~
```

Move to the second "1." and Visually select three lines, pressing g CTRL-A results in:

```
1. ~
2. ~
3. ~
4. ~
```

CTRL-X

CTRL-X Subtract [count] from the number or alphabetic character at or after the cursor. {not in Vi}

v_CTRL-X

{Visual}CTRL-X Subtract [count] from the number or alphabetic character in the highlighted text. {not in Vi}

On MS-Windows, this is mapped to cut Visual text |dos-standard-mappings|. If you want to disable the mapping, use this: >

```
silent! vunmap <C-X>
```

<

v_g_CTRL-X

{Visual}g CTRL-X Subtract [count] from the number or alphabetic character in the highlighted text. If several lines are highlighted, each value will be decremented by an additional [count] (so effectively creating a [count] decrementing sequence). {not in Vi}

The CTRL-A and CTRL-X commands can work for:
- signed and unsigned decimal numbers
- unsigned binary, octal and hexadecimal numbers
- alphabetic characters

This depends on the 'nrformats' option:
- When 'nrformats' includes "bin", Vim assumes numbers starting with '0b' or '0B' are binary.
- When 'nrformats' includes "octal", Vim considers numbers starting with a '0' to be octal, unless the number includes a '8' or '9'. Other numbers are decimal and may have a preceding minus sign.
  If the cursor is on a number, the commands apply to that number; otherwise Vim uses the number to the right of the cursor.
- When 'nrformats' includes "hex", Vim assumes numbers starting with '0x' or '0X' are hexadecimal. The case of the rightmost letter in the number determines the case of the resulting hexadecimal number. If there is no letter in the current number, Vim uses the previously detected case.
- When 'nrformats' includes "alpha", Vim will change the alphabetic character under or after the cursor. This is useful to make lists with an alphabetic index.

For decimals a leading negative sign is considered for incrementing/decrementing, for binary, octal and hex values, it won't be considered. To ignore the sign Visually select the number before using CTRL-A or CTRL-X.

For numbers with leading zeros (including all octal and hexadecimal numbers),

Vim preserves the number of characters in the number when possible. CTRL-A on
"0077" results in "0100", CTRL-X on "0x100" results in "0x0ff".
There is one exception: When a number that starts with a zero is found not to
be octal (it contains a '8' or '9'), but 'nrformats' does include "octal",
leading zeros are removed to avoid that the result may be recognized as an
octal number.

Note that when 'nrformats' includes "octal", decimal numbers with leading
zeros cause mistakes, because they can be confused with octal numbers.

Note similarly, when 'nrformats' includes "bin", binary numbers with a leading
'0x' or '0X' can be interpreted as hexadecimal rather than binary since '0b'
are valid hexadecimal digits.

The CTRL-A command is very useful in a macro. Example: Use the following
steps to make a numbered list.

1. Create the first list entry, make sure it starts with a number.
2. qa - start recording into register 'a'
3. Y - yank the entry
4. p - put a copy of the entry below the first one
5. CTRL-A - increment the number
6. q - stop recording
7. <count>@a - repeat the yank, put and increment <count> times

SHIFTING LINES LEFT OR RIGHT *shift-left-right*

<
<{motion} Shift {motion} lines one 'shiftwidth' leftwards.

<<
<< Shift [count] lines one 'shiftwidth' leftwards.

v_<
{Visual}[count]< Shift the highlighted lines [count] 'shiftwidth'
leftwards (for {Visual} see |Visual-mode|). {not in
Vi}

>
>{motion} Shift {motion} lines one 'shiftwidth' rightwards.

>>
>> Shift [count] lines one 'shiftwidth' rightwards.

v_>
{Visual}[count]> Shift the highlighted lines [count] 'shiftwidth'
rightwards (for {Visual} see |Visual-mode|). {not in
Vi}

:<
:[range]< Shift [range] lines one 'shiftwidth' left. Repeat '<'
for shifting multiple 'shiftwidth's.

:[range]< {count} Shift {count} lines one 'shiftwidth' left, starting
with [range] (default current line |cmdline-ranges|).
Repeat '<' for shifting multiple 'shiftwidth's.

:[range]le[ft] [indent] left align lines in [range]. Sets the indent in the
lines to [indent] (default 0). {not in Vi}

:>

```
:[range]> [flags]       Shift {count} [range] lines one 'shiftwidth' right.
                        Repeat '>' for shifting multiple 'shiftwidth's.
                        See |ex-flags| for [flags].

:[range]> {count} [flags]
                        Shift {count} lines one 'shiftwidth' right, starting
                        with [range] (default current line |cmdline-ranges|).
                        Repeat '>' for shifting multiple 'shiftwidth's.
                        See |ex-flags| for [flags].

The ">" and "<" commands are handy for changing the indentation within
programs.  Use the 'shiftwidth' option to set the size of the white space
which these commands insert or delete.  Normally the 'shiftwidth' option is 8,
but you can set it to, say, 3 to make smaller indents.  The shift leftwards
stops when there is no indent.  The shift right does not affect empty lines.

If the 'shiftround' option is on, the indent is rounded to a multiple of
'shiftwidth'.

If the 'smartindent' option is on, or 'cindent' is on and 'cinkeys' contains
'#' with a zero value, shift right does not affect lines starting with '#'
(these are supposed to be C preprocessor lines that must stay in column 1).

When the 'expandtab' option is off (this is the default) Vim uses <Tab>s as
much as possible to make the indent.  You can use ">><<" to replace an indent
made out of spaces with the same indent made out of <Tab>s (and a few spaces
if necessary).  If the 'expandtab' option is on, Vim uses only spaces.  Then
you can use ">><<" to replace <Tab>s in the indent by spaces (or use
`:retab!`).

To move a line several 'shiftwidth's, use Visual mode or the `:` commands.
For example: >
        Vjj4>           move three lines 4 indents to the right
        :<<<            move current line 3 indents to the left
        :>> 5           move 5 lines 2 indents to the right
        :5>>            move line 5 2 indents to the right

==============================================================================
4. Complex changes                                      *complex-change*

4.1 Filter commands                                     *filter*

A filter is a program that accepts text at standard input, changes it in some
way, and sends it to standard output.  You can use the commands below to send
some text through a filter, so that it is replaced by the filter output.
Examples of filters are "sort", which sorts lines alphabetically, and
"indent", which formats C program files (you need a version of indent that
works like a filter; not all versions do).  The 'shell' option specifies the
shell Vim uses to execute the filter command (See also the 'shelltype'
option).  You can repeat filter commands with ".".  Vim does not recognize a
comment (starting with '"') after the `:!` command.

                                                        *!*
!{motion}{filter}       Filter {motion} text lines through the external
                        program {filter}.

                                                        *!!*
!!{filter}              Filter [count] lines through the external program
                        {filter}.

                                                        *v_!*
{Visual}!{filter}       Filter the highlighted lines through the external
```

program {filter} (for {Visual} see |Visual-mode|).
{not in Vi}

:{range}![!]{filter} [!][arg] *:range!*
Filter {range} lines through the external program
{filter}. Vim replaces the optional bangs with the
latest given command and appends the optional [arg].
Vim saves the output of the filter command in a
temporary file and then reads the file into the buffer
|tempfile|. Vim uses the 'shellredir' option to
redirect the filter output to the temporary file.
However, if the 'shelltemp' option is off then pipes
are used when possible (on Unix).
When the 'R' flag is included in 'cpoptions' marks in
the filtered lines are deleted, unless the
|:keepmarks| command is used. Example: >
:keepmarks '<,'>!sort
< When the number of lines after filtering is less than
before, marks in the missing lines are deleted anyway.

=
={motion} Filter {motion} lines through the external program
given with the 'equalprg' option. When the 'equalprg'
option is empty (this is the default), use the
internal formatting function |C-indenting| and
|'lisp'|. But when 'indentexpr' is not empty, it will
be used instead |indent-expression|. When Vim was
compiled without internal formatting then the "indent"
program is used as a last resort.

==
== Filter [count] lines like with ={motion}.

v_=
{Visual}= Filter the highlighted lines like with ={motion}.
{not in Vi}

tempfile *setuid*
Vim uses temporary files for filtering, generating diffs and also for
tempname(). For Unix, the file will be in a private directory (only
accessible by the current user) to avoid security problems (e.g., a symlink
attack or other people reading your file). When Vim exits the directory and
all files in it are deleted. When Vim has the setuid bit set this may cause
problems, the temp file is owned by the setuid user but the filter command
probably runs as the original user.
On MS-DOS and OS/2 the first of these directories that works is used: $TMP,
$TEMP, c:\TMP, c:\TEMP.
For Unix the list of directories is: $TMPDIR, /tmp, current-dir, $HOME.
For MS-Windows the GetTempFileName() system function is used.
For other systems the tmpnam() library function is used.

4.2 Substitute *:substitute*
:s *:su*

:[range]s[ubstitute]/{pattern}/{string}/[flags] [count]
For each line in [range] replace a match of {pattern}
with {string}.
For the {pattern} see |pattern|.
{string} can be a literal string, or something
special; see |sub-replace-special|.

E939
When [range] and [count] are omitted, replace in the current line only. When [count] is given, replace in [count] lines, starting with the last line in [range]. When [range] is omitted start in the current line. [count] must be a positive number. Also see |cmdline-ranges|.

See |:s_flags| for [flags].

:[range]s[ubstitute] [flags] [count]
:[range]&[&][flags] [count] *:&*
Repeat last :substitute with same search pattern and substitute string, but without the same flags. You may add [flags], see |:s_flags|.
Note that after `:substitute` the '&' flag can't be used, it's recognized as a pattern separator.
The space between `:substitute` and the 'c', 'g', 'i', 'I' and 'r' flags isn't required, but in scripts it's a good idea to keep it to avoid confusion.

:[range]~[&][flags] [count] *:~*
Repeat last substitute with same substitute string but with last used search pattern. This is like `:&r`. See |:s_flags| for [flags].

&
& Synonym for `:s` (repeat last substitute). Note that the flags are not remembered, thus it might actually work differently. You can use `:&&` to keep the flags.

g&
g& Synonym for `:%s//~/&` (repeat last substitute with last search pattern on all lines with the same flags).
For example, when you first do a substitution with `:s/pattern/repl/flags` and then `/search` for something else, `g&` will do `:%s/search/repl/flags`.
Mnemonic: global substitute. {not in Vi}

:snomagic *:sno*
:[range]sno[magic] ... Same as `:substitute`, but always use 'nomagic'. {not in Vi}

:smagic *:sm*
:[range]sm[agic] ... Same as `:substitute`, but always use 'magic'. {not in Vi}

:s_flags
The flags that you can use for the substitute commands:

:&&
[&] Must be the first one: Keep the flags from the previous substitute command. Examples: >
:&&
:s/this/that/&
< Note that `:s` and `:&` don't keep the flags. {not in Vi}

[c] Confirm each substitution. Vim highlights the matching string (with |hl-IncSearch|). You can type: *:s_c*
'y' to substitute this match

```
            'l'     to substitute this match and then quit ("last")
            'n'     to skip this match
            <Esc>   to quit substituting
            'a'     to substitute this and all remaining matches {not in Vi}
            'q'     to quit substituting {not in Vi}
            CTRL-E  to scroll the screen up {not in Vi, not available when
                        compiled without the |+insert_expand| feature}
            CTRL-Y  to scroll the screen down {not in Vi, not available when
                        compiled without the |+insert_expand| feature}
        If the 'edcompatible' option is on, Vim remembers the [c] flag and
        toggles it each time you use it, but resets it when you give a new
        search pattern.
        {not in Vi: highlighting of the match, other responses than 'y' or 'n'}

[e]     When the search pattern fails, do not issue an error message and, in
        particular, continue in maps as if no error occurred.  This is most
        useful to prevent the "No match" error from breaking a mapping.  Vim
        does not suppress the following error messages, however:
                Regular expressions can't be delimited by letters
                \ should be followed by /, ? or &
                No previous substitute regular expression
                Trailing characters
                Interrupted
        {not in Vi}

[g]     Replace all occurrences in the line.  Without this argument,
        replacement occurs only for the first occurrence in each line.  If
        the 'edcompatible' option is on, Vim remembers this flag and toggles
        it each time you use it, but resets it when you give a new search
        pattern.  If the 'gdefault' option is on, this flag is on by default
        and the [g] argument switches it off.

[i]     Ignore case for the pattern.  The 'ignorecase' and 'smartcase' options
        are not used.
        {not in Vi}

[I]     Don't ignore case for the pattern.  The 'ignorecase' and 'smartcase'
        options are not used.
        {not in Vi}

[n]     Report the number of matches, do not actually substitute.  The [c]
        flag is ignored.  The matches are reported as if 'report' is zero.
        Useful to |count-items|.
        If \= |sub-replace-expression| is used, the expression will be
        evaluated in the |sandbox| at every match.

[p]     Print the line containing the last substitute.

[#]     Like [p] and prepend the line number.

[l]     Like [p] but print the text like |:list|.

[r]     Only useful in combination with `:&` or `:s` without arguments.  `:&r`
        works the same way as `:~`:  When the search pattern is empty, use the
        previously used search pattern instead of the search pattern from the
        last substitute or `:global`.  If the last command that did a search
        was a substitute or `:global`, there is no effect.  If the last
        command was a search command such as "/", use the pattern from that
        command.
        For `:s` with an argument this already happens: >
                :s/blue/red/
                /green
```

```
                :s//red/   or  :~   or  :&r
<       The last commands will replace "green" with "red". >
                :s/blue/red/
                /green
                :&
<       The last command will replace "blue" with "red".
        {not in Vi}

Note that there is no flag to change the "magicness" of the pattern.  A
different command is used instead, or you can use |/\v| and friends.  The
reason is that the flags can only be found by skipping the pattern, and in
order to skip the pattern the "magicness" must be known.  Catch 22!

If the {pattern} for the substitute command is empty, the command uses the
pattern from the last substitute or `:global` command.  If there is none, but
there is a previous search pattern, that one is used.  With the [r] flag, the
command uses the pattern from the last substitute, `:global`, or search
command.

If the {string} is omitted the substitute is done as if it's empty.  Thus the
matched pattern is deleted.  The separator after {pattern} can also be left
out then.  Example: >
        :%s/TESTING
This deletes "TESTING" from all lines, but only one per line.

For compatibility with Vi these two exceptions are allowed:
"\/{string}/" and "\?{string}?" do the same as "//{string}/r".
"\&{string}&" does the same as "//{string}/".
                                                        *E146*
Instead of the '/' which surrounds the pattern and replacement string, you
can use any other single-byte character, but not an alphanumeric character,
'\', '"' or '|'.  This is useful if you want to include a '/' in the search
pattern or replacement string.  Example: >
        :s+/+//+

For the definition of a pattern, see |pattern|.  In Visual block mode, use
|/\%V| in the pattern to have the substitute work in the block only.
Otherwise it works on whole lines anyway.

                                        *sub-replace-special* *:s\=*
When the {string} starts with "\=" it is evaluated as an expression, see
|sub-replace-expression|.  You can use that for complex replacement or special
characters.

Otherwise these characters in {string} have a special meaning:
                                                                *:s%*
When {string} is equal to "%" and '/' is included with the 'cpoptions' option,
then the {string} of the previous substitute command is used, see |cpo-/|

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

```
 \~        ~      replaced with ~                                        *s/\~*
      \u          next character made uppercase                          *s/\u*
      \U          following characters made uppercase, until \E          *s/\U*
      \l          next character made lowercase                          *s/\l*
      \L          following characters made lowercase, until \E          *s/\L*
      \e          end of \u, \U, \l and \L (NOTE: not <Esc>!)            *s/\e*
      \E          end of \u, \U, \l and \L                               *s/\E*
      <CR>        split line in two at this point
                  (Type the <CR> as CTRL-V <Enter>)                      *s<CR>*
      \r          idem                                                   *s/\r*
      \<CR>       insert a carriage-return (CTRL-M)
                  (Type the <CR> as CTRL-V <Enter>)                      *s/\<CR>*
      \n          insert a <NL> (<NUL> in the file)
                  (does NOT break the line)                              *s/\n*
      \b          insert a <BS>                                          *s/\b*
      \t          insert a <Tab>                                         *s/\t*
      \\          insert a single backslash                              *s/\\*
      \x          where x is any character not mentioned above:
                  Reserved for future expansion

The special meaning is also used inside the third argument {sub} of
the |substitute()| function with the following exceptions:
  - A % inserts a percent literally without regard to 'cpoptions'.
  - magic is always set without regard to 'magic'.
  - A ~ inserts a tilde literally.
  - <CR> and \r inserts a carriage-return (CTRL-M).
  - \<CR> does not have a special meaning. it's just one of \x.

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

Note: "\L\u" can be used to capitalize the first letter of a word.  This is
not compatible with Vi and older versions of Vim, where the "\u" would cancel
out the "\L". Same for "\U\l".

Note: In previous versions CTRL-V was handled in a special way.  Since this is
not Vi compatible, this was removed.  Use a backslash instead.

command         text    result ~
:s/aa/a^Ma/     aa      a<line-break>a
:s/aa/a\^Ma/    aa      a^Ma
:s/aa/a\\^Ma/   aa      a\<line-break>a

(you need to type CTRL-V <CR> to get a ^M here)

The numbering of "\1", "\2" etc. is done based on which "\(" comes first in
the pattern (going left to right).  When a parentheses group matches several
times, the last one will be used for "\1", "\2", etc.  Example: >
  :s/\(\(a[a-d] \)*\)/\2/      modifies "aa ab x" to "ab x"
The "\2" is for "\(a[a-d] \)".  At first it matches "aa ", secondly "ab ".

When using parentheses in combination with '|', like in \([ab]\)\|\([cd]\),
either the first or second pattern in parentheses did not match, so either
\1 or \2 is empty.  Example: >
  :s/\([ab]\)\|\([cd]\)/\1x/g   modifies "a b c d"  to "ax bx x x"
<
```

```
               *:sc* *:sce* *:scg* *:sci* *:scI* *:scl* *:scp* *:sg* *:sgc*
               *:sge* *:sgi* *:sgI* *:sgl* *:sgn* *:sgp* *:sgr* *:sI* *:si*
               *:sic* *:sIc* *:sie* *:sIe* *:sIg* *:sIl* *:sin* *:sIn* *:sIp*
               *:sip* *:sIr* *:sir* *:sr* *:src* *:srg* *:sri* *:srI* *:srl*
               *:srn* *:srp*
2-letter and 3-letter :substitute commands ~

     List of :substitute commands
     |      c    e    g    i    I    n    p    l    r
     | c  :sc  :sce :scg :sci :scI :scn :scp :scl  ---
     | e
     | g  :sgc :sge :sg  :sgi :sgI :sgn :sgp :sgl :sgr
     | i  :sic :sie  --- :si  :siI :sin :sip  --- :sir
     | I  :sIc :sIe :sIg :sIi :sI  :sIn :sIp :sIl :sIr
     | n
     | p
     | l
     | r  :src  --- :srg :sri :srI :srn :srp :srl :sr

Exceptions:
     :scr  is  `:scriptnames`
     :se   is  `:set`
     :sig  is  `:sign`
     :sil  is  `:silent`
     :sn   is  `:snext`
     :sp   is  `:split`
     :sl   is  `:sleep`
     :sre  is  `:srewind`


Substitute with an expression                   *sub-replace-expression*
                                                *sub-replace-\=* *s/\=*
When the substitute string starts with "\=" the remainder is interpreted as an
expression.

The special meaning for characters as mentioned at |sub-replace-special| does
not apply except for "<CR>".  A <NL> character is used as a line break, you
can get one with a double-quote string: "\n".  Prepend a backslash to get a
real <NL> character (which will be a NUL in the file).

The "\=" notation can also be used inside the third argument {sub} of
|substitute()| function.  In this case, the special meaning for characters as
mentioned at |sub-replace-special| does not apply at all. Especially, <CR> and
<NL> are interpreted not as a line break but as a carriage-return and a
new-line respectively.

When the result is a |List| then the items are joined with separating line
breaks.  Thus each item becomes a line, except that they can contain line
breaks themselves.

The whole matched text can be accessed with "submatch(0)".  The text matched
with the first pair of () with "submatch(1)".  Likewise for further
sub-matches in ().

Be careful: The separation character must not appear in the expression!
Consider using a character like "@" or ":".  There is no problem if the result
of the expression contains the separation character.

Examples: >
        :s@\n@\="\r" . expand("$HOME") . "\r"@
This replaces an end-of-line with a new line containing the value of $HOME. >
```

```
        s/E/\="\<Char-0x20ac>"/g
This replaces each 'E' character with a euro sign.  Read more in |<Char->|.


4.3 Search and replace                                  *search-replace*

                                                        *:pro* *:promptfind*
:promptf[ind] [string]
                        Put up a Search dialog.  When [string] is given, it is
                        used as the initial search string.
                        {only for Win32, Motif and GTK GUI}

                                                *:promptr* *:promptrepl*
:promptr[epl] [string]
                        Put up a Search/Replace dialog.  When [string] is
                        given, it is used as the initial search string.
                        {only for Win32, Motif and GTK GUI}


4.4 Changing tabs                                       *change-tabs*
                                                        *:ret* *:retab* *:retab!*
:[range]ret[ab][!] [new_tabstop]
                        Replace all sequences of white-space containing a
                        <Tab> with new strings of white-space using the new
                        tabstop value given.  If you do not specify a new
                        tabstop size or it is zero, Vim uses the current value
                        of 'tabstop'.
                        The current value of 'tabstop' is always used to
                        compute the width of existing tabs.
                        With !, Vim also replaces strings of only normal
                        spaces with tabs where appropriate.
                        With 'expandtab' on, Vim replaces all tabs with the
                        appropriate number of spaces.
                        This command sets 'tabstop' to the new value given,
                        and if performed on the whole file, which is default,
                        should not make any visible change.
                        Careful: This command modifies any <Tab> characters
                        inside of strings in a C program.  Use "\t" to avoid
                        this (that's a good habit anyway).
                        `:retab!` may also change a sequence of spaces by
                        <Tab> characters, which can mess up a printf().
                        {not in Vi}

                                                        *retab-example*
Example for using autocommands and ":retab" to edit a file which is stored
with tabstops at 8 but edited with tabstops set at 4.  Warning: white space
inside of strings can change!  Also see 'softtabstop' option. >

  :auto BufReadPost     *.xx    retab! 4
  :auto BufWritePre     *.xx    retab! 8
  :auto BufWritePost    *.xx    retab! 4
  :auto BufNewFile      *.xx    set ts=4

==============================================================================
5. Copying and moving text                              *copy-move*

                                                        *quote*
"{a-zA-Z0-9.%#:-"}      Use register {a-zA-Z0-9.%#:-"} for next delete, yank
                        or put (use uppercase character to append with
                        delete and yank) ({.%#:} only work with put).

                                                        *:reg* *:registers*
```

```
:reg[isters]            Display the contents of all numbered and named
                        registers.  If a register is written to for |:redir|
                        it will not be listed.
                        {not in Vi}


:reg[isters] {arg}      Display the contents of the numbered and named
                        registers that are mentioned in {arg}.  For example: >
                                :reg 1a
<                       to display registers '1' and 'a'.  Spaces are allowed
                        in {arg}.  {not in Vi}

                                                        *:di* *:display*
:di[splay] [arg]        Same as :registers.  {not in Vi}

                                                        *y* *yank*
["x]y{motion}           Yank {motion} text [into register x].  When no
                        characters are to be yanked (e.g., "y0" in column 1),
                        this is an error when 'cpoptions' includes the 'E'
                        flag.

                                                        *yy*
["x]yy                  Yank [count] lines [into register x] |linewise|.

                                                        *Y*
["x]Y                   yank [count] lines [into register x] (synonym for
                        yy, |linewise|).  If you like "Y" to work from the
                        cursor to the end of line (which is more logical,
                        but not Vi-compatible) use ":map Y y$".

                                                        *v_y*
{Visual}["x]y           Yank the highlighted text [into register x] (for
                        {Visual} see |Visual-mode|).  {not in Vi}

                                                        *v_Y*
{Visual}["x]Y           Yank the highlighted lines [into register x] (for
                        {Visual} see |Visual-mode|).  {not in Vi}

                                                        *:y* *:yank* *E850*
:[range]y[ank] [x]      Yank [range] lines [into register x]. Yanking to the
                        "* or "+ registers is possible only when the
                        |+clipboard| feature is included.

:[range]y[ank] [x] {count}
                        Yank {count} lines, starting with last line number
                        in [range] (default: current line |cmdline-ranges|),
                        [into register x].

                                                        *p* *put* *E353*
["x]p                   Put the text [from register x] after the cursor
                        [count] times.  {Vi: no count}

                                                        *P*
["x]P                   Put the text [from register x] before the cursor
                        [count] times.  {Vi: no count}

                                                        *<MiddleMouse>*
["x]<MiddleMouse>       Put the text from a register before the cursor [count]
                        times.  Uses the "* register, unless another is
                        specified.
                        Leaves the cursor at the end of the new text.
                        Using the mouse only works when 'mouse' contains 'n'
```

```
                        or 'a'.
                        {not in Vi}
                        If you have a scrollwheel and often accidentally paste
                        text, you can use these mappings to disable the
                        pasting with the middle mouse button: >
                                :map <MiddleMouse> <Nop>
                                :imap <MiddleMouse> <Nop>
<                       You might want to disable the multi-click versions
                        too, see |double-click|.

                                                        *gp*
["x]gp                  Just like "p", but leave the cursor just after the new
                        text.  {not in Vi}

                                                        *gP*
["x]gP                  Just like "P", but leave the cursor just after the new
                        text.  {not in Vi}

                                                        *:pu* *:put*
:[line]pu[t] [x]        Put the text [from register x] after [line] (default
                        current line).  This always works |linewise|, thus
                        this command can be used to put a yanked block as new
                        lines.
                        If no register is specified, it depends on the 'cb'
                        option: If 'cb' contains "unnamedplus", paste from the
                        + register |quoteplus|.  Otherwise, if 'cb' contains
                        "unnamed", paste from the * register |quotestar|.
                        Otherwise, paste from the unnamed register
                        |quote_quote|.
                        The register can also be '=' followed by an optional
                        expression.  The expression continues until the end of
                        the command.  You need to escape the '|' and '"'
                        characters to prevent them from terminating the
                        command.  Example: >
                                :put ='path' . \",/test\"
<                       If there is no expression after '=', Vim uses the
                        previous expression.  You can see it with ":dis =".

:[line]pu[t]! [x]       Put the text [from register x] before [line] (default
                        current line).

["x]]p              or                              *]p* *]<MiddleMouse>*
["x]]<MiddleMouse>      Like "p", but adjust the indent to the current line.
                        Using the mouse only works when 'mouse' contains 'n'
                        or 'a'.  {not in Vi}

["x][P              or                              *[P*
["x]]P              or                              *]P*
["x][p              or                              *[p* *[<MiddleMouse>*
["x][<MiddleMouse>      Like "P", but adjust the indent to the current line.
                        Using the mouse only works when 'mouse' contains 'n'
                        or 'a'.  {not in Vi}

You can use these commands to copy text from one place to another.  Do this
by first getting the text into a register with a yank, delete or change
command, then inserting the register contents with a put command.  You can
also use these commands to move text from one file to another, because Vim
preserves all registers when changing buffers (the CTRL-^ command is a quick
way to toggle between two files).

                                *linewise-register* *characterwise-register*
You can repeat the put commands with "." (except for :put) and undo them.  If
```

the command that was used to get the text into the register was |linewise|,
Vim inserts the text below ("p") or above ("P") the line where the cursor is.
Otherwise Vim inserts the text after ("p") or before ("P") the cursor. With
the ":put" command, Vim always inserts the text in the next line. You can
exchange two characters with the command sequence "xp". You can exchange two
lines with the command sequence "ddp". You can exchange two words with the
command sequence "deep" (start with the cursor in the blank space before the
first word). You can use the "']" or "`]" command after the put command to
move the cursor to the end of the inserted text, or use "'[" or "`[" to move
the cursor to the start.

put-Visual-mode *v_p* *v_P*
When using a put command like |p| or |P| in Visual mode, Vim will try to
replace the selected text with the contents of the register. Whether this
works well depends on the type of selection and the type of the text in the
register. With blockwise selection it also depends on the size of the block
and whether the corners are on an existing character. (Implementation detail:
it actually works by first putting the register after the selection and then
deleting the selection.)
The previously selected text is put in the unnamed register. If you want to
put the same text into a Visual selection several times you need to use
another register. E.g., yank the text to copy, Visually select the text to
replace and use "0p . You can repeat this as many times as you like, the
unnamed register will be changed each time.

When you use a blockwise Visual mode command and yank only a single line into
a register, a paste on a visual selected area will paste that single line on
each of the selected lines (thus replacing the blockwise selected region by a
block of the pasted line).

blockwise-register
If you use a blockwise Visual mode command to get the text into the register,
the block of text will be inserted before ("P") or after ("p") the cursor
column in the current and next lines. Vim makes the whole block of text start
in the same column. Thus the inserted text looks the same as when it was
yanked or deleted. Vim may replace some <Tab> characters with spaces to make
this happen. However, if the width of the block is not a multiple of a <Tab>
width and the text after the inserted block contains <Tab>s, that text may be
misaligned.

Note that after a characterwise yank command, Vim leaves the cursor on the
first yanked character that is closest to the start of the buffer. This means
that "yl" doesn't move the cursor, but "yh" moves the cursor one character
left.
Rationale: In Vi the "y" command followed by a backwards motion would
sometimes not move the cursor to the first yanked character,
because redisplaying was skipped. In Vim it always moves to
the first character, as specified by Posix.
With a linewise yank command the cursor is put in the first line, but the
column is unmodified, thus it may not be on the first yanked character.

There are ten types of registers: *registers* *E354*
1. The unnamed register ""
2. 10 numbered registers "0 to "9
3. The small delete register "-
4. 26 named registers "a to "z or "A to "Z
5. three read-only registers ":, "., "%
6. alternate buffer register "#
7. the expression register "=
8. The selection and drop registers "*, "+ and "~
9. The black hole register "_
10. Last search pattern register "/

1. Unnamed register ""   *quote_quote* *quotequote*
Vim fills this register with text deleted with the "d", "c", "s", "x" commands
or copied with the yank "y" command, regardless of whether or not a specific
register was used (e.g. "xdd). This is like the unnamed register is pointing
to the last used register. Thus when appending using an uppercase register
name, the unnamed register contains the same text as the named register.
An exception is the '_' register: "_dd does not store the deleted text in any
register.
Vim uses the contents of the unnamed register for any put command (p or P)
which does not specify a register. Additionally you can access it with the
name '""'. This means you have to type two double quotes. Writing to the ""
register writes to register "0.
{Vi: register contents are lost when changing files, no '""'}

2. Numbered registers "0 to "9   *quote_number* *quote0* *quote1*
                                        quote2 *quote3* *quote4* *quote9*
Vim fills these registers with text from yank and delete commands.
   Numbered register 0 contains the text from the most recent yank command,
unless the command specified another register with ["x].
   Numbered register 1 contains the text deleted by the most recent delete or
change command, unless the command specified another register or the text is
less than one line (the small delete register is used then). An exception is
made for the delete operator with these movement commands: |%|, |(|, |)|, |`|,
|/|, |?|, |n|, |N|, |{| and |}|. Register "1 is always used then (this is Vi
compatible). The "- register is used as well if the delete is within a line.
Note that these characters may be mapped. E.g. |%| is mapped by the matchit
plugin.
   With each successive deletion or change, Vim shifts the previous contents
of register 1 into register 2, 2 into 3, and so forth, losing the previous
contents of register 9.
{Vi: numbered register contents are lost when changing files; register 0 does
not exist}

3. Small delete register "-   *quote_-* *quote-*
This register contains text from commands that delete less than one line,
except when the command specifies a register with ["x].
{not in Vi}

4. Named registers "a to "z or "A to "Z   *quote_alpha* *quotea*
Vim fills these registers only when you say so. Specify them as lowercase
letters to replace their previous contents or as uppercase letters to append
to their previous contents. When the '>' flag is present in 'cpoptions' then
a line break is inserted before the appended text.

5. Read-only registers ":, ". and "%
These are '%', '#', ':' and '.'. You can use them only with the "p", "P",
and ":put" commands and with CTRL-R. {not in Vi}
                                                quote_. *quote.* *E29*
        ".   Contains the last inserted text (the same as what is inserted
                with the insert mode commands CTRL-A and CTRL-@). Note: this
                doesn't work with CTRL-R on the command-line. It works a bit
                differently, like inserting the text instead of putting it
                ('textwidth' and other options affect what is inserted).
                                                        quote_% *quote%*
        "%   Contains the name of the current file.
                                                quote_: *quote:* *E30*
        ":   Contains the most recent executed command-line. Example: Use
                "@:" to repeat the previous command-line command.
                The command-line is only stored in this register when at least
                one character of it was typed. Thus it remains unchanged if
                the command was completely from a mapping.

{not available when compiled without the |+cmdline_hist|
feature}
quote_# *quote#*
6. Alternate file register "#
Contains the name of the alternate file for the current window. It will
change how the |CTRL-^| command works.
This register is writable, mainly to allow for restoring it after a plugin has
changed it. It accepts buffer number: >
    let altbuf = bufnr(@#)
    ...
    let @# = altbuf
It will give error |E86| if you pass buffer number and this buffer does not
exist.
It can also accept a match with an existing buffer name: >
    let @# = 'buffer_name'
Error |E93| if there is more than one buffer matching the given name or |E94|
if none of buffers matches the given name.

7. Expression register "= *quote_=* *quote=* *@=*
This is not really a register that stores text, but is a way to use an
expression in commands which use a register. The expression register is
read-write.

When typing the '=' after " or CTRL-R the cursor moves to the command-line,
where you can enter any expression (see |expression|). All normal
command-line editing commands are available, including a special history for
expressions. When you end the command-line by typing <CR>, Vim computes the
result of the expression. If you end it with <Esc>, Vim abandons the
expression. If you do not enter an expression, Vim uses the previous
expression (like with the "/" command).

The expression must evaluate to a String. A Number is always automatically
converted to a String. For the "p" and ":put" command, if the result is a
Float it's converted into a String. If the result is a List each element is
turned into a String and used as a line. A Dictionary or FuncRef results in
an error message (use string() to convert).

If the "= register is used for the "p" command, the String is split up at <NL>
characters. If the String ends in a <NL>, it is regarded as a linewise
register. {not in Vi}

8. Selection and drop registers "*, "+ and "~
Use these registers for storing and retrieving the selected text for the GUI.
See |quotestar| and |quoteplus|. When the clipboard is not available or not
working, the unnamed register is used instead. For Unix systems the clipboard
is only available when the |+xterm_clipboard| feature is present. {not in Vi}

Note that there is only a distinction between "* and "+ for X11 systems. For
an explanation of the difference, see |x11-selection|. Under MS-Windows, use
of "* and "+ is actually synonymous and refers to the |gui-clipboard|.

quote_~ *quote~* *<Drop>*
The read-only "~ register stores the dropped text from the last drag'n'drop
operation. When something has been dropped onto Vim, the "~ register is
filled in and the <Drop> pseudo key is sent for notification. You can remap
this key if you want; the default action (for all modes) is to insert the
contents of the "~ register at the cursor position. {not in Vi}
{only available when compiled with the |+dnd| feature, currently only with the
GTK GUI}

Note: The "~ register is only used when dropping plain text onto Vim.
Drag'n'drop of URI lists is handled internally.

9. Black hole register "_ *quote_*
When writing to this register, nothing happens. This can be used to delete
text without affecting the normal registers. When reading from this register,
nothing is returned. {not in Vi}

10. Last search pattern register "/ *quote_/* *quote/*
Contains the most recent search-pattern. This is used for "n" and 'hlsearch'.
It is writable with `:let`, you can change it to have 'hlsearch' highlight
other matches without actually searching. You can't yank or delete into this
register. The search direction is available in |v:searchforward|.
Note that the value is restored when returning from a function
|function-search-undo|.
{not in Vi}

@/
You can write to a register with a `:let` command |:let-@|. Example: >
:let @/ = "the"

If you use a put command without specifying a register, Vim uses the register
that was last filled (this is also the contents of the unnamed register). If
you are confused, use the `:dis` command to find out what Vim will put (this
command displays all named and numbered registers; the unnamed register is
labelled '""').

The next three commands always work on whole lines.

:[range]co[py] {address} *:co* *:copy*
Copy the lines given by [range] to below the line
given by {address}.

:t
:t Synonym for copy.

:[range]m[ove] {address} *:m* *:mo* *:move* *E134*
Move the lines given by [range] to below the line
given by {address}.

==========
6. Formatting text *formatting*

:[range]ce[nter] [width] *:ce* *:center*
Center lines in [range] between [width] columns
(default 'textwidth' or 80 when 'textwidth' is 0).
{not in Vi}

:[range]ri[ght] [width] *:ri* *:right*
Right-align lines in [range] at [width] columns
(default 'textwidth' or 80 when 'textwidth' is 0).
{not in Vi}

:le *:left*
:[range]le[ft] [indent]
Left-align lines in [range]. Sets the indent in the
lines to [indent] (default 0). {not in Vi}

gq
gq{motion} Format the lines that {motion} moves over.
Formatting is done with one of three methods:
1. If 'formatexpr' is not empty the expression is
evaluated. This can differ for each buffer.
2. If 'formatprg' is not empty an external program

```
                           is used.
                        3. Otherwise formatting is done internally.

                        In the third case the 'textwidth' option controls the
                        length of each formatted line (see below).
                        If the 'textwidth' option is 0, the formatted line
                        length is the screen width (with a maximum width of
                        79).
                        The 'formatoptions' option controls the type of
                        formatting |fo-table|.
                        The cursor is left on the first non-blank of the last
                        formatted line.
                        NOTE: The "Q" command formerly performed this
                        function.  If you still want to use "Q" for
                        formatting, use this mapping: >
                                :nnoremap Q gq

gqgq                                                    *gqgq* *gqq*
gqq                     Format the current line.  With a count format that
                        many lines.  {not in Vi}

                                                        *v_gq*
{Visual}gq              Format the highlighted text.  (for {Visual} see
                        |Visual-mode|).  {not in Vi}

                                                        *gw*
gw{motion}              Format the lines that {motion} moves over.  Similar to
                        |gq| but puts the cursor back at the same position in
                        the text.  However, 'formatprg' and 'formatexpr' are
                        not used.  {not in Vi}

gwgw                                                    *gwgw* *gww*
gww                     Format the current line as with "gw".  {not in Vi}

                                                        *v_gw*
{Visual}gw              Format the highlighted text as with "gw".  (for
                        {Visual} see |Visual-mode|).  {not in Vi}

Example: To format the current paragraph use:                   *gqap*  >
        gqap

The "gq" command leaves the cursor in the line where the motion command takes
the cursor.  This allows you to repeat formatting repeated with ".".  This
works well with "gqj" (format current and next line) and "gq}" (format until
end of paragraph).  Note: When 'formatprg' is set, "gq" leaves the cursor on
the first formatted line (as with using a filter command).

If you want to format the current paragraph and continue where you were, use: >
        gwap
If you always want to keep paragraphs formatted you may want to add the 'a'
flag to 'formatoptions'.  See |auto-format|.

If the 'autoindent' option is on, Vim uses the indent of the first line for
the following lines.

Formatting does not change empty lines (but it does change lines with only
white space!).

The 'joinspaces' option is used when lines are joined together.

You can set the 'formatexpr' option to an expression or the 'formatprg' option
to the name of an external program for Vim to use for text formatting.  The
```

'textwidth' and other options have no effect on formatting by an external
program.

right-justify
There is no command in Vim to right justify text. You can do it with
an external command, like "par" (e.g.: "!}par" to format until the end of the
paragraph) or set 'formatprg' to "par".

format-comments
An overview of comment formatting is in section |30.6| of the user manual.

Vim can automatically insert and format comments in a special way. Vim
recognizes a comment by a specific string at the start of the line (ignoring
white space). Three types of comments can be used:

- A comment string that repeats at the start of each line. An example is the
  type of comment used in shell scripts, starting with "#".
- A comment string that occurs only in the first line, not in the following
  lines. An example is this list with dashes.
- Three-piece comments that have a start string, an end string, and optional
  lines in between. The strings for the start, middle and end are different.
  An example is the C style comment:
```
        /*
         * this is a C comment
         */
```

The 'comments' option is a comma-separated list of parts. Each part defines a
type of comment string. A part consists of:
        {flags}:{string}

{string} is the literal text that must appear.

{flags}:
  n   Nested comment. Nesting with mixed parts is allowed. If 'comments'
        is "n:),n:>" a line starting with ">) >" is a comment.

  b   Blank (<Space>, <Tab> or <EOL>) required after {string}.

  f   Only the first line has the comment string. Do not repeat comment on
        the next line, but preserve indentation (e.g., a bullet-list).

  s   Start of three-piece comment

  m   Middle of a three-piece comment

  e   End of a three-piece comment

  l   Left align. Used together with 's' or 'e', the leftmost character of
        start or end will line up with the leftmost character from the middle.
        This is the default and can be omitted. See below for more details.

  r   Right align. Same as above but rightmost instead of leftmost. See
        below for more details.

  O   Don't consider this comment for the "O" command.

  x   Allows three-piece comments to be ended by just typing the last
        character of the end-comment string as the first action on a new
        line when the middle-comment string has been inserted automatically.
        See below for more details.

  {digits}

```
        When together with 's' or 'e': add {digit} amount of offset to an
        automatically inserted middle or end comment leader. The offset begins
        from a left alignment. See below for more details.

  -{digits}
        Like {digits} but reduce the indent.  This only works when there is
        some indent for the start or end part that can be removed.

When a string has none of the 'f', 's', 'm' or 'e' flags, Vim assumes the
comment string repeats at the start of each line.  The flags field may be
empty.

Any blank space in the text before and after the {string} is part of the
{string}, so do not include leading or trailing blanks unless the blanks are a
required part of the comment string.

When one comment leader is part of another, specify the part after the whole.
For example, to include both "-" and "->", use >
        :set comments=f:->,f:-

A three-piece comment must always be given as start,middle,end, with no other
parts in between.  An example of a three-piece comment is >
        sr:/*,mb:*,ex:*/
for C-comments.  To avoid recognizing "*ptr" as a comment, the middle string
includes the 'b' flag.  For three-piece comments, Vim checks the text after
the start and middle strings for the end string.  If Vim finds the end string,
the comment does not continue on the next line.  Three-piece comments must
have a middle string because otherwise Vim can't recognize the middle lines.

Notice the use of the "x" flag in the above three-piece comment definition.
When you hit Return in a C-comment, Vim will insert the middle comment leader
for the new line: " * ".  To close this comment you just have to type "/"
before typing anything else on the new line.  This will replace the
middle-comment leader with the end-comment leader and apply any specified
alignment, leaving just " */".  There is no need to hit Backspace first.

When there is a match with a middle part, but there also is a matching end
part which is longer, the end part is used.  This makes a C style comment work
without requiring the middle part to end with a space.

Here is an example of alignment flags at work to make a comment stand out
(kind of looks like a 1 too). Consider comment string: >
        :set comments=sr:/***,m:**,ex-2:******/
<
                                    /*** ~
                                      **<--right aligned from "r" flag ~
                                      ** ~
offset 2 spaces for the "-2" flag--->** ~
                                    ******/ ~
In this case, the first comment was typed, then return was pressed 4 times,
then "/" was pressed to end the comment.

Here are some finer points of three part comments. There are three times when
alignment and offset flags are taken into consideration: opening a new line
after a start-comment, opening a new line before an end-comment, and
automatically ending a three-piece comment.  The end alignment flag has a
backwards perspective; the result is that the same alignment flag used with
"s" and "e" will result in the same indent for the starting and ending pieces.
Only one alignment per comment part is meant to be used, but an offset number
will override the "r" and "l" flag.

Enabling 'cindent' will override the alignment flags in many cases.
```

```
Reindenting using a different method like |gq| or |=| will not consult
alignment flags either. The same behaviour can be defined in those other
formatting options. One consideration is that 'cindent' has additional options
for context based indenting of comments but cannot replicate many three piece
indent alignments.  However, 'indentexpr' has the ability to work better with
three piece comments.

Other examples: >
   "b:*"        Includes lines starting with "*", but not if the "*" is
                followed by a non-blank.  This avoids a pointer dereference
                like "*str" to be recognized as a comment.
   "n:>"        Includes a line starting with ">", ">>", ">>>", etc.
   "fb:-"       Format a list that starts with "- ".

By default, "b:#" is included.  This means that a line that starts with
"#include" is not recognized as a comment line.  But a line that starts with
"# define" is recognized.  This is a compromise.

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

                                                              *fo-table*
You can use the 'formatoptions' option  to influence how Vim formats text.
'formatoptions' is a string that can contain any of the letters below.  The
default setting is "tcq".  You can separate the option letters with commas for
readability.

letter   meaning when present in 'formatoptions'    ~

t       Auto-wrap text using textwidth
c       Auto-wrap comments using textwidth, inserting the current comment
        leader automatically.
r       Automatically insert the current comment leader after hitting
        <Enter> in Insert mode.
o       Automatically insert the current comment leader after hitting 'o' or
        'O' in Normal mode.
q       Allow formatting of comments with "gq".
        Note that formatting will not change blank lines or lines containing
        only the comment leader.  A new paragraph starts after such a line,
        or when the comment leader changes.
w       Trailing white space indicates a paragraph continues in the next line.
        A line that ends in a non-white character ends a paragraph.
a       Automatic formatting of paragraphs.  Every time text is inserted or
        deleted the paragraph will be reformatted.  See |auto-format|.
        When the 'c' flag is present this only happens for recognized
        comments.
n       When formatting text, recognize numbered lists.  This actually uses
        the 'formatlistpat' option, thus any kind of list can be used.  The
        indent of the text after the number is used for the next line.  The
        default is to find a number, optionally followed by '.', ':', ')',
        ']' or '}'.  Note that 'autoindent' must be set too.  Doesn't work
        well together with "2".
        Example: >
                1. the first item
                   wraps
                2. the second item
2       When formatting text, use the indent of the second line of a paragraph
        for the rest of the paragraph, instead of the indent of the first
        line.  This supports paragraphs in which the first line has a
        different indent than the rest.  Note that 'autoindent' must be set
        too.  Example: >
                        first line of a paragraph
                second line of the same paragraph
```

```
                third line.
<       This also works inside comments, ignoring the comment leader.
v       Vi-compatible auto-wrapping in insert mode: Only break a line at a
        blank that you have entered during the current insert command.  (Note:
        this is not 100% Vi compatible.  Vi has some "unexpected features" or
        bugs in this area.  It uses the screen column instead of the line
        column.)
b       Like 'v', but only auto-wrap if you enter a blank at or before
        the wrap margin.  If the line was longer than 'textwidth' when you
        started the insert, or you do not enter a blank in the insert before
        reaching 'textwidth', Vim does not perform auto-wrapping.
l       Long lines are not broken in insert mode: When a line was longer than
        'textwidth' when the insert command started, Vim does not
        automatically format it.
m       Also break at a multi-byte character above 255.  This is useful for
        Asian text where every character is a word on its own.
M       When joining lines, don't insert a space before or after a multi-byte
        character.  Overrules the 'B' flag.
B       When joining lines, don't insert a space between two multi-byte
        characters.  Overruled by the 'M' flag.
1       Don't break a line after a one-letter word.  It's broken before it
        instead (if possible).
j       Where it makes sense, remove a comment leader when joining lines.  For
        example, joining:
                int i;   // the index ~
                         // in the list ~
        Becomes:
                int i;   // the index in the list ~


With 't' and 'c' you can specify when Vim performs auto-wrapping:
value   action  ~
""      no automatic formatting (you can use "gq" for manual formatting)
"t"     automatic formatting of text, but not comments
"c"     automatic formatting for comments, but not text (good for C code)
"tc"    automatic formatting for text and comments

Note that when 'textwidth' is 0, Vim does no automatic formatting anyway (but
does insert comment leaders according to the 'comments' option).  An exception
is when the 'a' flag is present. |auto-format|

Note that when 'paste' is on, Vim does no formatting at all.

Note that 'textwidth' can be non-zero even if Vim never performs auto-wrapping;
'textwidth' is still useful for formatting with "gq".

If the 'comments' option includes "/*", "*" and/or "*/", then Vim has some
built in stuff to treat these types of comments a bit more cleverly.
Opening a new line before or after "/*" or "*/" (with 'r' or 'o' present in
'formatoptions') gives the correct start of the line automatically.  The same
happens with formatting and auto-wrapping.  Opening a line after a line
starting with "/*" or "*" and containing "*/", will cause no comment leader to
be inserted, and the indent of the new line is taken from the line containing
the start of the comment.
E.g.:
    /* ~
     * Your typical comment. ~
     */ ~
    The indent on this line is the same as the start of the above
    comment.

All of this should be really cool, especially in conjunction with the new
```

```
:autocmd command to prepare different settings for different types of file.

Some examples:
  for C code (only format comments): >
        :set fo=croq
< for Mail/news (format all, don't start comment with "o" command): >
        :set fo=tcrq
<

Automatic formatting                              *auto-format* *autoformat*

When the 'a' flag is present in 'formatoptions' text is formatted
automatically when inserting text or deleting text.  This works nice for
editing text paragraphs.  A few hints on how to use this:

- You need to properly define paragraphs.  The simplest is paragraphs that are
  separated by a blank line.  When there is no separating blank line, consider
  using the 'w' flag and adding a space at the end of each line in the
  paragraphs except the last one.

- You can set the 'formatoptions' based on the type of file |filetype| or
  specifically for one file with a |modeline|.

- Set 'formatoptions' to "aw2tq" to make text with indents like this:

             bla bla foobar bla
         bla foobar bla foobar bla
             bla bla foobar bla
         bla foobar bla bla foobar

- Add the 'c' flag to only auto-format comments.  Useful in source code.

- Set 'textwidth' to the desired width.  If it is zero then 79 is used, or the
  width of the screen if this is smaller.

And a few warnings:

- When part of the text is not properly separated in paragraphs, making
  changes in this text will cause it to be formatted anyway.  Consider doing >

        :set fo-=a

- When using the 'w' flag (trailing space means paragraph continues) and
  deleting the last line of a paragraph with |dd|, the paragraph will be
  joined with the next one.

- Changed text is saved for undo.  Formatting is also a change.  Thus each
  format action saves text for undo.  This may consume quite a lot of memory.

- Formatting a long paragraph and/or with complicated indenting may be slow.

==============================================================================
7. Sorting text                                             *sorting*

Vim has a sorting function and a sorting command.  The sorting function can be
found here: |sort()|, |uniq()|.

                                                        *:sor* *:sort*
:[range]sor[t][!] [b][f][i][n][o][r][u][x] [/{pattern}/]
                        Sort lines in [range].  When no range is given all
                        lines are sorted.
```

```
                    With [!] the order is reversed.

                    With [i] case is ignored.

                    Options [n][f][x][o][b] are mutually exclusive.

                    With [n] sorting is done on the first decimal number
                    in the line (after or inside a {pattern} match).
                    One leading '-' is included in the number.

                    With [f] sorting is done on the Float in the line.
                    The value of Float is determined similar to passing
                    the text (after or inside a {pattern} match) to
                    str2float() function. This option is available only
                    if Vim was compiled with Floating point support.

                    With [x] sorting is done on the first hexadecimal
                    number in the line (after or inside a {pattern}
                    match).  A leading "0x" or "0X" is ignored.
                    One leading '-' is included in the number.

                    With [o] sorting is done on the first octal number in
                    the line (after or inside a {pattern} match).

                    With [b] sorting is done on the first binary number in
                    the line (after or inside a {pattern} match).

                    With [u] (u stands for unique) only keep the first of
                    a sequence of identical lines (ignoring case when [i]
                    is used).  Without this flag, a sequence of identical
                    lines will be kept in their original order.
                    Note that leading and trailing white space may cause
                    lines to be different.

                    When /{pattern}/ is specified and there is no [r] flag
                    the text matched with {pattern} is skipped, so that
                    you sort on what comes after the match.
                    Instead of the slash any non-letter can be used.
                    For example, to sort on the second comma-separated
                    field: >
                            :sort /[^,]*,/
<                   To sort on the text at virtual column 10 (thus
                    ignoring the difference between tabs and spaces): >
                            :sort /.*\%10v/
<                   To sort on the first number in the line, no matter
                    what is in front of it: >
                            :sort /.\{-}\ze\d/
<                   (Explanation: ".\{-}" matches any text, "\ze" sets the
                    end of the match and \d matches a digit.)
                    With [r] sorting is done on the matching {pattern}
                    instead of skipping past it as described above.
                    For example, to sort on only the first three letters
                    of each line: >
                            :sort /\a\a\a/ r

<                   If a {pattern} is used, any lines which don't have a
                    match for {pattern} are kept in their current order,
                    but separate from the lines which do match {pattern}.
                    If you sorted in reverse, they will be in reverse
                    order after the sorted lines, otherwise they will be
                    in their original order, right before the sorted
                    lines.
```

                        If {pattern} is empty (e.g. // is specified), the
                        last search pattern is used. This allows trying out
                        a pattern first.

Note that using `:sort` with `:global` doesn't sort the matching lines, it's
quite useless.

The details about sorting depend on the library function used. There is no
guarantee that sorting obeys the current locale. You will have to try it out.
Vim does do a "stable" sort.

The sorting can be interrupted, but if you interrupt it too late in the
process you may end up with duplicated lines. This also depends on the system
library function used.

 vim:tw=78:ts=8:ft=help:norl:
indent.txt   For Vim version 8.0. Last change: 2014 Dec 06

                  VIM REFERENCE MANUAL   by Bram Moolenaar

This file is about indenting C programs and other files.

==
1. Indenting C style programs   *C-indenting*

The basics for C style indenting are explained in section |30.2| of the user
manual.

Vim has options for automatically indenting C style program files. Many
programming languages including Java and C++ follow very closely the
formatting conventions established with C. These options affect only the
indent and do not perform other formatting. There are additional options that
affect other kinds of formatting as well as indenting, see |format-comments|,
|fo-table|, |gq| and |formatting| for the main ones.

Note that this will not work when the |+smartindent| or |+cindent| features
have been disabled at compile time.

There are in fact four main methods available for indentation, each one
overrides the previous if it is enabled, or non-empty for 'indentexpr':
'autoindent'   uses the indent from the previous line.
'smartindent'   is like 'autoindent' but also recognizes some C syntax to
                increase/reduce the indent where appropriate.
'cindent'   Works more cleverly than the other two and is configurable to
                different indenting styles.
'indentexpr'   The most flexible of all: Evaluates an expression to compute
                the indent of a line. When non-empty this method overrides
                the other ones. See |indent-expression|.
The rest of this section describes the 'cindent' option.

Note that 'cindent' indenting does not work for every code scenario. Vim
is not a C compiler: it does not recognize all syntax. One requirement is
that toplevel functions have a '{' in the first column. Otherwise they are
easily confused with declarations.

These four options control C program indenting:

```
'cindent'       Enables Vim to perform C program indenting automatically.
'cinkeys'       Specifies which keys trigger reindenting in insert mode.
'cinoptions'    Sets your preferred indent style.
'cinwords'      Defines keywords that start an extra indent in the next line.

If 'lisp' is not on and 'equalprg' is empty, the "=" operator indents using
Vim's built-in algorithm rather than calling an external program.

See |autocommand| for how to set the 'cindent' option automatically for C code
files and reset it for others.

                                        *cinkeys-format* *indentkeys-format*
The 'cinkeys' option is a string that controls Vim's indenting in response to
typing certain characters or commands in certain contexts.  Note that this not
only triggers C-indenting.  When 'indentexpr' is not empty 'indentkeys' is
used instead.  The format of 'cinkeys' and 'indentkeys' is equal.

The default is "0{,0},0),:,0#,!^F,o,O,e" which specifies that indenting occurs
as follows:

        "0{"    if you type '{' as the first character in a line
        "0}"    if you type '}' as the first character in a line
        "0)"    if you type ')' as the first character in a line
        ":"     if you type ':' after a label or case statement
        "0#"    if you type '#' as the first character in a line
        "!^F"   if you type CTRL-F (which is not inserted)
        "o"     if you type a <CR> anywhere or use the "o" command (not in
                insert mode!)
        "O"     if you use the "O" command (not in insert mode!)
        "e"     if you type the second 'e' for an "else" at the start of a
                line

Characters that can precede each key:                           *i_CTRL-F*
!       When a '!' precedes the key, Vim will not insert the key but will
        instead reindent the current line.  This allows you to define a
        command key for reindenting the current line.  CTRL-F is the default
        key for this.  Be careful if you define CTRL-I for this because CTRL-I
        is the ASCII code for <Tab>.
*       When a '*' precedes the key, Vim will reindent the line before
        inserting the key.  If 'cinkeys' contains "*<Return>", Vim reindents
        the current line before opening a new line.
0       When a zero precedes the key (but appears after '!' or '*') Vim will
        reindent the line only if the key is the first character you type in
        the line.  When used before "=" Vim will only reindent the line if
        there is only white space before the word.

When neither '!' nor '*' precedes the key, Vim reindents the line after you
type the key.  So ';' sets the indentation of a line which includes the ';'.

Special key names:
<>      Angle brackets mean spelled-out names of keys.  For example: "<Up>",
        "<Ins>" (see |key-notation|).
^       Letters preceded by a caret (^) are control characters.  For example:
        "^F" is CTRL-F.
o       Reindent a line when you use the "o" command or when Vim opens a new
        line below the current one (e.g., when you type <Enter> in insert
        mode).
O       Reindent a line when you use the "O" command.
e       Reindent a line that starts with "else" when you type the second 'e'.
:       Reindent a line when a ':' is typed which is after a label or case
        statement.  Don't reindent for a ":" in "class::method" for C++.  To
        Reindent for any ":", use "<:>".
```

```
=word   Reindent when typing the last character of "word".  "word" may
        actually be part of another word.  Thus "=end" would cause reindenting
        when typing the "d" in "endif" or "endwhile".  But not when typing
        "bend".  Also reindent when completion produces a word that starts
        with "word".  "0=word" reindents when there is only white space before
        the word.
=~word  Like =word, but ignore case.

If you really want to reindent when you type 'o', 'O', 'e', '0', '<', '>',
'*', ':' or '!', use "<o>", "<O>", "<e>", "<0>", "<<>", "<>>", "<*>", "<:>" or
"<!>", respectively, for those keys.

For an emacs-style indent mode where lines aren't indented every time you
press <Enter> but only if you press <Tab>, I suggest:
        :set cinkeys=0{,0},:,0#,!<Tab>,!^F
You might also want to switch off 'autoindent' then.

Note: If you change the current line's indentation manually, Vim ignores the
cindent settings for that line.  This prevents vim from reindenting after you
have changed the indent by typing <BS>, <Tab>, or <Space> in the indent or
used CTRL-T or CTRL-D.

                                                    *cinoptions-values*
The 'cinoptions' option sets how Vim performs indentation.  The value after
the option character can be one of these (N is any number):
        N       indent N spaces
        -N      indent N spaces to the left
        Ns      N times 'shiftwidth' spaces
        -Ns     N times 'shiftwidth' spaces to the left

In the list below,
"N" represents a number of your choice (the number can be negative).  When
there is an 's' after the number, Vim multiplies the number by 'shiftwidth':
"1s" is 'shiftwidth', "2s" is two times 'shiftwidth', etc.  You can use a
decimal point, too: "-0.5s" is minus half a 'shiftwidth'.
The examples below assume a 'shiftwidth' of 4.
                                                            *cino->*
        >N    Amount added for "normal" indent.  Used after a line that should
              increase the indent (lines starting with "if", an opening brace,
              etc.).  (default 'shiftwidth').

                cino=               cino=>2             cino=>2s >
                  if (cond)           if (cond)           if (cond)
                  {                   {                   {
                      foo;                foo;                    foo;
                  }                   }                   }
<
                                                            *cino-e*
        eN    Add N to the prevailing indent inside a set of braces if the
              opening brace at the End of the line (more precise: is not the
              first character in a line).  This is useful if you want a
              different indent when the '{' is at the start of the line from
              when '{' is at the end of the line.  (default 0).

                cino=               cino=e2             cino=e-2 >
                  if (cond) {         if (cond) {         if (cond) {
                      foo;                foo;              foo;
                  }                   }                   }
                  else                else                else
                  {                   {                   {
                      bar;                bar;                bar;
                  }                   }                   }
```

```
<
                                                        *cino-n*
        nN    Add N to the prevailing indent for a statement after an "if",
              "while", etc., if it is NOT inside a set of braces.  This is
              useful if you want a different indent when there is no '{'
              before the statement from when there is a '{' before it.
              (default 0).

                cino=               cino=n2             cino=n-2 >
                  if (cond)           if (cond)           if (cond)
                      foo;                    foo;            foo;
                  else                else                else
                  {                   {                   {
                      bar;                bar;                bar;
                  }                   }                   }
<
                                                        *cino-f*
        fN    Place the first opening brace of a function or other block in
              column N.  This applies only for an opening brace that is not
              inside other braces and is at the start of the line.  What comes
              after the brace is put relative to this brace.  (default 0).

                cino=               cino=f.5s           cino=f1s >
                  func()              func()              func()
                  {                     {                   {
                      int foo;              int foo;            int foo;
<
                                                        *cino-{*
        {N    Place opening braces N characters from the prevailing indent.
              This applies only for opening braces that are inside other
              braces.  (default 0).

                cino=               cino={.5s           cino={1s >
                  if (cond)           if (cond)           if (cond)
                  {                     {                   {
                      foo;                foo;                foo;
<
                                                        *cino-}*
        }N    Place closing braces N characters from the matching opening
              brace.  (default 0).

                cino=               cino={2,}-0.5s      cino=}2 >
                  if (cond)           if (cond)           if (cond)
                  {                     {                   {
                      foo;                foo;                foo;
                  }                   }                     }
<
                                                        *cino-^*
        ^N    Add N to the prevailing indent inside a set of braces if the
              opening brace is in column 0.  This can specify a different
              indent for whole of a function (some may like to set it to a
              negative number).  (default 0).

                cino=               cino=^-2            cino=^-s >
                  func()              func()              func()
                  {                   {                   {
                      if (cond)         if (cond)         if (cond)
                      {                 {                 {
                          a = b;            a = b;            a = b;
                      }                 }                 }
                  }                   }                   }
<
```

cino-L

LN Controls placement of jump labels. If N is negative, the label will be placed at column 1. If N is non-negative, the indent of the label will be the prevailing indent minus N. (default -1).

```
  cino=               cino=L2             cino=Ls >
    func()              func()              func()
    {                   {                   {
        {                   {                   {
            stmt;               stmt;               stmt;
    LABEL:              LABEL:              LABEL:
        }                   }                   }
    }                   }                   }
<
```

cino-:

:N Place case labels N characters from the indent of the switch(). (default 'shiftwidth').

```
  cino=               cino=:0 >
    switch (x)          switch(x)
    {                   {
        case 1:         case 1:
            a = b;          a = b;
        default:        default:
    }                   }
<
```

cino-=

=N Place statements occurring after a case label N characters from the indent of the label. (default 'shiftwidth').

```
  cino=               cino==10 >
     case 11:            case 11:  a = a + 1;
         a = a + 1;                  b = b + 1;
<
```

cino-l

lN If N != 0 Vim will align with a case label instead of the statement after it in the same line.

```
  cino=                       cino=l1 >
      switch (a) {              switch (a) {
          case 1: {                 case 1: {
                      break;                    break;
                  }                 }
<
```

cino-b

bN If N != 0 Vim will align a final "break" with the case label, so that case..break looks like a sort of block. (default: 0). When using 1, consider adding "0=break" to 'cinkeys'.

```
  cino=               cino=b1 >
    switch (x)          switch(x)
    {                   {
        case 1:             case 1:
            a = b;              a = b;
            break;          break;

        default:            default:
            a = 0;              a = 0;
            break;          break;
    }                   }
<
```

cino-g

```
        gN    Place C++ scope declarations N characters from the indent of the
              block they are in.  (default 'shiftwidth').  A scope declaration
              can be "public:", "protected:" or "private:".

                cino=               cino=g0 >
                  {                   {
                      public:           public:
                          a = b;            a = b;
                      private:          private:
                  }                   }
<
                                                          *cino-h*
        hN    Place statements occurring after a C++ scope declaration N
              characters from the indent of the label.  (default
              'shiftwidth').

                cino=               cino=h10 >
                   public:              public:   a = a + 1;
                       a = a + 1;                b = b + 1;
<
                                                          *cino-N*
        NN    Indent inside C++ namespace N characters extra compared to a
              normal block.  (default 0).

                cino=                       cino=N-s >
                  namespace {                 namespace {
                      void function();        void function();
                  }                           }

                  namespace my                namespace my
                  {                           {
                      void function();        void function();
                  }                           }
<
                                                          *cino-E*
        EN    Indent inside C++ linkage specifications (extern "C" or
              extern "C++") N characters extra compared to a normal block.
              (default 0).

                cino=                       cino=E-s >
                  extern "C" {                extern "C" {
                      void function();        void function();
                  }                           }

                  extern "C"                  extern "C"
                  {                           {
                      void function();        void function();
                  }                           }
<
                                                          *cino-p*
        pN    Parameter declarations for K&R-style function declarations will
              be indented N characters from the margin.  (default
              'shiftwidth').

                cino=               cino=p0             cino=p2s >
                  func(a, b)          func(a, b)          func(a, b)
                      int a;          int a;                      int a;
                      char b;         char b;                     char b;
<
                                                          *cino-t*
        tN    Indent a function return type declaration N characters from the
              margin.  (default 'shiftwidth').
```

```
                cino=               cino=t0             cino=t7 >
                     int              int                      int
                  func()              func()               func()
<
                                                        *cino-i*
        iN    Indent C++ base class declarations and constructor
              initializations, if they start in a new line (otherwise they
              are aligned at the right side of the ':').
              (default 'shiftwidth').

                cino=                       cino=i0 >
                  class MyClass :             class MyClass :
                      public BaseClass        public BaseClass
                  {}                          {}
                  MyClass::MyClass() :        MyClass::MyClass() :
                      BaseClass(3)            BaseClass(3)
                  {}                          {}
<
                                                        *cino-+*
        +N    Indent a continuation line (a line that spills onto the next)
              inside a function N additional characters.  (default
              'shiftwidth').
              Outside of a function, when the previous line ended in a
              backslash, the 2 * N is used.

                cino=                       cino=+10 >
                  a = b + 9 *                 a = b + 9 *
                      c;                                c;
<
                                                        *cino-c*
        cN    Indent comment lines after the comment opener, when there is no
              other text with which to align, N characters from the comment
              opener.  (default 3).  See also |format-comments|.

                cino=                       cino=c5 >
                  /*                          /*
                     text.                          text.
                   */                          */
<
                                                        *cino-C*
        CN    When N is non-zero, indent comment lines by the amount specified
              with the c flag above even if there is other text behind the
              comment opener.  (default 0).

                cino=c0                     cino=c0,C1 >
                  /********                  /********
                    text.                     text.
                  ********/                   ********/
<             (Example uses ":set comments& comments-=s1:/* comments^=s0:/*")

                                                        *cino-/*
        /N    Indent comment lines N characters extra.  (default 0).
                cino=                       cino=/4 >
                  a = b;                      a = b;
                  /* comment */                   /* comment */
                  c = d;                      c = d;
<
                                                        *cino-(*
        (N    When in unclosed parentheses, indent N characters from the line
              with the unclosed parentheses.  Add a 'shiftwidth' for every
              unclosed parentheses.  When N is 0 or the unclosed parentheses
```

```
              is the first non-white character in its line, line up with the
              next non-white character after the unclosed parentheses.
              (default 'shiftwidth' * 2).

                cino=                       cino=(0 >
                  if (c1 && (c2 ||            if (c1 && (c2 ||
                             c3))                        c3))
                      foo;                        foo;
                  if (c1 &&                   if (c1 &&
                          (c2 || c3))             (c2 || c3))
                     {                           {
<
                                                          *cino-u*
        uN    Same as (N, but for one level deeper.  (default 'shiftwidth').

                cino=                       cino=u2 >
                  if (c123456789              if (c123456789
                          && (c22345                  && (c22345
                              || c3))                   || c3))
<
                                                          *cino-U*
        UN    When N is non-zero, do not ignore the indenting specified by
              ( or u in case that the unclosed parentheses is the first
              non-white character in its line.  (default 0).

                cino= or cino=(s            cino=(s,U1 >
                  c = c1 &&                   c = c1 &&
                      (                           (
                       c2 ||                       c2 ||
                       c3                          c3
                      ) && c4;                    ) && c4;
<
                                                          *cino-w*
        wN    When in unclosed parentheses and N is non-zero and either
              using "(0" or "u0", respectively, or using "U0" and the unclosed
              parentheses is the first non-white character in its line, line
              up with the character immediately after the unclosed parentheses
              rather than the first non-white character.  (default 0).

                cino=(0                     cino=(0,w1 >
                  if (   c1                   if (   c1
                         && (   c2                && (   c2
                                || c3))                  || c3))
                      foo;                        foo;
<
                                                          *cino-W*
        WN    When in unclosed parentheses and N is non-zero and either
              using "(0" or "u0", respectively and the unclosed parentheses is
              the last non-white character in its line and it is not the
              closing parentheses, indent the following line N characters
              relative to the outer context (i.e. start of the line or the
              next unclosed parentheses).  (default: 0).

                cino=(0                      cino=(0,W4 >
                  a_long_line(                a_long_line(
                              argument,                   argument,
                              argument);                  argument);
                  a_short_line(argument,      a_short_line(argument,
                               argument);                  argument);
<
                                                          *cino-k*
        kN    When in unclosed parentheses which follow "if", "for" or
```

```
            "while" and N is non-zero, overrides the behaviour defined by
            "(N": causes the indent to be N characters relative to the outer
            context (i.e. the line where "if", "for" or "while" is).  Has
            no effect on deeper levels of nesting.  Affects flags like "wN"
            only for the "if", "for" and "while" conditions.  If 0, defaults
            to behaviour defined by the "(N" flag.  (default: 0).

              cino=(0                    cino=(0,ks >
                if (condition1            if (condition1
                    && condition2)                && condition2)
                    action();                 action();
                function(argument1        function(argument1
                         && argument2);            && argument2);
<
                                                        *cino-m*
        mN    When N is non-zero, line up a line starting with a closing
              parentheses with the first character of the line with the
              matching opening parentheses.  (default 0).

              cino=(s                   cino=(s,m1 >
                c = c1 && (               c = c1 && (
                    c2 ||                     c2 ||
                    c3                        c3
                    ) && c4;                ) && c4;
                if (                      if (
                    c1 && c2                  c1 && c2
                   )                        )
                    foo;                      foo;
<
                                                        *cino-M*
        MN    When N is non-zero, line up a line starting with a closing
              parentheses with the first character of the previous line.
              (default 0).

              cino=                     cino=M1 >
                if (cond1 &&              if (cond1 &&
                       cond2                     cond2
                   )                         )
<
                                *java-cinoptions* *java-indenting* *cino-j*
        jN    Indent Java anonymous classes correctly.  Also works well for
              Javascript.  The value 'N' is currently unused but must be
              non-zero (e.g. 'j1').  'j1' will indent for example the
              following code snippet correctly: >

              object.add(new ChangeListener() {
                  public void stateChanged(ChangeEvent e) {
                      do_something();
                  }
              });
<
                        *javascript-cinoptions* *javascript-indenting* *cino-J*
        JN    Indent JavaScript object declarations correctly by not confusing
              them with labels.  The value 'N' is currently unused but must be
              non-zero (e.g. 'J1').  If you enable this you probably also want
              to set |cino-j|. >

              var bar = {
                  foo: {
                      that: this,
                      some: ok,
                  },
```

```
                    "bar":{
                        a : 2,
                        b: "123abc",
                        x: 4,
                        "y": 5
                    }
                }
<
```

cino-)

)N Vim searches for unclosed parentheses at most N lines away. This limits the time needed to search for parentheses. (default 20 lines).

cino-star

N Vim searches for unclosed comments at most N lines away. This limits the time needed to search for the start of a comment. If your / */ comments stop indenting after N lines this is the value you will want to change. (default 70 lines).

cino-#

#N When N is non-zero recognize shell/Perl comments starting with '#', do not recognize preprocessor lines; allow right-shifting lines that start with "#". When N is zero (default): don't recognize '#' comments, do recognize preprocessor lines; right-shifting lines that start with "#" does not work.

The defaults, spelled out in full, are:

```
        cinoptions=>s,e0,n0,f0,{0,}0,^0,L-1,:s,=s,l0,b0,gs,hs,N0,E0,ps,ts,is,+s,
                        c3,C0,/0,(2s,us,U0,w0,W0,k0,m0,j0,J0,)20,*70,#0
```

Vim puts a line in column 1 if:

- It starts with '#' (preprocessor directives), if 'cinkeys' contains '#0'.
- It starts with a label (a keyword followed by ':', other than "case" and "default") and 'cinoptions' does not contain an 'L' entry with a positive value.
- Any combination of indentations causes the line to have less than 0 indentation.

==

2. Indenting by expression *indent-expression*

The basics for using flexible indenting are explained in section |30.3| of the user manual.

If you want to write your own indent file, it must set the 'indentexpr' option. Setting the 'indentkeys' option is often useful. See the $VIMRUNTIME/indent directory for examples.

REMARKS ABOUT SPECIFIC INDENT FILES ~

CLOJURE *ft-clojure-indent* *clojure-indent*

Clojure indentation differs somewhat from traditional Lisps, due in part to the use of square and curly brackets, and otherwise by community convention. These conventions are not universally followed, so the Clojure indent script offers a few configurable options, listed below.

```
If the current vim does not include searchpairpos(), the indent script falls
back to normal 'lisp' indenting, and the following options are ignored.

                                                        *g:clojure_maxlines*

Set maximum scan distance of searchpairpos(). Larger values trade performance
for correctness when dealing with very long forms. A value of 0 will scan
without limits.
>
        " Default
        let g:clojure_maxlines = 100
<
                                              *g:clojure_fuzzy_indent*
                                        *g:clojure_fuzzy_indent_patterns*
                                        *g:clojure_fuzzy_indent_blacklist*

The 'lispwords' option is a list of comma-separated words that mark special
forms whose subforms must be indented with two spaces.

For example:
>
        (defn bad []
              "Incorrect indentation")

        (defn good []
          "Correct indentation")
<
If you would like to specify 'lispwords' with a |pattern| instead, you can use
the fuzzy indent feature:
>
        " Default
        let g:clojure_fuzzy_indent = 1
        let g:clojure_fuzzy_indent_patterns = ['^with', '^def', '^let']
        let g:clojure_fuzzy_indent_blacklist =
                \ ['-fn$', '\v^with-%(meta|out-str|loading-context)$']

        " Legacy comma-delimited string version; the list format above is
        " recommended. Note that patterns are implicitly anchored with ^ and $
        let g:clojure_fuzzy_indent_patterns = 'with.*,def.*,let.*'
<
|g:clojure_fuzzy_indent_patterns| and |g:clojure_fuzzy_indent_blacklist| are
|Lists| of patterns that will be matched against the unquoted, unqualified
symbol at the head of a list. This means that a pattern like "^foo" will match
all these candidates: "foobar", "my.ns/foobar", and "#'foobar".

Each candidate word is tested for special treatment in this order:

        1. Return true if word is literally in 'lispwords'
        2. Return false if word matches a pattern in
           |g:clojure_fuzzy_indent_blacklist|
        3. Return true if word matches a pattern in
           |g:clojure_fuzzy_indent_patterns|
        4. Return false and indent normally otherwise

                                        *g:clojure_special_indent_words*

Some forms in Clojure are indented so that every subform is indented only two
spaces, regardless of 'lispwords'. If you have a custom construct that should
be indented in this idiosyncratic fashion, you can add your symbols to the
default list below.
>
        " Default
```

```
        let g:clojure_special_indent_words =
           \ 'deftype,defrecord,reify,proxy,extend-type,extend-protocol,letfn'
<
                                          *g:clojure_align_multiline_strings*

Align subsequent lines in multiline strings to the column after the opening
quote, instead of the same column.

For example:
>
        (def default
          "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
          eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
          enim ad minim veniam, quis nostrud exercitation ullamco laboris
          nisi ut aliquip ex ea commodo consequat.")

        (def aligned
          "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
           eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
           enim ad minim veniam, quis nostrud exercitation ullamco laboris
           nisi ut aliquip ex ea commodo consequat.")
<
This option is off by default.
>
        " Default
        let g:clojure_align_multiline_strings = 0
<
                                                *g:clojure_align_subforms*

By default, parenthesized compound forms that look like function calls and
whose head subform is on its own line have subsequent subforms indented by
two spaces relative to the opening paren:
>
        (foo
          bar
          baz)
<
Setting this option changes this behavior so that all subforms are aligned to
the same column, emulating the default behavior of clojure-mode.el:
>
        (foo
         bar
         baz)
<
This option is off by default.
>
        " Default
        let g:clojure_align_subforms = 0
<

FORTRAN                                                  *ft-fortran-indent*

Block if, select case, where, and forall constructs are indented.  So are
type, interface, associate, block, and enum constructs.  The indenting of
subroutines, functions, modules, and program blocks is optional.  Comments,
labelled statements and continuation lines are indented if the Fortran is in
free source form, whereas they are not indented if the Fortran is in fixed
source form because of the left margin requirements.  Hence manual indent
corrections will be necessary for labelled statements and continuation lines
when fixed source form is being used.  For further discussion of the method
used for the detection of source format see |ft-fortran-syntax|.
```

```
Do loops ~
All do loops are left unindented by default.  Do loops can be unstructured in
Fortran with (possibly multiple) loops ending on a labelled executable
statement of almost arbitrary type.  Correct indentation requires
compiler-quality parsing.  Old code with do loops ending on labelled statements
of arbitrary type can be indented with elaborate programs such as Tidy
(http://www.unb.ca/chem/ajit/f_tidy.htm).  Structured do/continue loops are
also left unindented because continue statements are also used for purposes
other than ending a do loop.  Programs such as Tidy can convert structured
do/continue loops to the do/enddo form.  Do loops of the do/enddo variety can
be indented.  If you use only structured loops of the do/enddo form, you should
declare this by setting the fortran_do_enddo variable in your .vimrc as
follows >

   let fortran_do_enddo=1

in which case do loops will be indented.  If all your loops are of do/enddo
type only in, say, .f90 files, then you should set a buffer flag with an
autocommand such as >

  au! BufRead,BufNewFile *.f90 let b:fortran_do_enddo=1

to get do loops indented in .f90 files and left alone in Fortran files with
other extensions such as .for.

Program units ~
The indenting of program units (subroutines, functions, modules, and program
blocks) is enabled by default but can be suppressed if a lighter, screen-width
preserving indent style is desired.  To suppress the indenting of program
units for all fortran files set the global fortran_indent_less variable in
your .vimrc as follows >

  let fortran_indent_less=1

A finer level of suppression can be achieved by setting the corresponding
buffer-local variable as follows >

  let b:fortran_indent_less=1


HTML                              *ft-html-indent* *html-indent* *html-indenting*

This is about variables you can set in your vimrc to customize HTML indenting.

You can set the indent for the first line after <script> and <style>
"blocktags" (default "zero"): >

      :let g:html_indent_script1 = "inc"
      :let g:html_indent_style1 = "inc"
<
      VALUE     MEANING ~
      "zero"    zero indent
      "auto"    auto indent (same indent as the blocktag)
      "inc"     auto indent + one indent step

Many tags increase the indent for what follows per default (see "Add Indent
Tags" in the script).  You can add further tags with: >

      :let g:html_indent_inctags = "html,body,head,tbody"

You can also remove such tags with: >
```

```
      :let g:html_indent_autotags = "th,td,tr,tfoot,thead"

Default value is empty for both variables.  Note: the initial "inctags" are
only defined once per Vim session.

User variables are only read when the script is sourced.  To enable your
changes during a session, without reloading the HTML file, you can manually
do: >

      :call HtmlIndent_CheckUserSettings()

Detail:
  Calculation of indent inside "blocktags" with "alien" content:
      BLOCKTAG   INDENT EXPR        WHEN APPLICABLE ~
      <script> : {customizable}     if first line of block
               : cindent(v:lnum)    if attributes empty or contain "java"
               : -1                 else (vbscript, tcl, ...)
      <style>  : {customizable}     if first line of block
               : GetCSSIndent()     else
      <!-- --> : -1


PHP                              *ft-php-indent* *php-indent* *php-indenting*

NOTE:   PHP files will be indented correctly only if PHP |syntax| is active.

If you are editing a file in Unix 'fileformat' and '\r' characters are present
before new lines, indentation won't proceed correctly ; you have to remove
those useless characters first with a command like: >

    :%s /\r$//g

Or, you can simply |:let| the variable PHP_removeCRwhenUnix to 1 and the
script will silently remove them when Vim loads a PHP file (at each |BufRead|).

OPTIONS: ~

PHP indenting can be altered in several ways by modifying the values of some
global variables:

                                          *php-comment* *PHP_autoformatcomment*
To not enable auto-formatting of comments by default (if you want to use your
own 'formatoptions'): >
    :let g:PHP_autoformatcomment = 0

Else, 't' will be removed from the 'formatoptions' string and "qrowcb" will be
added, see |fo-table| for more information.
-------------

                                                        *PHP_outdentSLComments*
To add extra indentation to single-line comments: >
    :let g:PHP_outdentSLComments = N

With N being the number of 'shiftwidth' to add.

Only single-line comments will be affected such as: >
    # Comment
    // Comment
    /* Comment */
-------------

                                                        *PHP_default_indenting*
```

```
To add extra indentation to every PHP lines with N being the number of
'shiftwidth' to add: >
    :let g:PHP_default_indenting = N

For example, with N = 1, this will give:
>
    <?php
        if (!isset($History_lst_sel))
            if (!isset($History_lst_sel))
                if (!isset($History_lst_sel)) {
                    $History_lst_sel=0;
                } else
                    $foo="bar";

        $command_hist = TRUE;
    ?>
(Notice the extra indentation between the PHP container markers and the code)
-------------

                                                        *PHP_outdentphpescape*
To indent PHP escape tags as the surrounding non-PHP code (only affects the
PHP escape tags): >
:let g:PHP_outdentphpescape = 0
-------------

                                                        *PHP_removeCRwhenUnix*
To automatically remove '\r' characters when the 'fileformat' is set to Unix: >
    :let g:PHP_removeCRwhenUnix = 1
-------------

                                                        *PHP_BracesAtCodeLevel*
To indent braces at the same level than the code they contain: >
    :let g:PHP_BracesAtCodeLevel = 1

This will give the following result: >
    if ($foo)
        {
        foo();
        }
Instead of: >
    if ($foo)
    {
        foo();
    }

NOTE:   Indenting will be a bit slower if this option is used because some
        optimizations won't be available.
-------------


*PHP_vintage_case_default_indent*
To indent 'case:' and 'default:' statements in switch() blocks: >
    :let g:PHP_vintage_case_default_indent = 1

In PHP braces are not required inside 'case/default' blocks therefore 'case:'
and 'default:' are indented at the same level than the 'switch()' to avoid
meaningless indentation. You can use the above option to return to the
traditional way.


PYTHON                                                  *ft-python-indent*
```

```
The amount of indent can be set for the following situations.  The examples
given are the defaults.  Note that the variables are set to an expression, so
that you can change the value of 'shiftwidth' later.

Indent after an open paren: >
        let g:pyindent_open_paren = '&sw * 2'
Indent after a nested paren: >
        let g:pyindent_nested_paren = '&sw'
Indent for a continuation line: >
        let g:pyindent_continue = '&sw * 2'


R                                                                  *ft-r-indent*

Function arguments are aligned if they span for multiple lines. If you prefer
do not have the arguments of functions aligned, put in your |vimrc|:
>
   let r_indent_align_args = 0
<
All lines beginning with a comment character, #, get the same indentation
level of the normal R code. Users of Emacs/ESS may be used to have lines
beginning with a single # indented in the 40th column, ## indented as R code,
and ### not indented. If you prefer that lines beginning with comment
characters are aligned as they are by Emacs/ESS, put in your |vimrc|:
>
   let r_indent_ess_comments = 1
<
If you prefer that lines beginning with a single # are aligned at a column
different from the 40th one, you should set a new value to the variable
r_indent_comment_column, as in the example below:
>
   let r_indent_comment_column = 30
<
Any code after a line that ends with "<-" is indented. Emacs/ESS does not
indent the code if it is a top level function. If you prefer that the
Vim-R-plugin behaves like Emacs/ESS in this regard, put in your |vimrc|:
>
   let r_indent_ess_compatible = 1
<
Below is an example of indentation with and without this option enabled:
>
   ### r_indent_ess_compatible = 1           ### r_indent_ess_compatible = 0
   foo <-                                    foo <-
       function(x)                               function(x)
   {                                         {
       paste(x)                                      paste(x)
   }                                         }
<

SHELL                                                        *ft-sh-indent*

The amount of indent applied under various circumstances in a shell file can
be configured by setting the following keys in the |Dictionary|
b:sh_indent_defaults to a specific amount or to a |Funcref| that references a
function that will return the amount desired:

b:sh_indent_options['default']  Default amount of indent.

b:sh_indent_options['continuation-line']
                                Amount of indent to add to a continued line.

b:sh_indent_options['case-labels']
```

```
                                Amount of indent to add for case labels.
                                (not actually implemented)

b:sh_indent_options['case-statements']
                                Amount of indent to add for case statements.

b:sh_indent_options['case-breaks']
                                Amount of indent to add (or more likely
                                remove) for case breaks.

VERILOG                                                 *ft-verilog-indent*

General block statements such as if, for, case, always, initial, function,
specify and begin, etc., are indented.  The module block statements (first
level blocks) are not indented by default.  you can turn on the indent with
setting a variable in the .vimrc as follows: >

  let b:verilog_indent_modules = 1

then the module blocks will be indented.  To stop this, remove the variable: >

  :unlet b:verilog_indent_modules

To set the variable only for Verilog file.  The following statements can be
used: >

  au BufReadPost * if exists("b:current_syntax")
  au BufReadPost *   if b:current_syntax == "verilog"
  au BufReadPost *     let b:verilog_indent_modules = 1
  au BufReadPost *   endif
  au BufReadPost * endif

Furthermore, setting the variable b:verilog_indent_width to change the
indenting width (default is 'shiftwidth'): >

  let b:verilog_indent_width = 4
  let b:verilog_indent_width = &sw * 2

In addition, you can turn the verbose mode for debug issue: >

  let b:verilog_indent_verbose = 1

Make sure to do ":set cmdheight=2" first to allow the display of the message.


VHDL                                                    *ft-vhdl-indent*

Alignment of generic/port mapping statements are performed by default. This
causes the following alignment example: >

  ENTITY sync IS
  PORT (
         clk        : IN  STD_LOGIC;
         reset_n    : IN  STD_LOGIC;
         data_input : IN  STD_LOGIC;
         data_out   : OUT STD_LOGIC
       );
  END ENTITY sync;

To turn this off, add >

  let g:vhdl_indent_genportmap = 0
```

```
to the .vimrc file, which causes the previous alignment example to change: >

  ENTITY sync IS
  PORT (
    clk        : IN  STD_LOGIC;
    reset_n    : IN  STD_LOGIC;
    data_input : IN  STD_LOGIC;
    data_out   : OUT STD_LOGIC
  );
  END ENTITY sync;

----------------------------------------

Alignment of right-hand side assignment "<=" statements are performed by
default. This causes the following alignment example: >

  sig_out <= (bus_a(1) AND
              (sig_b OR sig_c)) OR
              (bus_a(0) AND sig_d);

To turn this off, add >

  let g:vhdl_indent_rhsassign = 0

to the .vimrc file, which causes the previous alignment example to change: >

  sig_out <= (bus_a(1) AND
    (sig_b OR sig_c)) OR
    (bus_a(0) AND sig_d);

----------------------------------------

Full-line comments (lines that begin with "--") are indented to be aligned with
the very previous line's comment, PROVIDED that a whitespace follows after
"--".

For example: >

  sig_a <= sig_b; -- start of a comment
                  -- continuation of the comment
                  -- more of the same comment

While in Insert mode, after typing "-- " (note the space " "), hitting CTRL-F
will align the current "-- " with the previous line's "--".

If the very previous line does not contain "--", THEN the full-line comment
will be aligned with the start of the next non-blank line that is NOT a
full-line comment.

Indenting the following code: >

  sig_c <= sig_d; -- comment 0
         -- comment 1
                -- comment 2
    --debug_code:
    --PROCESS(debug_in)
         --BEGIN
            --  FOR i IN 15 DOWNTO 0 LOOP
             --    debug_out(8*i+7 DOWNTO 8*i) <= debug_in(15-i);
            --  END LOOP;
     --END PROCESS debug_code;
```

```
      -- comment 3
  sig_e <= sig_f; -- comment 4
           -- comment 5

results in: >

  sig_c <= sig_d; -- comment 0
                  -- comment 1
                  -- comment 2
  --debug_code:
  --PROCESS(debug_in)
  --BEGIN
  --  FOR i IN 15 DOWNTO 0 LOOP
  --    debug_out(8*i+7 DOWNTO 8*i) <= debug_in(15-i);
  --  END LOOP;
  --END PROCESS debug_code;

  -- comment 3
  sig_e <= sig_f; -- comment 4
                  -- comment 5

Notice that "--debug_code:" does not align with "-- comment 2"
because there is no whitespace that follows after "--" in "--debug_code:".

Given the dynamic nature of indenting comments, indenting should be done TWICE.
On the first pass, code will be indented. On the second pass, full-line
comments will be indented according to the correctly indented code.


VIM                                                     *ft-vim-indent*

For indenting Vim scripts there is one variable that specifies the amount of
indent for a continuation line, a line that starts with a backslash: >

        :let g:vim_indent_cont = &sw * 3

Three times shiftwidth is the default value.


 vim:tw=78:ts=8:ft=help:norl:
*undo.txt*      For Vim version 8.0.  Last change: 2014 May 24


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Undo and redo                                           *undo-redo*

The basics are explained in section |02.5| of the user manual.

1. Undo and redo commands       |undo-commands|
2. Two ways of undo             |undo-two-ways|
3. Undo blocks                  |undo-blocks|
4. Undo branches                |undo-branches|
5. Undo persistence             |undo-persistence|
6. Remarks about undo           |undo-remarks|

==============================================================================
1. Undo and redo commands                               *undo-commands*

<Undo>          or                                      *undo* *<Undo>* *u*
```

```
u                       Undo [count] changes.  {Vi: only one level}

                                                        *:u* *:un* *:undo*
:u[ndo]                 Undo one change.  {Vi: only one level}
                                                                  *E830*
:u[ndo] {N}             Jump to after change number {N}.  See |undo-branches|
                        for the meaning of {N}.  {not in Vi}

                                                        *CTRL-R*
CTRL-R                  Redo [count] changes which were undone.  {Vi: redraw
                        screen}

                                                        *:red* *:redo* *redo*
:red[o]                 Redo one change which was undone.  {Vi: no redo}

                                                        *U*
U                       Undo all latest changes on one line, the line where
                        the latest change was made. |U| itself also counts as
                        a change, and thus |U| undoes a previous |U|.
                        {Vi: while not moved off of the last modified line}
```

The last changes are remembered. You can use the undo and redo commands above to revert the text to how it was before each change. You can also apply the changes again, getting back the text before the undo.

The "U" command is treated by undo/redo just like any other command. Thus a "u" command undoes a "U" command and a 'CTRL-R' command redoes it again. When mixing "U", "u" and 'CTRL-R' you will notice that the "U" command will restore the situation of a line to before the previous "U" command. This may be confusing. Try it out to get used to it.
The "U" command will always mark the buffer as changed. When "U" changes the buffer back to how it was without changes, it is still considered changed. Use "u" to undo changes until the buffer becomes unchanged.

==
2. Two ways of undo *undo-two-ways*

How undo and redo commands work depends on the 'u' flag in 'cpoptions'. There is the Vim way ('u' excluded) and the Vi-compatible way ('u' included). In the Vim way, "uu" undoes two changes. In the Vi-compatible way, "uu" does nothing (undoes an undo).

'u' excluded, the Vim way:
You can go back in time with the undo command. You can then go forward again with the redo command. If you make a new change after the undo command, the redo will not be possible anymore.

'u' included, the Vi-compatible way:
The undo command undoes the previous change, and also the previous undo command. The redo command repeats the previous undo command. It does NOT repeat a change command, use "." for that.

Examples	Vim way	Vi-compatible way ~
"uu"	two times undo	no-op
"u CTRL-R"	no-op	two times undo

Rationale: Nvi uses the "." command instead of CTRL-R. Unfortunately, this is not Vi compatible. For example "dwdwu." in Vi deletes two words, in Nvi it does nothing.

==
3. Undo blocks *undo-blocks*

One undo command normally undoes a typed command, no matter how many changes
that command makes. This sequence of undo-able changes forms an undo block.
Thus if the typed key(s) call a function, all the commands in the function are
undone together.

If you want to write a function or script that doesn't create a new undoable
change but joins in with the previous change use this command:

:undoj *:undojoin* *E790*
:undoj[oin] Join further changes with the previous undo block.
Warning: Use with care, it may prevent the user from
properly undoing changes. Don't use this after undo
or redo.
{not in Vi}

This is most useful when you need to prompt the user halfway through a change.
For example in a function that calls |getchar()|. Do make sure that there was
a related change before this that you must join with.

This doesn't work by itself, because the next key press will start a new
change again. But you can do something like this: >

```
:undojoin | delete
```

After this an "u" command will undo the delete command and the previous
change.

To do the opposite, break a change into two undo blocks, in Insert mode use
CTRL-G u. This is useful if you want an insert command to be undoable in
parts. E.g., for each sentence. |i_CTRL-G_u|
Setting the value of 'undolevels' also breaks undo. Even when the new value
is equal to the old value.

==========
4. Undo branches *undo-branches* *undo-tree*

Above we only discussed one line of undo/redo. But it is also possible to
branch off. This happens when you undo a few changes and then make a new
change. The undone changes become a branch. You can go to that branch with
the following commands.

This is explained in the user manual: |usr_32.txt|.

:undol *:undolist*
:undol[ist] List the leafs in the tree of changes. Example:

```
   number changes  when               saved ~
       88      88  2010/01/04 14:25:53
      108     107  08/07 12:47:51
      136      46  13:33:01             7
      166     164  3 seconds ago
```

The "number" column is the change number. This number
continuously increases and can be used to identify a
specific undo-able change, see |:undo|.
The "changes" column is the number of changes to this
leaf from the root of the tree.
The "when" column is the date and time when this
change was made. The four possible formats are:

```
    N seconds ago
    HH:MM:SS             hour, minute, seconds
    MM/DD HH:MM:SS       idem, with month and day
```

```
                        YYYY/MM/DD HH:MM:SS  idem, with year
                    The "saved" column specifies, if this change was
                    written to disk and which file write it was. This can
                    be used with the |:later| and |:earlier| commands.
                    For more details use the |undotree()| function.

                                                        *g-*
g-                  Go to older text state.  With a count repeat that many
                    times.  {not in Vi}
                                                        *:ea* *:earlier*
:earlier {count}    Go to older text state {count} times.
:earlier {N}s       Go to older text state about {N} seconds before.
:earlier {N}m       Go to older text state about {N} minutes before.
:earlier {N}h       Go to older text state about {N} hours before.
:earlier {N}d       Go to older text state about {N} days before.

:earlier {N}f       Go to older text state {N} file writes before.
                    When changes were made since the last write
                    ":earlier 1f" will revert the text to the state when
                    it was written.  Otherwise it will go to the write
                    before that.
                    When at the state of the first file write, or when
                    the file was not written, ":earlier 1f" will go to
                    before the first change.

                                                        *g+*
g+                  Go to newer text state.  With a count repeat that many
                    times.  {not in Vi}
                                                        *:lat* *:later*
:later {count}      Go to newer text state {count} times.
:later {N}s         Go to newer text state about {N} seconds later.
:later {N}m         Go to newer text state about {N} minutes later.
:later {N}h         Go to newer text state about {N} hours later.
:later {N}d         Go to newer text state about {N} days later.

:later {N}f         Go to newer text state {N} file writes later.
                    When at the state of the last file write, ":later 1f"
                    will go to the newest text state.


Note that text states will become unreachable when undo information is cleared
for 'undolevels'.

Don't be surprised when moving through time shows multiple changes to take
place at a time.  This happens when moving through the undo tree and then
making a new change.

EXAMPLE

Start with this text:
        one two three ~

Delete the first word by pressing "x" three times:
        ne two three ~
        e two three ~
         two three ~

Now undo that by pressing "u" three times:
        e two three ~
        ne two three ~
        one two three ~
```

```
Delete the second word by pressing "x" three times:
        one wo three ~
        one o three ~
        one  three ~

Now undo that by using "g-" three times:
        one o three ~
        one wo three ~
         two three ~

You are now back in the first undo branch, after deleting "one".  Repeating
"g-" will now bring you back to the original text:
        e two three ~
        ne two three ~
        one two three ~

Jump to the last change with ":later 1h":
        one  three ~

And back to the start again with ":earlier 1h":
        one two three ~


Note that using "u" and CTRL-R will not get you to all possible text states
while repeating "g-" and "g+" does.

==============================================================================
5. Undo persistence             *undo-persistence* *persistent-undo*

When unloading a buffer Vim normally destroys the tree of undos created for
that buffer.  By setting the 'undofile' option, Vim will automatically save
your undo history when you write a file and restore undo history when you edit
the file again.

The 'undofile' option is checked after writing a file, before the BufWritePost
autocommands.  If you want to control what files to write undo information
for, you can use a BufWritePre autocommand: >
        au BufWritePre /tmp/* setlocal noundofile

Vim saves undo trees in a separate undo file, one for each edited file, using
a simple scheme that maps filesystem paths directly to undo files. Vim will
detect if an undo file is no longer synchronized with the file it was written
for (with a hash of the file contents) and ignore it when the file was changed
after the undo file was written, to prevent corruption.  An undo file is also
ignored if its owner differs from the owner of the edited file, except when
the owner of the undo file is the current user.  Set 'verbose' to get a
message about that when opening a file.

Undo files are normally saved in the same directory as the file.  This can be
changed with the 'undodir' option.

When the file is encrypted, the text in the undo file is also crypted.  The
same key and method is used. |encryption|

You can also save and restore undo histories by using ":wundo" and ":rundo"
respectively:
                                                        *:wundo* *:rundo*
:wundo[!] {file}
                Write undo history to {file}.
                When {file} exists and it does not look like an undo file
                (the magic number at the start of the file is wrong), then
                this fails, unless the ! was added.
```

```
                If it exists and does look like an undo file it is
                overwritten. If there is no undo-history, nothing will be
                written.
                Implementation detail: Overwriting happens by first deleting
                the existing file and then creating a new file with the same
                name. So it is not possible to overwrite an existing undofile
                in a write-protected directory.
                {not in Vi}

:rundo {file}   Read undo history from {file}.
                {not in Vi}

You can use these in autocommands to explicitly specify the name of the
history file.  E.g.: >

        au BufReadPost * call ReadUndo()
        au BufWritePost * call WriteUndo()
        func ReadUndo()
          if filereadable(expand('%:h'). '/UNDO/' . expand('%:t'))
            rundo %:h/UNDO/%:t
          endif
        endfunc
        func WriteUndo()
          let dirname = expand('%:h') . '/UNDO'
          if !isdirectory(dirname)
            call mkdir(dirname)
          endif
          wundo %:h/UNDO/%:t
        endfunc

You should keep 'undofile' off, otherwise you end up with two undo files for
every write.

You can use the |undofile()| function to find out the file name that Vim would
use.

Note that while reading/writing files and 'undofile' is set most errors will
be silent, unless 'verbose' is set.  With :wundo and :rundo you will get more
error messages, e.g., when the file cannot be read or written.

NOTE: undo files are never deleted by Vim.  You need to delete them yourself.

Reading an existing undo file may fail for several reasons:
*E822*  It cannot be opened, because the file permissions don't allow it.
*E823*  The magic number at the start of the file doesn't match.  This usually
        means it is not an undo file.
*E824*  The version number of the undo file indicates that it's written by a
        newer version of Vim.  You need that newer version to open it.  Don't
        write the buffer if you want to keep the undo info in the file.
"File contents changed, cannot use undo info"
        The file text differs from when the undo file was written.  This means
        the undo file cannot be used, it would corrupt the text.  This also
        happens when 'encoding' differs from when the undo file was written.
*E825*  The undo file does not contain valid contents and cannot be used.
*E826*  The undo file is encrypted but decryption failed.
*E827*  The undo file is encrypted but this version of Vim does not support
        encryption.  Open the file with another Vim.
*E832*  The undo file is encrypted but 'key' is not set, the text file is not
        encrypted.  This would happen if the text file was written by Vim
        encrypted at first, and later overwritten by not encrypted text.
        You probably want to delete this undo file.
"Not reading undo file, owner differs"
```

        The undo file is owned by someone else than the owner of the text
        file. For safety the undo file is not used.

Writing an undo file may fail for these reasons:
E828 The file to be written cannot be created. Perhaps you do not have
        write permissions in the directory.
"Cannot write undo file in any directory in 'undodir'"
        None of the directories in 'undodir' can be used.
"Will not overwrite with undo file, cannot read"
        A file exists with the name of the undo file to be written, but it
        cannot be read. You may want to delete this file or rename it.
"Will not overwrite, this is not an undo file"
        A file exists with the name of the undo file to be written, but it
        does not start with the right magic number. You may want to delete
        this file or rename it.
"Skipping undo file write, nothing to undo"
        There is no undo information to be written, nothing has been changed
        or 'undolevels' is negative.
E829 An error occurred while writing the undo file. You may want to try
        again.

==
6. Remarks about undo   *undo-remarks*

The number of changes that are remembered is set with the 'undolevels' option.
If it is zero, the Vi-compatible way is always used. If it is negative no
undo is possible. Use this if you are running out of memory.

                                                        clear-undo
When you set 'undolevels' to -1 the undo information is not immediately
cleared, this happens at the next change. To force clearing the undo
information you can use these commands: >
        :let old_undolevels = &undolevels
        :set undolevels=-1
        :exe "normal a \<BS>\<Esc>"
        :let &undolevels = old_undolevels
        :unlet old_undolevels

Marks for the buffer ('a to 'z) are also saved and restored, together with the
text. {Vi does this a little bit different}

When all changes have been undone, the buffer is not considered to be changed.
It is then possible to exit Vim with ":q" instead of ":q!" {not in Vi}. Note
that this is relative to the last write of the file. Typing "u" after ":w"
actually changes the buffer, compared to what was written, so the buffer is
considered changed then.

When manual |folding| is being used, the folds are not saved and restored.
Only changes completely within a fold will keep the fold as it was, because
the first and last line of the fold don't change.

The numbered registers can also be used for undoing deletes. Each time you
delete text, it is put into register "1. The contents of register "1 are
shifted to "2, etc. The contents of register "9 are lost. You can now get
back the most recent deleted text with the put command: '"1P'. (also, if the
deleted text was the result of the last delete or copy operation, 'P' or 'p'
also works as this puts the contents of the unnamed register). You can get
back the text of three deletes ago with '"3P'.

                                                redo-register
If you want to get back more than one part of deleted text, you can use a
special feature of the repeat command ".". It will increase the number of the

register used. So if you first do ""1P", the following "." will result in a
'"2P'. Repeating this will result in all numbered registers being inserted.

Example: If you deleted text with 'dd....' it can be restored with
                '"1P....'.

If you don't know in which register the deleted text is, you can use the
:display command. An alternative is to try the first register with '"1P', and
if it is not what you want do 'u.'. This will remove the contents of the
first put, and repeat the put command for the second register. Repeat the
'u.' until you got what you want.

 vim:tw=78:ts=8:ft=help:norl:
repeat.txt   For Vim version 8.0. Last change: 2017 Jun 10

                  VIM REFERENCE MANUAL   by Bram Moolenaar

Repeating commands, Vim scripts and debugging   *repeating*

Chapter 26 of the user manual introduces repeating |usr_26.txt|.

==
1. Single repeats   *single-repeat*

                                                        .
.   Repeat last change, with count replaced with [count].
                        Also repeat a yank command, when the 'y' flag is
                        included in 'cpoptions'. Does not repeat a
                        command-line command.

Simple changes can be repeated with the "." command. Without a count, the
count of the last change is used. If you enter a count, it will replace the
last one. |v:count| and |v:count1| will be set.

If the last change included a specification of a numbered register, the
register number will be incremented. See |redo-register| for an example how
to use this.

Note that when repeating a command that used a Visual selection, the same SIZE
of area is used, see |visual-repeat|.

                                                        @:
@:   Repeat last command-line [count] times.
                        {not available when compiled without the
                        |+cmdline_hist| feature}

==
2. Multiple repeats   *multi-repeat*

                                              :g *:global* *E148*

```
:[range]g[lobal]/{pattern}/[cmd]
                        Execute the Ex command [cmd] (default ":p") on the
                        lines within [range] where {pattern} matches.

:[range]g[lobal]!/{pattern}/[cmd]
                        Execute the Ex command [cmd] (default ":p") on the
                        lines within [range] where {pattern} does NOT match.

                                                        *:v* *:vglobal*
:[range]v[global]/{pattern}/[cmd]
                        Same as :g!.

Instead of the '/' which surrounds the {pattern}, you can use any other
single byte character, but not an alphabetic character, '\', '"' or '|'.
This is useful if you want to include a '/' in the search pattern or
replacement string.

For the definition of a pattern, see |pattern|.

NOTE [cmd] may contain a range; see |collapse| and |edit-paragraph-join| for
examples.

The global commands work by first scanning through the [range] lines and
marking each line where a match occurs (for a multi-line pattern, only the
start of the match matters).
In a second scan the [cmd] is executed for each marked line, as if the cursor
was in that line.  For ":v" and ":g!" the command is executed for each not
marked line.  If a line is deleted its mark disappears.
The default for [range] is the whole buffer (1,$).  Use "CTRL-C" to interrupt
the command.  If an error message is given for a line, the command for that
line is aborted and the global command continues with the next marked or
unmarked line.
                                                                  *E147*
When the command is used recursively, it only works on one line.  Giving a
range is then not allowed. This is useful to find all lines that match a
pattern and do not match another pattern: >
        :g/found/v/notfound/{cmd}
This first finds all lines containing "found", but only executes {cmd} when
there is no match for "notfound".

To execute a non-Ex command, you can use the `:normal` command: >
        :g/pat/normal {commands}
Make sure that {commands} ends with a whole command, otherwise Vim will wait
for you to type the rest of the command for each match.  The screen will not
have been updated, so you don't know what you are doing.  See |:normal|.

The undo/redo command will undo/redo the whole global command at once.
The previous context mark will only be set once (with "''" you go back to
where the cursor was before the global command).

The global command sets both the last used search pattern and the last used
substitute pattern (this is vi compatible).  This makes it easy to globally
replace a string:
        :g/pat/s//PAT/g
This replaces all occurrences of "pat" with "PAT".  The same can be done with:
        :%s/pat/PAT/g
Which is two characters shorter!

When using "global" in Ex mode, a special case is using ":visual" as a
command.  This will move to a matching line, go to Normal mode to let you
execute commands there until you use |Q| to return to Ex mode.  This will be
repeated for each matching line.  While doing this you cannot use ":global".
```

To abort this type CTRL-C twice.

==
3. Complex repeats *complex-repeat*

q *recording*

q{0-9a-zA-Z"} Record typed characters into register {0-9a-zA-Z"} (uppercase to append). The 'q' command is disabled while executing a register, and it doesn't work inside a mapping and |:normal|.

Note: If the register being used for recording is also used for |y| and |p| the result is most likely not what is expected, because the put will paste the recorded macro and the yank will overwrite the recorded macro. {Vi: no recording}

q Stops recording. (Implementation note: The 'q' that stops recording is not stored in the register, unless it was the result of a mapping) {Vi: no recording}

@

@{0-9a-z".=*+} Execute the contents of register {0-9a-z".=*+} [count] times. Note that register '%' (name of the current file) and '#' (name of the alternate file) cannot be used.
The register is executed like a mapping, that means that the difference between 'wildchar' and 'wildcharm' applies.
For "@=" you are prompted to enter an expression. The result of the expression is then executed.
See also |@:|. {Vi: only named registers}

@@ *E748*

@@ Repeat the previous @{0-9a-z":*} [count] times.

:[addr]*{0-9a-z".=+} *:@* *:star*
:[addr]@{0-9a-z".=*+} Execute the contents of register {0-9a-z".=*+} as an Ex command. First set cursor at line [addr] (default is current line). When the last line in the register does not have a <CR> it will be added automatically when the 'e' flag is present in 'cpoptions'.
Note that the ":*" command is only recognized when the '*' flag is present in 'cpoptions'. This is NOT the default when 'nocompatible' is used.
For ":@=" the last used expression is used. The result of evaluating the expression is executed as an Ex command.
Mappings are not recognized in these commands.
{Vi: only in some versions} Future: Will execute the register for each line in the address range.

:@:

:[addr]@: Repeat last command-line. First set cursor at line [addr] (default is current line). {not in Vi}

:[addr]@ *:@@*
:[addr]@@ Repeat the previous :@{0-9a-z"}. First set cursor at line [addr] (default is current line). {Vi: only in some versions}

==

```
4. Using Vim scripts                                    *using-scripts*

For writing a Vim script, see chapter 41 of the user manual |usr_41.txt|.

                                        *:so* *:source* *load-vim-script*
:so[urce] {file}        Read Ex commands from {file}.  These are commands that
                        start with a ":".
                        Triggers the |SourcePre| autocommand.

:so[urce]! {file}       Read Vim commands from {file}.  These are commands
                        that are executed from Normal mode, like you type
                        them.
                        When used after |:global|, |:argdo|, |:windo|,
                        |:bufdo|, in a loop or when another command follows
                        the display won't be updated while executing the
                        commands.
                        {not in Vi}

                                                        *:ru* *:runtime*
:ru[ntime][!] [where] {file} ..
                        Read Ex commands from {file} in each directory given
                        by 'runtimepath' and/or 'packpath'.  There is no error
                        for non-existing files.

                        Example: >
                                :runtime syntax/c.vim

<                       There can be multiple {file} arguments, separated by
                        spaces.  Each {file} is searched for in the first
                        directory from 'runtimepath', then in the second
                        directory, etc.  Use a backslash to include a space
                        inside {file} (although it's better not to use spaces
                        in file names, it causes trouble).

                        When [!] is included, all found files are sourced.
                        When it is not included only the first found file is
                        sourced.

                        When [where] is omitted only 'runtimepath' is used.
                        Other values:
                                START   search under "start" in 'packpath'
                                OPT     search under "opt" in 'packpath'
                                PACK    search under "start" and "opt" in
                                        'packpath'
                                ALL     first use 'runtimepath', then search
                                        under "start" and "opt" in 'packpath'

                        When {file} contains wildcards it is expanded to all
                        matching files.  Example: >
                                :runtime! plugin/*.vim
<                       This is what Vim uses to load the plugin files when
                        starting up.  This similar command: >
                                :runtime plugin/*.vim
<                       would source the first file only.

                        When 'verbose' is one or higher, there is a message
                        when no file could be found.
                        When 'verbose' is two or higher, there is a message
                        about each searched file.
                        {not in Vi}

                                                        *:pa* *:packadd* *E919*
```

```
:pa[ckadd][!] {name}    Search for an optional plugin directory in 'packpath'
                        and source any plugin files found.  The directory must
                        match:
                                pack/*/opt/{name} ~
                        The directory is added to 'runtimepath' if it wasn't
                        there yet.
                        If the directory pack/*/opt/{name}/after exists it is
                        added at the end of 'runtimepath'.

                        Note that {name} is the directory name, not the name
                        of the .vim file.  All the files matching the pattern
                                pack/*/opt/{name}/plugin/**/*.vim ~
                        will be sourced.  This allows for using subdirectories
                        below "plugin", just like with plugins in
                        'runtimepath'.

                        If the filetype detection was not enabled yet (this
                        is usually done with a "syntax enable" or "filetype
                        on" command in your .vimrc file), this will also look
                        for "{name}/ftdetect/*.vim" files.

                        When the optional ! is added no plugin files or
                        ftdetect scripts are loaded, only the matching
                        directories are added to 'runtimepath'.  This is
                        useful in your .vimrc.  The plugins will then be
                        loaded during initialization, see |load-plugins|.

                        Also see |pack-add|.

                                                *:packl* *:packloadall*
:packl[oadall][!]       Load all packages in the "start" directory under each
                        entry in 'packpath'.

                        First all the directories found are added to
                        'runtimepath', then the plugins found in the
                        directories are sourced.  This allows for a plugin to
                        depend on something of another plugin, e.g. an
                        "autoload" directory.  See |packload-two-steps| for
                        how this can be useful.

                        This is normally done automatically during startup,
                        after loading your .vimrc file.  With this command it
                        can be done earlier.

                        Packages will be loaded only once.  After this command
                        it won't happen again.  When the optional ! is added
                        this command will load packages even when done before.

                        An error only causes sourcing the script where it
                        happens to be aborted, further plugins will be loaded.
                        See |packages|.

:scripte[ncoding] [encoding]            *:scripte* *:scriptencoding* *E167*
                        Specify the character encoding used in the script.
                        The following lines will be converted from [encoding]
                        to the value of the 'encoding' option, if they are
                        different.  Examples: >
                                scriptencoding iso-8859-5
                                scriptencoding cp932
<
                        When [encoding] is empty, no conversion is done.  This
                        can be used to restrict conversion to a sequence of
```

```
                        lines: >
                                scriptencoding euc-jp
                                ... lines to be converted ...
                                scriptencoding
                                ... not converted ...

<                       When conversion isn't supported by the system, there
                        is no error message and no conversion is done.  When a
                        line can't be converted there is no error and the
                        original line is kept.

                        Don't use "ucs-2" or "ucs-4", scripts cannot be in
                        these encodings (they would contain NUL bytes).
                        When a sourced script starts with a BOM (Byte Order
                        Mark) in utf-8 format Vim will recognize it, no need
                        to use ":scriptencoding utf-8" then.

                        If you set the 'encoding' option in your |.vimrc|,
                        `:scriptencoding` must be placed after that. E.g.: >
                                set encoding=utf-8
                                scriptencoding utf-8
<
                        When compiled without the |+multi_byte| feature this
                        command is ignored.
                        {not in Vi}

                                                *:scr* *:scriptnames*
:scr[iptnames]          List all sourced script names, in the order they were
                        first sourced.  The number is used for the script ID
                        |<SID>|.
                        {not in Vi} {not available when compiled without the
                        |+eval| feature}

                                                *:fini* *:finish* *E168*
:fini[sh]               Stop sourcing a script.  Can only be used in a Vim
                        script file.  This is a quick way to skip the rest of
                        the file.  If it 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 in the script.  The outermost ":endtry"
                        then stops sourcing the script.  {not in Vi}

All commands and command sequences can be repeated by putting them in a named
register and then executing it.  There are two ways to get the commands in the
register:
- Use the record command "q".  You type the commands once, and while they are
  being executed they are stored in a register.  Easy, because you can see
  what you are doing.  If you make a mistake, "p"ut the register into the
  file, edit the command sequence, and then delete it into the register
  again.  You can continue recording by appending to the register (use an
  uppercase letter).
- Delete or yank the command sequence into the register.

Often used command sequences can be put under a function key with the ':map'
command.

An alternative is to put the commands in a file, and execute them with the
':source!' command.  Useful for long command sequences.  Can be combined with
the ':map' command to put complicated commands under a function key.

The ':source' command reads Ex commands from a file line by line.  You will
```

have to type any needed keyboard input. The ':source!' command reads from a script file character by character, interpreting each character as if you typed it.

Example: When you give the ":!ls" command you get the |hit-enter| prompt. If you ':source' a file with the line "!ls" in it, you will have to type the <Enter> yourself. But if you ':source!' a file with the line ":!ls" in it, the next characters from that file are read until a <CR> is found. You will not have to type <CR> yourself, unless ":!ls" was the last line in the file.

It is possible to put ':source[!]' commands in the script file, so you can make a top-down hierarchy of script files. The ':source' command can be nested as deep as the number of files that can be opened at one time (about 15). The ':source!' command can be nested up to 15 levels deep.

You can use the "<sfile>" string (literally, this is not a special key) inside of the sourced file, in places where a file name is expected. It will be replaced by the file name of the sourced file. For example, if you have a "other.vimrc" file in the same directory as your ".vimrc" file, you can source it from your ".vimrc" file with this command: >

```
:source <sfile>:h/other.vimrc
```

In script files terminal-dependent key codes are represented by terminal-independent two character codes. This means that they can be used in the same way on different kinds of terminals. The first character of a key code is 0x80 or 128, shown on the screen as "~@". The second one can be found in the list |key-notation|. Any of these codes can also be entered with CTRL-V followed by the three digit decimal code. This does NOT work for the <t_xx> termcap codes, these can only be used in mappings.

:source_crnl *W15*

MS-DOS, Win32 and OS/2: Files that are read with ":source" normally have <CR><NL> <EOL>s. These always work. If you are using a file with <NL> <EOL>s (for example, a file made on Unix), this will be recognized if 'fileformats' is not empty and the first line does not end in a <CR>. This fails if the first line has something like ":map <F1> :help^M", where "^M" is a <CR>. If the first line ends in a <CR>, but following ones don't, you will get an error message, because the <CR> from the first lines will be lost.

Mac Classic: Files that are read with ":source" normally have <CR> <EOL>s. These always work. If you are using a file with <NL> <EOL>s (for example, a file made on Unix), this will be recognized if 'fileformats' is not empty and the first line does not end in a <CR>. Be careful not to use a file with <NL> linebreaks which has a <CR> in first line.

On other systems, Vim expects ":source"ed files to end in a <NL>. These always work. If you are using a file with <CR><NL> <EOL>s (for example, a file made on MS-DOS), all lines will have a trailing <CR>. This may cause problems for some commands (e.g., mappings). There is no automatic <EOL> detection, because it's common to start with a line that defines a mapping that ends in a <CR>, which will confuse the automaton.

line-continuation

Long lines in a ":source"d Ex command script file can be split by inserting a line continuation symbol "\" (backslash) at the start of the next line. There can be white space before the backslash, which is ignored.

Example: the lines >

```
:set comments=sr:/*,mb:*,el:*/,
                \://,
                \b:#,
                \:%,
```

```
                    \n:>,
                    \fb:-
are interpreted as if they were given in one line:
        :set comments=sr:/*,mb:*,el:*/,://,b:#,:%,n:>,fb:-

All leading whitespace characters in the line before a backslash are ignored.
Note however that trailing whitespace in the line before it cannot be
inserted freely; it depends on the position where a command is split up
whether additional whitespace is allowed or not.

When a space is required it's best to put it right after the backslash.  A
space at the end of a line is hard to see and may be accidentally deleted. >
        :syn match Comment
                \ "very long regexp"
                \ keepend

There is a problem with the ":append" and ":insert" commands: >
   :1append
   \asdf
   .
The backslash is seen as a line-continuation symbol, thus this results in the
command: >
   :1appendasdf
   .
To avoid this, add the 'C' flag to the 'cpoptions' option: >
   :set cpo+=C
   :1append
   \asdf
   .
   :set cpo-=C

Note that when the commands are inside a function, you need to add the 'C'
flag when defining the function, it is not relevant when executing it. >
   :set cpo+=C
   :function Foo()
   :1append
   \asdf
   .
   :endfunction
   :set cpo-=C

Rationale:
        Most programs work with a trailing backslash to indicate line
        continuation.  Using this in Vim would cause incompatibility with Vi.
        For example for this Vi mapping: >
                :map xx  asdf\
<       Therefore the unusual leading backslash is used.

==============================================================================
5. Using Vim packages                                   *packages*

A Vim package is a directory that contains one or more plugins.  The
advantages over normal plugins:
- A package can be downloaded as an archive and unpacked in its own directory.
  Thus the files are not mixed with files of other plugins.  That makes it
  easy to update and remove.
- A package can be a git, mercurial, etc. repository.  That makes it really
  easy to update.
- A package can contain multiple plugins that depend on each other.
- A package can contain plugins that are automatically loaded on startup and
  ones that are only loaded when needed with `:packadd`.
```

```
Using a package and loading automatically ~

Let's assume your Vim files are in the "~/.vim" directory and you want to add a
package from a zip archive "/tmp/foopack.zip":
        % mkdir -p ~/.vim/pack/foo
        % cd ~/.vim/pack/foo
        % unzip /tmp/foopack.zip

The directory name "foo" is arbitrary, you can pick anything you like.

You would now have these files under ~/.vim:
        pack/foo/README.txt
        pack/foo/start/foobar/plugin/foo.vim
        pack/foo/start/foobar/syntax/some.vim
        pack/foo/opt/foodebug/plugin/debugger.vim

When Vim starts up, after processing your .vimrc, it scans all directories in
'packpath' for plugins under the "pack/*/start" directory.  First all those
directories are added to 'runtimepath'.  Then all the plugins are loaded.
See |packload-two-steps| for how these two steps can be useful.

In the example Vim will find "pack/foo/start/foobar/plugin/foo.vim" and adds
"~/.vim/pack/foo/start/foobar" to 'runtimepath'.

If the "foobar" plugin kicks in and sets the 'filetype' to "some", Vim will
find the syntax/some.vim file, because its directory is in 'runtimepath'.

Vim will also load ftdetect files, if there are any.

Note that the files under "pack/foo/opt" are not loaded automatically, only the
ones under "pack/foo/start".  See |pack-add| below for how the "opt" directory
is used.

Loading packages automatically will not happen if loading plugins is disabled,
see |load-plugins|.

To load packages earlier, so that 'runtimepath' gets updated: >
        :packloadall
This also works when loading plugins is disabled.  The automatic loading will
only happen once.

If the package has an "after" directory, that directory is added to the end of
'runtimepath', so that anything there will be loaded later.


Using a single plugin and loading it automatically ~

If you don't have a package but a single plugin, you need to create the extra
directory level:
        % mkdir -p ~/.vim/pack/foo/start/foobar
        % cd ~/.vim/pack/foo/start/foobar
        % unzip /tmp/someplugin.zip

You would now have these files:
        pack/foo/start/foobar/plugin/foo.vim
        pack/foo/start/foobar/syntax/some.vim

From here it works like above.


Optional plugins ~
```

pack-add
To load an optional plugin from a pack use the `:packadd` command: >
        :packadd foodebug
This searches for "pack/*/opt/foodebug" in 'packpath' and will find
~/.vim/pack/foo/opt/foodebug/plugin/debugger.vim and source it.

This could be done if some conditions are met. For example, depending on
whether Vim supports a feature or a dependency is missing.

You can also load an optional plugin at startup, by putting this command in
your |.vimrc|: >
        :packadd! foodebug
The extra "!" is so that the plugin isn't loaded if Vim was started with
|--noplugin|.

It is perfectly normal for a package to only have files in the "opt"
directory. You then need to load each plugin when you want to use it.

Where to put what ~

Since color schemes, loaded with `:colorscheme`, are found below
"pack/*/start" and "pack/*/opt", you could put them anywhere. We recommend
you put them below "pack/*/opt", for example
".vim/pack/mycolors/opt/dark/colors/very_dark.vim".

Filetype plugins should go under "pack/*/start", so that they are always
found. Unless you have more than one plugin for a file type and want to
select which one to load with `:packadd`. E.g. depending on the compiler
version: >
        if foo_compiler_version > 34
          packadd foo_new
        else
          packadd foo_old
        endif

The "after" directory is most likely not useful in a package. It's not
disallowed though.

==
6. Creating Vim packages *package-create*

This assumes you write one or more plugins that you distribute as a package.

If you have two unrelated plugins you would use two packages, so that Vim
users can chose what they include or not. Or you can decide to use one
package with optional plugins, and tell the user to add the ones he wants with
`:packadd`.

Decide how you want to distribute the package. You can create an archive or
you could use a repository. An archive can be used by more users, but is a
bit harder to update to a new version. A repository can usually be kept
up-to-date easily, but it requires a program like "git" to be available.
You can do both, github can automatically create an archive for a release.

Your directory layout would be like this:
   start/foobar/plugin/foo.vim   " always loaded, defines commands
   start/foobar/plugin/bar.vim   " always loaded, defines commands
   start/foobar/autoload/foo.vim   " loaded when foo command used
   start/foobar/doc/foo.txt   " help for foo.vim
   start/foobar/doc/tags   " help tags
   opt/fooextra/plugin/extra.vim   " optional plugin, defines commands

```
   opt/fooextra/autoload/extra.vim      " loaded when extra command used
   opt/fooextra/doc/extra.txt           " help for extra.vim
   opt/fooextra/doc/tags                " help tags

This allows for the user to do: >
        mkdir ~/.vim/pack/myfoobar
        cd ~/.vim/pack/myfoobar
        git clone https://github.com/you/foobar.git

Here "myfoobar" is a name that the user can choose, the only condition is that
it differs from other packages.

In your documentation you explain what the plugins do, and tell the user how
to load the optional plugin: >
        :packadd! fooextra

You could add this packadd command in one of your plugins, to be executed when
the optional plugin is needed.

Run the `:helptags` command to generate the doc/tags file.  Including this
generated file in the package means that the user can drop the package in his
pack directory and the help command works right away.  Don't forget to re-run
the command after changing the plugin help: >
        :helptags path/start/foobar/doc
        :helptags path/opt/fooextra/doc


Dependencies between plugins ~
                                                        *packload-two-steps*
Suppose you have two plugins that depend on the same functionality. You can
put the common functionality in an autoload directory, so that it will be
found automatically.  Your package would have these files:

        pack/foo/start/one/plugin/one.vim  >
                call foolib#getit()
<       pack/foo/start/two/plugin/two.vim >
                call foolib#getit()
<       pack/foo/start/lib/autoload/foolib.vim >
                func foolib#getit()

This works, because loading packages will first add all found directories to
'runtimepath' before sourcing the plugins.

==============================================================================
7. Debugging scripts                                    *debug-scripts*

Besides the obvious messages that you can add to your scripts to find out what
they are doing, Vim offers a debug mode.  This allows you to step through a
sourced file or user function and set breakpoints.

NOTE: The debugging mode is far from perfect.  Debugging will have side
effects on how Vim works.  You cannot use it to debug everything.  For
example, the display is messed up by the debugging messages.
{Vi does not have a debug mode}

An alternative to debug mode is setting the 'verbose' option.  With a bigger
number it will give more verbose messages about what Vim is doing.


STARTING DEBUG MODE                                             *debug-mode*

To enter debugging mode use one of these methods:
```

```
1. Start Vim with the |-D| argument: >
        vim -D file.txt
<  Debugging will start as soon as the first vimrc file is sourced.  This is
   useful to find out what is happening when Vim is starting up.  A side
   effect is that Vim will switch the terminal mode before initialisations
   have finished, with unpredictable results.
   For a GUI-only version (Windows, Macintosh) the debugging will start as
   soon as the GUI window has been opened.  To make this happen early, add a
   ":gui" command in the vimrc file.
                                                                    *:debug*
2. Run a command with ":debug" prepended.  Debugging will only be done while
   this command executes.  Useful for debugging a specific script or user
   function.  And for scripts and functions used by autocommands.  Example: >
        :debug edit test.txt.gz

3. Set a breakpoint in a sourced file or user function.  You could do this in
   the command line: >
        vim -c "breakadd file */explorer.vim" .
<  This will run Vim and stop in the first line of the "explorer.vim" script.
   Breakpoints can also be set while in debugging mode.

In debugging mode every executed command is displayed before it is executed.
Comment lines, empty lines and lines that are not executed are skipped.  When
a line contains two commands, separated by "|", each command will be displayed
separately.


DEBUG MODE

Once in debugging mode, the usual Ex commands can be used.  For example, to
inspect the value of a variable: >
        echo idx
When inside a user function, this will print the value of the local variable
"idx".  Prepend "g:" to get the value of a global variable: >
        echo g:idx
All commands are executed in the context of the current function or script.
You can also set options, for example setting or resetting 'verbose' will show
what happens, but you might want to set it just before executing the lines you
are interested in: >
        :set verbose=20

Commands that require updating the screen should be avoided, because their
effect won't be noticed until after leaving debug mode.  For example: >
        :help
won't be very helpful.

There is a separate command-line history for debug mode.

The line number for a function line is relative to the start of the function.
If you have trouble figuring out where you are, edit the file that defines
the function in another Vim, search for the start of the function and do
"99j".  Replace "99" with the line number.

Additionally, these commands can be used:
                                                        *>cont*
        cont            Continue execution until the next breakpoint is hit.
                                                        *>quit*
        quit            Abort execution.  This is like using CTRL-C, some
                        things might still be executed, doesn't abort
                        everything.  Still stops at the next breakpoint.
                                                        *>next*
        next            Execute the command and come back to debug mode when
```

```
                        it's finished.  This steps over user function calls
                        and sourced files.
                                                        *>step*
        step            Execute the command and come back to debug mode for
                        the next command.  This steps into called user
                        functions and sourced files.
                                                        *>interrupt*
        interrupt       This is like using CTRL-C, but unlike ">quit" comes
                        back to debug mode for the next command that is
                        executed.  Useful for testing |:finally| and |:catch|
                        on interrupt exceptions.
                                                        *>finish*
        finish          Finish the current script or user function and come
                        back to debug mode for the command after the one that
                        sourced or called it.
                                                        *>bt*
                                                        *>backtrace*
                                                        *>where*
        backtrace       Show the call stacktrace for current debugging session.
        bt
        where
                                                        *>frame*
        frame N         Goes to N backtrace level. + and - signs make movement
                        relative.  E.g., ":frame +3" goes three frames up.
                                                        *>up*
        up              Goes one level up from call stacktrace.
                                                        *>down*
        down            Goes one level down from call stacktrace.

About the additional commands in debug mode:
- There is no command-line completion for them, you get the completion for the
  normal Ex commands only.
- You can shorten them, up to a single character, unless more than one command
  starts with the same letter.  "f" stands for "finish", use "fr" for "frame".
- Hitting <CR> will repeat the previous one.  When doing another command, this
  is reset (because it's not clear what you want to repeat).
- When you want to use the Ex command with the same name, prepend a colon:
  ":cont", ":next", ":finish" (or shorter).

The backtrace shows the hierarchy of function calls, e.g.:
        >bt ~
          3 function One[3] ~
          2 Two[3] ~
        ->1 Three[3] ~
          0 Four ~
        line 1: let four = 4 ~

The "->" points to the current frame.  Use "up", "down" and "frame N" to
select another frame.

In the current frame you can evaluate the local function variables.  There is
no way to see the command at the current line yet.


DEFINING BREAKPOINTS
                                                        *:breaka* *:breakadd*
:breaka[dd] func [lnum] {name}
                Set a breakpoint in a function.  Example: >
                        :breakadd func Explore
<               Doesn't check for a valid function name, thus the breakpoint
                can be set before the function is defined.
```

```
:breaka[dd] file [lnum] {name}
                Set a breakpoint in a sourced file.  Example: >
                        :breakadd file 43 .vimrc

:breaka[dd] here
                Set a breakpoint in the current line of the current file.
                Like doing: >
                        :breakadd file <cursor-line> <current-file>
<               Note that this only works for commands that are executed when
                sourcing the file, not for a function defined in that file.

The [lnum] is the line number of the breakpoint.  Vim will stop at or after
this line.  When omitted line 1 is used.

                                                        *:debug-name*
{name} is a pattern that is matched with the file or function name.  The
pattern is like what is used for autocommands.  There must be a full match (as
if the pattern starts with "^" and ends in "$").  A "*" matches any sequence
of characters.  'ignorecase' is not used, but "\c" can be used in the pattern
to ignore case |/\c|.  Don't include the () for the function name!

The match for sourced scripts is done against the full file name.  If no path
is specified the current directory is used.  Examples: >
        breakadd file explorer.vim
matches "explorer.vim" in the current directory. >
        breakadd file *explorer.vim
matches ".../plugin/explorer.vim", ".../plugin/iexplorer.vim", etc. >
        breakadd file */explorer.vim
matches ".../plugin/explorer.vim" and "explorer.vim" in any other directory.

The match for functions is done against the name as it's shown in the output
of ":function".  For local functions this means that something like "<SNR>99_"
is prepended.

Note that functions are first loaded and later executed.  When they are loaded
the "file" breakpoints are checked, when they are executed the "func"
breakpoints.


DELETING BREAKPOINTS
                                            *:breakd* *:breakdel* *E161*
:breakd[el] {nr}
                Delete breakpoint {nr}.  Use |:breaklist| to see the number of
                each breakpoint.

:breakd[el] *
                Delete all breakpoints.

:breakd[el] func [lnum] {name}
                Delete a breakpoint in a function.

:breakd[el] file [lnum] {name}
                Delete a breakpoint in a sourced file.

:breakd[el] here
                Delete a breakpoint at the current line of the current file.

When [lnum] is omitted, the first breakpoint in the function or file is
deleted.
The {name} must be exactly the same as what was typed for the ":breakadd"
command.  "explorer", "*explorer.vim" and "*explorer*" are different.
```

```
LISTING BREAKPOINTS
                                                        *:breakl* *:breaklist*
:breakl[ist]
                List all breakpoints.


OBSCURE

                                                *:debugg* *:debuggreedy*
:debugg[reedy]
                Read debug mode commands from the normal input stream, instead
                of getting them directly from the user.  Only useful for test
                scripts.  Example: >
                  echo 'q^Mq' | vim -e -s -c debuggreedy -c 'breakadd file
script.vim' -S script.vim

:0debugg[reedy]
                Undo ":debuggreedy": get debug mode commands directly from the
                user, don't use typeahead for debug commands.

==============================================================================
8. Profiling                                            *profile* *profiling*

Profiling means that Vim measures the time that is spent on executing
functions and/or scripts.  The |+profile| feature is required for this.
It is only included when Vim was compiled with "huge" features.
{Vi does not have profiling}

You can also use the |reltime()| function to measure time.  This only requires
the |+reltime| feature, which is present more often.

For profiling syntax highlighting see |:syntime|.

For example, to profile the one_script.vim script file: >
        :profile start /tmp/one_script_profile
        :profile file one_script.vim
        :source one_script.vim
        :exit


:prof[ile] start {fname}                        *:prof* *:profile* *E750*
                Start profiling, write the output in {fname} upon exit.
                "~/" and environment variables in {fname} will be expanded.
                If {fname} already exists it will be silently overwritten.
                The variable |v:profiling| is set to one.

:prof[ile] pause
                Don't profile until the following ":profile continue".  Can be
                used when doing something that should not be counted (e.g., an
                external command).  Does not nest.

:prof[ile] continue
                Continue profiling after ":profile pause".

:prof[ile] func {pattern}
                Profile function that matches the pattern {pattern}.
                See |:debug-name| for how {pattern} is used.

:prof[ile][!] file {pattern}
                Profile script file that matches the pattern {pattern}.
                See |:debug-name| for how {pattern} is used.
```

This only profiles the script itself, not the functions
defined in it.
When the [!] is added then all functions defined in the script
will also be profiled.
Note that profiling only starts when the script is loaded
after this command. A :profile command in the script itself
won't work.

:profd[el] ... *:profd* *:profdel*
Stop profiling for the arguments specified. See |:breakdel|
for the arguments.

You must always start with a ":profile start fname" command. The resulting
file is written when Vim exits. Here is an example of the output, with line
numbers prepended for the explanation:

```
  1 FUNCTION  Test2() ~
  2 Called 1 time ~
  3 Total time:   0.155251 ~
  4  Self time:   0.002006 ~
  5  ~
  6 count  total (s)   self (s) ~
  7     9              0.000096   for i in range(8) ~
  8     8   0.153655   0.000410     call Test3() ~
  9     8              0.000070   endfor ~
 10                              " Ask a question ~
 11     1              0.001341   echo input("give me an answer: ") ~
```

The header (lines 1-4) gives the time for the whole function. The "Total"
time is the time passed while the function was executing. The "Self" time is
the "Total" time reduced by time spent in:
- other user defined functions
- sourced scripts
- executed autocommands
- external (shell) commands

Lines 7-11 show the time spent in each executed line. Lines that are not
executed do not count. Thus a comment line is never counted.

The Count column shows how many times a line was executed. Note that the
"for" command in line 7 is executed one more time as the following lines.
That is because the line is also executed to detect the end of the loop.

The time Vim spends waiting for user input isn't counted at all. Thus how
long you take to respond to the input() prompt is irrelevant.

Profiling should give a good indication of where time is spent, but keep in
mind there are various things that may clobber the results:

- The accuracy of the time measured depends on the gettimeofday() system
  function. It may only be as accurate as 1/100 second, even though the times
  are displayed in micro seconds.

- Real elapsed time is measured, if other processes are busy they may cause
  delays at unpredictable moments. You may want to run the profiling several
  times and use the lowest results.

- If you have several commands in one line you only get one time. Split the
  line to see the time for the individual commands.

```
- The time of the lines added up is mostly less than the time of the whole
  function.  There is some overhead in between.

- Functions that are deleted before Vim exits will not produce profiling
  information.  You can check the |v:profiling| variable if needed: >
        :if !v:profiling
        :   delfunc MyFunc
        :endif
<
- Profiling may give weird results on multi-processor systems, when sleep
  mode kicks in or the processor frequency is reduced to save power.

- The "self" time is wrong when a function is used recursively.


 vim:tw=78:ts=8:ft=help:norl:
*visual.txt*    For Vim version 8.0.  Last change: 2017 Sep 02


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Visual mode                              *Visual* *Visual-mode* *visual-mode*

Visual mode is a flexible and easy way to select a piece of text for an
operator.  It is the only way to select a block of text.

This is introduced in section |04.4| of the user manual.

1. Using Visual mode                    |visual-use|
2. Starting and stopping Visual mode    |visual-start|
3. Changing the Visual area             |visual-change|
4. Operating on the Visual area         |visual-operators|
5. Blockwise operators                  |blockwise-operators|
6. Repeating                            |visual-repeat|
7. Examples                             |visual-examples|
8. Select mode                          |Select-mode|

{Vi has no Visual mode, the name "visual" is used for Normal mode, to
distinguish it from Ex mode}
{Since Vim 7.4.200 the |+visual| feature is always included}

==============================================================================
1. Using Visual mode                                        *visual-use*

Using Visual mode consists of three parts:
1. Mark the start of the text with "v", "V" or CTRL-V.
   The character under the cursor will be used as the start.
2. Move to the end of the text.
   The text from the start of the Visual mode up to and including the
   character under the cursor is highlighted.
3. Type an operator command.
   The highlighted characters will be operated upon.

The 'highlight' option can be used to set the display mode to use for
highlighting in Visual mode.
The 'virtualedit' option can be used to allow positioning the cursor to
positions where there is no actual character.

The highlighted text normally includes the character under the cursor.
However, when the 'selection' option is set to "exclusive" and the cursor is
after the Visual area, the character under the cursor is not included.
```

With "v" the text before the start position and after the end position will
not be highlighted. However, all uppercase and non-alpha operators, except
"~" and "U", will work on whole lines anyway. See the list of operators
below.

visual-block
With CTRL-V (blockwise Visual mode) the highlighted text will be a rectangle
between start position and the cursor. However, some operators work on whole
lines anyway (see the list below). The change and substitute operators will
delete the highlighted text and then start insertion at the top left
position.

==
2. Starting and stopping Visual mode *visual-start*

v *characterwise-visual*
[count]v Start Visual mode per character.
With [count] select the same number of characters or
lines as used for the last Visual operation, but at
the current cursor position, multiplied by [count].
When the previous Visual operation was on a block both
the width and height of the block are multiplied by
[count].
When there was no previous Visual operation [count]
characters are selected. This is like moving the
cursor right N * [count] characters. One less when
'selection' is not "exclusive".

V *linewise-visual*
[count]V Start Visual mode linewise.
With [count] select the same number of lines as used
for the last Visual operation, but at the current
cursor position, multiplied by [count]. When there
was no previous Visual operation [count] lines are
selected.

CTRL-V *blockwise-visual*
[count]CTRL-V Start Visual mode blockwise. Note: Under Windows
CTRL-V could be mapped to paste text, it doesn't work
to start Visual mode then, see |CTRL-V-alternative|.
[count] is used as with `v` above.

If you use <Esc>, click the left mouse button or use any command that
does a jump to another buffer while in Visual mode, the highlighting stops
and no text is affected. Also when you hit "v" in characterwise Visual mode,
"CTRL-V" in blockwise Visual mode or "V" in linewise Visual mode. If you hit
CTRL-Z the highlighting stops and the editor is suspended or a new shell is
started |CTRL-Z|.

new mode after typing: *v_v* *v_CTRL-V* *v_V*
old mode "v" "CTRL-V" "V" ~

Normal	Visual	blockwise Visual	linewise Visual
Visual	Normal	blockwise Visual	linewise Visual
blockwise Visual	Visual	Normal	linewise Visual
linewise Visual	Visual	blockwise Visual	Normal

gv *v_gv* *reselect-Visual*
gv Start Visual mode with the same area as the previous
area and the same mode.
In Visual mode the current and the previous Visual

area are exchanged.
After using "p" or "P" in Visual mode the text that
was put will be selected.

gn *v_gn*
gn Search forward for the last used search pattern, like
with `n`, and start Visual mode to select the match.
If the cursor is on the match, visually selects it.
If an operator is pending, operates on the match.
E.g., "dgn" deletes the text of the next match.
If Visual mode is active, extends the selection
until the end of the next match.

gN *v_gN*
gN Like |gn| but searches backward, like with `N`.

<LeftMouse>
<LeftMouse> Set the current cursor position. If Visual mode is
active it is stopped. Only when 'mouse' option is
contains 'n' or 'a'. If the position is within 'so'
lines from the last line on the screen the text is
scrolled up. If the position is within 'so' lines from
the first line on the screen the text is scrolled
down.

<RightMouse>
<RightMouse> Start Visual mode if it is not active. The text from
the cursor position to the position of the click is
highlighted. If Visual mode was already active move
the start or end of the highlighted text, which ever
is closest, to the position of the click. Only when
'mouse' option contains 'n' or 'a'.

Note: when 'mousemodel' is set to "popup",
<S-LeftMouse> has to be used instead of <RightMouse>.

<LeftRelease>
<LeftRelease> This works like a <LeftMouse>, if it is not at
the same position as <LeftMouse>. In an older version
of xterm you won't see the selected area until the
button is released, unless there is access to the
display where the xterm is running (via the DISPLAY
environment variable or the -display argument). Only
when 'mouse' option contains 'n' or 'a'.

If Visual mode is not active and the "v", "V" or CTRL-V is preceded with a
count, the size of the previously highlighted area is used for a start. You
can then move the end of the highlighted area and give an operator. The type
of the old area is used (character, line or blockwise).
- Linewise Visual mode: The number of lines is multiplied with the count.
- Blockwise Visual mode: The number of lines and columns is multiplied with
  the count.
- Normal Visual mode within one line: The number of characters is multiplied
  with the count.
- Normal Visual mode with several lines: The number of lines is multiplied
  with the count, in the last line the same number of characters is used as
  in the last line in the previously highlighted area.
The start of the text is the Cursor position. If the "$" command was used as
one of the last commands to extend the highlighted text, the area will be
extended to the rightmost column of the longest line.

If you want to highlight exactly the same area as the last time, you can use

"gv" |gv| |v_gv|.

v_<Esc>
<Esc> In Visual mode: Stop Visual mode.

v_CTRL-C
CTRL-C In Visual mode: Stop Visual mode. When insert mode is pending (the mode message shows "-- (insert) VISUAL --"), it is also stopped.

==
3. Changing the Visual area *visual-change*

v_o
o Go to Other end of highlighted text: The current cursor position becomes the start of the highlighted text and the cursor is moved to the other end of the highlighted text. The highlighted area remains the same.

v_O
O Go to Other end of highlighted text. This is like "o", but in Visual block mode the cursor moves to the other corner in the same line. When the corner is at a character that occupies more than one position on the screen (e.g., a <Tab>), the highlighted text may change.

v_$
When the "$" command is used with blockwise Visual mode, the right end of the highlighted text will be determined by the longest highlighted line. This stops when a motion command is used that does not move straight up or down.

For moving the end of the block many commands can be used, but you cannot use Ex commands, commands that make changes or abandon the file. Commands (starting with) ".", "&", CTRL-^, "Z", CTRL-], CTRL-T, CTRL-R, CTRL-I and CTRL-O cause a beep and Visual mode continues.

When switching to another window on the same buffer, the cursor position in that window is adjusted, so that the same Visual area is still selected. This is especially useful to view the start of the Visual area in one window, and the end in another. You can then use <RightMouse> (or <S-LeftMouse> when 'mousemodel' is "popup") to drag either end of the Visual area.

==
4. Operating on the Visual area *visual-operators*

The operators that can be used are:

~	switch case	|v_~|
d	delete	|v_d|
c	change (4)	|v_c|
y	yank	|v_y|
>	shift right (4)	|v_>|
<	shift left (4)	|v_<|
!	filter through external command (1)	|v_!|
=	filter through 'equalprg' option command (1)	|v_=|
gq	format lines to 'textwidth' length (1)	|v_gq|

The objects that can be used are:

aw	a word (with white space)	|v_aw|
iw	inner word	|v_iw|
aW	a WORD (with white space)	|v_aW|

```
        iW      inner WORD                                          |v_iW|
        as      a sentence (with white space)                       |v_as|
        is      inner sentence                                      |v_is|
        ap      a paragraph (with white space)                      |v_ap|
        ip      inner paragraph                                     |v_ip|
        ab      a () block (with parenthesis)                       |v_ab|
        ib      inner () block                                      |v_ib|
        aB      a {} block (with braces)                            |v_aB|
        iB      inner {} block                                      |v_iB|
        at      a <tag> </tag> block (with tags)                    |v_at|
        it      inner <tag> </tag> block                            |v_it|
        a<      a <> block (with <>)                                |v_a<|
        i<      inner <> block                                      |v_i<|
        a[      a [] block (with [])                                |v_a[|
        i[      inner [] block                                      |v_i[|
        a"      a double quoted string (with quotes)                |v_aquote|
        i"      inner double quoted string                          |v_iquote|
        a'      a single quoted string (with quotes)                |v_a'|
        i'      inner simple quoted string                          |v_i'|
        a`      a string in backticks (with backticks)              |v_a`|
        i`      inner string in backticks                           |v_i`|

Additionally the following commands can be used:
        :       start Ex command for highlighted lines (1)          |v_:|
        r       change (4)                                          |v_r|
        s       change                                              |v_s|
        C       change (2)(4)                                       |v_C|
        S       change (2)                                          |v_S|
        R       change (2)                                          |v_R|
        x       delete                                              |v_x|
        D       delete (3)                                          |v_D|
        X       delete (2)                                          |v_X|
        Y       yank (2)                                            |v_Y|
        p       put                                                 |v_p|
        J       join (1)                                            |v_J|
        U       make uppercase                                      |v_U|
        u       make lowercase                                      |v_u|
        ^]      find tag                                            |v_CTRL-]|
        I       block insert                                        |v_b_I|
        A       block append                                        |v_b_A|

(1): Always whole lines, see |:visual_example|.
(2): Whole lines when not using CTRL-V.
(3): Whole lines when not using CTRL-V, delete until the end of the line when
     using CTRL-V.
(4): When using CTRL-V operates on the block only.

Note that the ":vmap" command can be used to specifically map keys in Visual
mode.  For example, if you would like the "/" command not to extend the Visual
area, but instead take the highlighted text and search for that: >
        :vmap / y/<C-R>"<CR>
(In the <> notation |<>|, when typing it you should type it literally; you
need to remove the 'B' and '<' flags from 'cpoptions'.)

If you want to give a register name using the """ command, do this just before
typing the operator character: "v{move-around}"xd".

If you want to give a count to the command, do this just before typing the
operator character: "v{move-around}3>" (move lines 3 indents to the right).

                                                        *{move-around}*
The {move-around} is any sequence of movement commands.  Note the difference
```

with {motion}, which is only ONE movement command.

Another way to operate on the Visual area is using the |/\%V| item in a
pattern. For example, to replace all '(' in the Visual area with '#': >

```
:'<,'>s/\%V(/#/g
```

Note that the "'<,'>" will appear automatically when you press ":" in Visual
mode.

==
5. Blockwise operators *blockwise-operators*

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

Reminder: Use 'virtualedit' to be able to select blocks that start or end
after the end of a line or halfway a tab.

Visual-block Insert *v_b_I*
With a blockwise selection, I{string}<ESC> will insert {string} at the start
of block on every line of the block, provided that the line extends into the
block. Thus lines that are short will remain unmodified. TABs are split to
retain visual columns. Works only for adding text to a line, not for
deletions. See |v_b_I_example|.

Visual-block Append *v_b_A*
With a blockwise selection, A{string}<ESC> will append {string} to the end of
block on every line of the block. There is some differing behavior where the
block RHS is not straight, due to different line lengths:

1. Block was created with <C-v>$
   In this case the string is appended to the end of each line.
2. Block was created with <C-v>{move-around}
   In this case the string is appended to the end of the block on each line,
   and whitespace is inserted to pad to the end-of-block column.
See |v_b_A_example|.
Note: "I" and "A" behave differently for lines that don't extend into the
selected block. This was done intentionally, so that you can do it the way
you want.
Works only for adding text to a line, not for deletions.

Visual-block change *v_b_c*
All selected text in the block will be replaced by the same text string. When
using "c" the selected text is deleted and Insert mode started. You can then
enter text (without a line break). When you hit <Esc>, the same string is
inserted in all previously selected lines.

Visual-block Change *v_b_C*
Like using "c", but the selection is extended until the end of the line for
all lines.

v_b_<
Visual-block Shift *v_b_>*
The block is shifted by 'shiftwidth'. The RHS of the block is irrelevant. The
LHS of the block determines the point from which to apply a right shift, and
padding includes TABs optimally according to 'ts' and 'et'. The LHS of the
block determines the point upto which to shift left.
See |v_b_>_example|.
See |v_b_<_example|.

Visual-block Replace *v_b_r*
Every screen char in the highlighted region is replaced with the same char, ie

TABs are split and the virtual whitespace is replaced, maintaining screen
layout.
See |v_b_r_example|.

==
6. Repeating *visual-repeat*

When repeating a Visual mode operator, the operator will be applied to the
same amount of text as the last time:
- Linewise Visual mode: The same number of lines.
- Blockwise Visual mode: The same number of lines and columns.
- Normal Visual mode within one line: The same number of characters.
- Normal Visual mode with several lines: The same number of lines, in the
  last line the same number of characters as in the last line the last time.
The start of the text is the Cursor position. If the "$" command was used as
one of the last commands to extend the highlighted text, the repeating will
be applied up to the rightmost column of the longest line.

==
7. Examples *visual-examples*

:visual_example
Currently the ":" command works on whole lines only. When you select part of
a line, doing something like ":!date" will replace the whole line. If you
want only part of the line to be replaced you will have to make a mapping for
it. In a future release ":" may work on partial lines.

Here is an example, to replace the selected text with the output of "date": >
        :vmap _a <Esc>`>a<CR><Esc>`<i<CR><Esc>!!date<CR>kJJ

(In the <> notation |<>|, when typing it you should type it literally; you
need to remove the 'B' and '<' flags from 'cpoptions')

What this does is:
<Esc>   stop Visual mode
`>   go to the end of the Visual area
a<CR><Esc>   break the line after the Visual area
`<   jump to the start of the Visual area
i<CR><Esc>   break the line before the Visual area
!!date<CR>   filter the Visual text through date
kJJ   Join the lines back together

visual-search
Here is an idea for a mapping that makes it possible to do a search for the
selected text: >
        :vmap X y/<C-R>"<CR>

(In the <> notation |<>|, when typing it you should type it literally; you
need to remove the 'B' and '<' flags from 'cpoptions')

Note that special characters (like '.' and '*') will cause problems.

Visual-block Examples *blockwise-examples*
With the following text, I will indicate the commands to produce the block and
the results below. In all cases, the cursor begins on the 'a' in the first
line of the test text.
The following modeline settings are assumed ":ts=8:sw=4:".

It will be helpful to
:set hls

```
/<TAB>
where <TAB> is a real TAB.  This helps visualise the operations.

The test text is:

abcdefghijklmnopqrstuvwxyz
abc             defghijklmnopqrstuvwxyz
abcdef  ghi             jklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz

1. fo<C-v>3jISTRING<ESC>                                        *v_b_I_example*

abcdefghijklmnSTRINGopqrstuvwxyz
abc           STRING  defghijklmnopqrstuvwxyz
abcdef  ghi   STRING    jklmnopqrstuvwxyz
abcdefghijklmnSTRINGopqrstuvwxyz

2. fo<C-v>3j$ASTRING<ESC>                                       *v_b_A_example*

abcdefghijklmnopqrstuvwxyzSTRING
abc             defghijklmnopqrstuvwxyzSTRING
abcdef  ghi             jklmnopqrstuvwxyzSTRING
abcdefghijklmnopqrstuvwxyzSTRING

3. fo<C-v>3j3l<..                                               *v_b_<_example*

abcdefghijklmnopqrstuvwxyz
abc           defghijklmnopqrstuvwxyz
abcdef  ghi   jklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz

4. fo<C-v>3j>..                                                 *v_b_>_example*

abcdefghijklmn            opqrstuvwxyz
abc                         defghijklmnopqrstuvwxyz
abcdef  ghi                         jklmnopqrstuvwxyz
abcdefghijklmn            opqrstuvwxyz

5. fo<C-v>5l3jrX                                                *v_b_r_example*

abcdefghijklmnXXXXXXuvwxyz
abc           XXXXXXhijklmnopqrstuvwxyz
abcdef  ghi   XXXXXX    jklmnopqrstuvwxyz
abcdefghijklmnXXXXXXuvwxyz

==============================================================================
8. Select mode                                      *Select* *Select-mode*

Select mode looks like Visual mode, but the commands accepted are quite
different.  This resembles the selection mode in Microsoft Windows.
When the 'showmode' option is set, "-- SELECT --" is shown in the last line.

Entering Select mode:
- Using the mouse to select an area, and 'selectmode' contains "mouse".
  'mouse' must also contain a flag for the current mode.
- Using a non-printable movement command, with the Shift key pressed, and
  'selectmode' contains "key".  For example: <S-Left> and <S-End>.  'keymodel'
  must also contain "startsel".
- Using "v", "V" or CTRL-V command, and 'selectmode' contains "cmd".
- Using "gh", "gH" or "g_CTRL-H" command in Normal mode.
- From Visual mode, press CTRL-G.                       *v_CTRL-G*
```

```
Commands in Select mode:
- Printable characters, <NL> and <CR> cause the selection to be deleted, and
  Vim enters Insert mode.  The typed character is inserted.
- Non-printable movement commands, with the Shift key pressed, extend the
  selection.  'keymodel' must include "startsel".
- Non-printable movement commands, with the Shift key NOT pressed, stop Select
  mode.  'keymodel' must include "stopsel".
- ESC stops Select mode.
- CTRL-O switches to Visual mode for the duration of one command. *v_CTRL-O*
- CTRL-G switches to Visual mode.

Otherwise, typed characters are handled as in Visual mode.

When using an operator in Select mode, and the selection is linewise, the
selected lines are operated upon, but like in characterwise selection.  For
example, when a whole line is deleted, it can later be pasted halfway a line.


Mappings and menus in Select mode.                      *Select-mode-mapping*

When mappings and menus are defined with the |:vmap| or |:vmenu| command they
work both in Visual mode and in Select mode.  When these are used in Select
mode Vim automatically switches to Visual mode, so that the same behavior as
in Visual mode is effective.  If you don't want this use |:xmap| or |:smap|.

Users will expect printable characters to replace the selected area.
Therefore avoid mapping printable characters in Select mode.  Or use
|:sunmap|  after |:map| and |:vmap| to remove it for Select mode.

After the mapping or menu finishes, the selection is enabled again and Select
mode entered, unless the selected area was deleted, another buffer became
the current one or the window layout was changed.

When a character was typed that causes the selection to be deleted and Insert
mode started, Insert mode mappings are applied to this character.  This may
cause some confusion, because it means Insert mode mappings apply to a
character typed in Select mode.  Language mappings apply as well.

                                                        *gV* *v_gV*
gV                      Avoid the automatic reselection of the Visual area
                        after a Select mode mapping or menu has finished.
                        Put this just before the end of the mapping or menu.
                        At least it should be after any operations on the
                        selection.

                                                        *gh*
gh                      Start Select mode, characterwise.  This is like "v",
                        but starts Select mode instead of Visual mode.
                        Mnemonic: "get highlighted".

                                                        *gH*
gH                      Start Select mode, linewise.  This is like "V",
                        but starts Select mode instead of Visual mode.
                        Mnemonic: "get Highlighted".

                                                        *g_CTRL-H*
g CTRL-H                Start Select mode, blockwise.  This is like CTRL-V,
                        but starts Select mode instead of Visual mode.
                        Mnemonic: "get Highlighted".

 vim:tw=78:ts=8:ft=help:norl:
*various.txt*   For Vim version 8.0.  Last change: 2017 Sep 16
```

VIM REFERENCE MANUAL by Bram Moolenaar

Various commands *various*

==========

1. Various commands *various-cmds*

CTRL-L

CTRL-L Clear and redraw the screen. The redraw may happen later, after processing typeahead.

:redr *:redraw*

:redr[aw][!] Redraw the screen right now. When ! is included it is cleared first.
Useful to update the screen halfway executing a script or function. Also when halfway a mapping and 'lazyredraw' is set.

:redraws *:redrawstatus*

:redraws[tatus][!] Redraw the status line of the current window. When ! is included all status lines are redrawn.
Useful to update the status line(s) when 'statusline' includes an item that doesn't cause automatic updating.

N<Del>

<Del> When entering a number: Remove the last digit.
Note: if you like to use <BS> for this, add this mapping to your .vimrc: >
:map CTRL-V <BS> CTRL-V <Del>
< See |:fixdel| if your <Del> key does not do what you want.

ga *:as* *:ascii*

:as[cii] or
ga Print the ascii value of the character under the cursor in decimal, hexadecimal and octal. For example, when the cursor is on a 'R':
<R> 82, Hex 52, Octal 122 ~
When the character is a non-standard ASCII character, but printable according to the 'isprint' option, the non-printable version is also given. When the character is larger than 127, the <M-x> form is also printed. For example:
<~A> <M-^A> 129, Hex 81, Octal 201 ~
<p> <|~> <M-~> 254, Hex fe, Octal 376 ~
(where <p> is a special character)
The <Nul> character in a file is stored internally as <NL>, but it will be shown as:
<^@> 0, Hex 00, Octal 000 ~
If the character has composing characters these are also shown. The value of 'maxcombine' doesn't matter.
Mnemonic: Get Ascii value. {not in Vi}

g8

g8 Print the hex values of the bytes used in the character under the cursor, assuming it is in |UTF-8|

encoding. This also shows composing characters. The
value of 'maxcombine' doesn't matter.
Example of a character with two composing characters:
e0 b8 81 + e0 b8 b9 + e0 b9 89 ~
{not in Vi} {only when compiled with the |+multi_byte|
feature}

8g8
8g8 Find an illegal UTF-8 byte sequence at or after the
cursor. This works in two situations:
1. when 'encoding' is any 8-bit encoding
2. when 'encoding' is "utf-8" and 'fileencoding' is
any 8-bit encoding
Thus it can be used when editing a file that was
supposed to be UTF-8 but was read as if it is an 8-bit
encoding because it contains illegal bytes.
Does not wrap around the end of the file.
Note that when the cursor is on an illegal byte or the
cursor is halfway a multi-byte character the command
won't move the cursor.
{not in Vi} {only when compiled with the |+multi_byte|
feature}

:p *:pr* *:print* *E749*
:[range]p[rint] [flags]
Print [range] lines (default current line).
Note: If you are looking for a way to print your text
on paper see |:hardcopy|. In the GUI you can use the
File.Print menu entry.
See |ex-flags| for [flags].
The |:filter| command can be used to only show lines
matching a pattern.

:[range]p[rint] {count} [flags]
Print {count} lines, starting with [range] (default
current line |cmdline-ranges|).
See |ex-flags| for [flags].

:P *:Print*
:[range]P[rint] [count] [flags]
Just as ":print". Was apparently added to Vi for
people that keep the shift key pressed too long...
Note: A user command can overrule this command.
See |ex-flags| for [flags].

:l *:list*
:[range]l[ist] [count] [flags]
Same as :print, but display unprintable characters
with '^' and put $ after the line. This can be
further changed with the 'listchars' option.
See |ex-flags| for [flags].

:nu *:number*
:[range]nu[mber] [count] [flags]
Same as :print, but precede each line with its line
number. (See also 'highlight' and 'numberwidth'
option).
See |ex-flags| for [flags].

:#
:[range]# [count] [flags]
synonym for :number.

:#!
:#!{anything} Ignored, so that you can start a Vim script with: >
#!vim -S
echo "this is a Vim script"
quit
<

:z *E144*
:{range}z[+-^.=]{count} Display several lines of text surrounding the line
specified with {range}, or around the current line
if there is no {range}. If there is a {count}, that's
how many lines you'll see; if there is only one window
then twice the value of the 'scroll' option is used,
otherwise the current window height minus 3 is used.

If there is a {count} the 'window' option is set to
its value.

:z can be used either alone or followed by any of
several punctuation marks. These have the following
effect:

mark	first line	last line	new cursor line ~
+	current line	1 scr forward	1 scr forward
-	1 scr back	current line	current line
^	2 scr back	1 scr back	1 scr back
.	1/2 scr back	1/2 scr fwd	1/2 scr fwd
=	1/2 scr back	1/2 scr fwd	current line

Specifying no mark at all is the same as "+".
If the mark is "=", a line of dashes is printed
around the current line.

:z#
:{range}z#[+-^.=]{count}
Like ":z", but number the lines.
{not in all versions of Vi, not with these arguments}

:=
:= [flags] Print the last line number.
See |ex-flags| for [flags].

:{range}= [flags] Prints the last line number in {range}. For example,
this prints the current line number: >
:.=
< See |ex-flags| for [flags].

:norm *:normal*
:norm[al][!] {commands}
Execute Normal mode commands {commands}. This makes
it possible to execute Normal mode commands typed on
the command-line. {commands} are executed like they
are typed. For undo all commands are undone together.
Execution stops when an error is encountered.

If the [!] is given, mappings will not be used.
Without it, when this command is called from a
non-remappable mapping (|:noremap|), the argument can
be mapped anyway.

{commands} should be a complete command. If
{commands} does not finish a command, the last one
will be aborted as if <Esc> or <C-C> was typed.

This implies that an insert command must be completed (to start Insert mode, see |:startinsert|). A ":" command must be completed as well. And you can't use "Q" or "gQ" to start Ex mode.

The display is not updated while ":normal" is busy.

{commands} cannot start with a space. Put a count of 1 (one) before it, "1 " is one space.

The 'insertmode' option is ignored for {commands}.

This command cannot be followed by another command, since any '|' is considered part of the command.

This command can be used recursively, but the depth is limited by 'maxmapdepth'.

An alternative is to use |:execute|, which uses an expression as argument. This allows the use of printable characters to represent special characters.

Example: >
```
	:exe "normal \<c-w>\<c-w>"
<
```
{not in Vi, of course}

:{range}norm[al][!] {commands} *:normal-range*

Execute Normal mode commands {commands} for each line in the {range}. Before executing the {commands}, the cursor is positioned in the first column of the range, for each line. Otherwise it's the same as the ":normal" command without a range.
{not in Vi}

:sh *:shell* *E371*

:sh[ell] This command starts a shell. When the shell exits (after the "exit" command) you return to Vim. The name for the shell command comes from 'shell' option.
E360

Note: This doesn't work when Vim on the Amiga was started in QuickFix mode from a compiler, because the compiler will have set stdin to a non-interactive mode.

:!cmd *:!* *E34*

:!{cmd} Execute {cmd} with the shell. See also the 'shell' and 'shelltype' option.

Any '!' in {cmd} is replaced with the previous external command (see also 'cpoptions'). But not when there is a backslash before the '!', then that backslash is removed. Example: ":!ls" followed by ":!echo ! \! \\!" executes "echo ls ! \!".

A '|' in {cmd} is passed to the shell, you cannot use it to append a Vim command. See |:bar|.

If {cmd} contains "%" it is expanded to the current file name. Special characters are not escaped, use quotes to avoid their special meaning: >
```
	:!ls "%"
<
```
If the file name contains a "$" single quotes might

```
                        work better (but a single quote causes trouble): >
                                :!ls '%'
<                       This should always work, but it's more typing: >
                                :exe "!ls " . shellescape(expand("%"))
<
                        A newline character ends {cmd}, what follows is
                        interpreted as a following ":" command.  However, if
                        there is a backslash before the newline it is removed
                        and {cmd} continues.  It doesn't matter how many
                        backslashes are before the newline, only one is
                        removed.

                        On Unix the command normally runs in a non-interactive
                        shell.  If you want an interactive shell to be used
                        (to use aliases) set 'shellcmdflag' to "-ic".
                        For Win32 also see |:!start|.

                        After the command has been executed, the timestamp and
                        size of the current file is checked |timestamp|.

                        Vim redraws the screen after the command is finished,
                        because it may have printed any text.  This requires a
                        hit-enter prompt, so that you can read any messages.
                        To avoid this use: >
                                :silent !{cmd}
<                       The screen is not redrawn then, thus you have to use
                        CTRL-L or ":redraw!" if the command did display
                        something.
                        Also see |shell-window|.

                                                        *:!!*
:!!                     Repeat last ":!{cmd}".

                                                        *:ve* *:version*
:ve[rsion]              Print the version number of the editor.  If the
                        compiler used understands "__DATE__" the compilation
                        date is mentioned.  Otherwise a fixed release-date is
                        shown.
                        The following lines contain information about which
                        features were enabled when Vim was compiled.  When
                        there is a preceding '+', the feature is included,
                        when there is a '-' it is excluded.  To change this,
                        you have to edit feature.h and recompile Vim.
                        To check for this in an expression, see |has()|.
                        Here is an overview of the features.
                        The first column shows the smallest version in which
                        they are included:
                           T    tiny
                           S    small
                           N    normal
                           B    big
                           H    huge
                           m    manually enabled or depends on other features
                         (none) system dependent
                        Thus if a feature is marked with "N", it is included
                        in the normal, big and huge versions of Vim.

                                                        *+feature-list*
   *+acl*               |ACL| support included
   *+ARP*               Amiga only: ARP support included
B  *+arabic*            |Arabic| language support
N  *+autocmd*           |:autocmd|, automatic commands
```

```
m  *+balloon_eval*      |balloon-eval| support. Included when compiling with
                        supported GUI (Motif, GTK, GUI) and either
                        Netbeans/Sun Workshop integration or |+eval| feature.
N  *+browse*            |:browse| command
N  *+builtin_terms*     some terminals builtin |builtin-terms|
B  *++builtin_terms*    maximal terminals builtin |builtin-terms|
N  *+byte_offset*       support for 'o' flag in 'statusline' option, "go"
                        and ":goto" commands.
m  *+channel*           inter process communication |channel|
N  *+cindent*           |'cindent'|, C indenting
N  *+clientserver*      Unix and Win32: Remote invocation |clientserver|
   *+clipboard*         |clipboard| support
N  *+cmdline_compl*     command line completion |cmdline-completion|
S  *+cmdline_hist*      command line history |cmdline-history|
N  *+cmdline_info*      |'showcmd'| and |'ruler'|
N  *+comments*          |'comments'| support
B  *+conceal*           "conceal" support, see |conceal| |:syn-conceal| etc.
N  *+cryptv*            encryption support |encryption|
B  *+cscope*            |cscope| support
m  *+cursorbind*        |'cursorbind'| support
m  *+cursorshape*       |termcap-cursor-shape| support
m  *+debug*             Compiled for debugging.
N  *+dialog_gui*        Support for |:confirm| with GUI dialog.
N  *+dialog_con*        Support for |:confirm| with console dialog.
N  *+dialog_con_gui*    Support for |:confirm| with GUI and console dialog.
N  *+diff*              |vimdiff| and 'diff'
N  *+digraphs*          |digraphs| *E196*
m  *+directx*           Win32 GUI only: DirectX and |'renderoptions'|
   *+dnd*               Support for DnD into the "~ register |quote_~|.
B  *+emacs_tags*        |emacs-tags| files
N  *+eval*              expression evaluation |eval.txt|
N  *+ex_extra*          always on now, used to be for Vim's extra Ex commands
N  *+extra_search*      |'hlsearch'| and |'incsearch'| options.
B  *+farsi*             |farsi| language
N  *+file_in_path*      |gf|, |CTRL-W_f| and |<cfile>|
N  *+find_in_path*      include file searches: |[I|, |:isearch|,
                        |CTRL-W_CTRL-I|, |:checkpath|, etc.
N  *+folding*           |folding|
   *+footer*            |gui-footer|
   *+fork*              Unix only: |fork| shell commands
   *+float*             Floating point support
N  *+gettext*           message translations |multi-lang|
   *+GUI_Athena*        Unix only: Athena |GUI|
   *+GUI_neXtaw*        Unix only: neXtaw |GUI|
   *+GUI_GTK*           Unix only: GTK+ |GUI|
   *+GUI_Motif*         Unix only: Motif |GUI|
   *+GUI_Photon*        QNX only:  Photon |GUI|
m  *+hangul_input*      Hangul input support |hangul|
   *+iconv*             Compiled with the |iconv()| function
   *+iconv/dyn*         Likewise |iconv-dynamic| |/dyn|
N  *+insert_expand*     |insert_expand| Insert mode completion
m  *+job*               starting and stopping jobs |job|
S  *+jumplist*          |jumplist|
B  *+keymap*            |'keymap'|
N  *+lambda*            |lambda| and |closure|
B  *+langmap*           |'langmap'|
N  *+libcall*           |libcall()|
N  *+linebreak*         |'linebreak'|, |'breakat'| and |'showbreak'|
N  *+lispindent*        |'lisp'|
N  *+listcmds*          Vim commands for the list of buffers |buffer-hidden|
                        and argument list |:argdelete|
N  *+localmap*          Support for mappings local to a buffer |:map-local|
```

```
m  *+lua*               |Lua| interface
m  *+lua/dyn*           |Lua| interface |/dyn|
N  *+menu*              |:menu|
N  *+mksession*         |:mksession|
N  *+modify_fname*      |filename-modifiers|
N  *+mouse*             Mouse handling |mouse-using|
N  *+mouseshape*        |'mouseshape'|
B  *+mouse_dec*         Unix only: Dec terminal mouse handling |dec-mouse|
N  *+mouse_gpm*         Unix only: Linux console mouse handling |gpm-mouse|
N  *+mouse_jsbterm*     JSB mouse handling |jsbterm-mouse|
B  *+mouse_netterm*     Unix only: netterm mouse handling |netterm-mouse|
N  *+mouse_pterm*       QNX only: pterm mouse handling |qnx-terminal|
N  *+mouse_sysmouse*    Unix only: *BSD console mouse handling |sysmouse|
B  *+mouse_sgr*         Unix only: sgr mouse handling |sgr-mouse|
B  *+mouse_urxvt*       Unix only: urxvt mouse handling |urxvt-mouse|
N  *+mouse_xterm*       Unix only: xterm mouse handling |xterm-mouse|
N  *+multi_byte*        16 and 32 bit characters |multibyte|
   *+multi_byte_ime*    Win32 input method for multibyte chars |multibyte-ime|
N  *+multi_lang*        non-English language support |multi-lang|
m  *+mzscheme*          Mzscheme interface |mzscheme|
m  *+mzscheme/dyn*      Mzscheme interface |mzscheme-dynamic| |/dyn|
m  *+netbeans_intg*     |netbeans|
   *+num64*             64-bit Number support |Number|
m  *+ole*               Win32 GUI only: |ole-interface|
N  *+packages*          Loading |packages|
N  *+path_extra*        Up/downwards search in 'path' and 'tags'
m  *+perl*              Perl interface |perl|
m  *+perl/dyn*          Perl interface |perl-dynamic| |/dyn|
N  *+persistent_undo*   Persistent undo |undo-persistence|
   *+postscript*        |:hardcopy| writes a PostScript file
N  *+printer*           |:hardcopy| command
H  *+profile*           |:profile| command
m  *+python*            Python 2 interface |python|
m  *+python/dyn*        Python 2 interface |python-dynamic| |/dyn|
m  *+python3*           Python 3 interface |python|
m  *+python3/dyn*       Python 3 interface |python-dynamic| |/dyn|
N  *+quickfix*          |:make| and |quickfix| commands
N  *+reltime*           |reltime()| function, 'hlsearch'/'incsearch' timeout,
                        'redrawtime' option
B  *+rightleft*         Right to left typing |'rightleft'|
m  *+ruby*              Ruby interface |ruby|
m  *+ruby/dyn*          Ruby interface |ruby-dynamic| |/dyn|
N  *+scrollbind*        |'scrollbind'|
B  *+signs*             |:sign|
N  *+smartindent*       |'smartindent'|
N  *+startuptime*       |--startuptime| argument
N  *+statusline*        Options 'statusline', 'rulerformat' and special
                        formats of 'titlestring' and 'iconstring'
m  *+sun_workshop*      |workshop|
N  *+syntax*            Syntax highlighting |syntax|
   *+system()*          Unix only: opposite of |+fork|
T  *+tag_binary*        binary searching in tags file |tag-binary-search|
N  *+tag_old_static*    old method for static tags |tag-old-static|
m  *+tag_any_white*     any white space allowed in tags file |tag-any-white|
m  *+tcl*               Tcl interface |tcl|
m  *+tcl/dyn*           Tcl interface |tcl-dynamic| |/dyn|
m  *+terminal*          Support for terminal window |terminal|
   *+terminfo*          uses |terminfo| instead of termcap
N  *+termresponse*      support for |t_RV| and |v:termresponse|
B  *+termguicolors*     24-bit color in xterm-compatible terminals support
N  *+textobjects*       |text-objects| selection
   *+tgetent*           non-Unix only: able to use external termcap
```

```
N  *+timers*            the |timer_start()| function
N  *+title*             Setting the window 'title' and 'icon'
N  *+toolbar*           |gui-toolbar|
N  *+user_commands*     User-defined commands. |user-commands|
N  *+viminfo*           |'viminfo'|
   *+vertsplit*         Vertically split windows |:vsplit|; Always enabled
                        since 8.0.1118.
                        in sync with the |+windows| feature
N  *+virtualedit*       |'virtualedit'|
S  *+visual*            Visual mode |Visual-mode| Always enabled since 7.4.200.
N  *+visualextra*       extra Visual mode commands |blockwise-operators|
N  *+vreplace*          |gR| and |gr|
N  *+wildignore*        |'wildignore'|
N  *+wildmenu*          |'wildmenu'|
   *+windows*           more than one window; Always enabled sinde 8.0.1118.
m  *+writebackup*       |'writebackup'| is default on
m  *+xim*               X input method |xim|
   *+xfontset*          X fontset support |xfontset|
   *+xpm*               pixmap support
m  *+xpm_w32*           Win32 GUI only: pixmap support |w32-xpm-support|
   *+xsmp*              XSMP (X session management) support
   *+xsmp_interact*     interactive XSMP (X session management) support
N  *+xterm_clipboard*   Unix only: xterm clipboard handling
m  *+xterm_save*        save and restore xterm screen |xterm-screens|
N  *+X11*               Unix only: can restore window title |X11|

                                                */dyn* *E370* *E448*
                        To some of the features "/dyn" is added when the
                        feature is only available when the related library can
                        be dynamically loaded.

:ve[rsion] {nr}         Is now ignored.  This was previously used to check the
                        version number of a .vimrc file.  It was removed,
                        because you can now use the ":if" command for
                        version-dependent behavior.  {not in Vi}

                                                *:redi* *:redir*
:redi[r][!] > {file}    Redirect messages to file {file}.  The messages which
                        are the output of commands are written to that file,
                        until redirection ends.  The messages are also still
                        shown on the screen.  When [!] is included, an
                        existing file is overwritten.  When [!] is omitted,
                        and {file} exists, this command fails.

                        Only one ":redir" can be active at a time.  Calls to
                        ":redir" will close any active redirection before
                        starting redirection to the new target.  For recursive
                        use check out |execute()|.

                        To stop the messages and commands from being echoed to
                        the screen, put the commands in a function and call it
                        with ":silent call Function()".
                        An alternative is to use the 'verbosefile' option,
                        this can be used in combination with ":redir".
                        {not in Vi}

:redi[r] >> {file}      Redirect messages to file {file}.  Append if {file}
                        already exists.  {not in Vi}

:redi[r] @{a-zA-Z}
:redi[r] @{a-zA-Z}>     Redirect messages to register {a-z}.  Append to the
                        contents of the register if its name is given
```

```
                        uppercase {A-Z}.  The ">" after the register name is
                        optional. {not in Vi}
:redi[r] @{a-z}>>       Append messages to register {a-z}. {not in Vi}

:redi[r] @*>
:redi[r] @+>            Redirect messages to the selection or clipboard. For
                        backward compatibility, the ">" after the register
                        name can be omitted. See |quotestar| and |quoteplus|.
                        {not in Vi}
:redi[r] @*>>
:redi[r] @+>>           Append messages to the selection or clipboard.
                        {not in Vi}

:redi[r] @">            Redirect messages to the unnamed register. For
                        backward compatibility, the ">" after the register
                        name can be omitted. {not in Vi}
:redi[r] @">>           Append messages to the unnamed register. {not in Vi}

:redi[r] => {var}       Redirect messages to a variable.  If the variable
                        doesn't exist, then it is created.  If the variable
                        exists, then it is initialized to an empty string.
                        The variable will remain empty until redirection ends.
                        Only string variables can be used.  After the
                        redirection starts, if the variable is removed or
                        locked or the variable type is changed, then further
                        command output messages will cause errors. {not in Vi}
                        To get the output of one command the |execute()|
                        function can be used.

:redi[r] =>> {var}      Append messages to an existing variable.  Only string
                        variables can be used. {not in Vi}

:redi[r] END            End redirecting messages.  {not in Vi}

                                                        *:filt* *:filter*
:filt[er][!] {pat} {command}
:filt[er][!] /{pat}/ {command}
                        Restrict the output of {command} to lines matching
                        with {pat}.  For example, to list only xml files: >
                                :filter /\.xml$/ oldfiles
<                       If the [!] is given, restrict the output of {command}
                        to lines that do NOT match {pat}.

                        {pat} 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 {pat}.  Without
                        the enclosing character the pattern cannot include the
                        bar character.

                        The pattern is matched against the relevant part of
                        the output, not necessarily the whole line. Only some
                        commands support filtering, try it out to check if it
                        works.

                        Only normal messages are filtered, error messages are
                        not.

                                                *:sil* *:silent* *:silent!*
:sil[ent][!] {command}  Execute {command} silently.  Normal messages will not
                        be given or added to the message history.
                        When [!] is added, error messages will also be
                        skipped, and commands and mappings will not be aborted
```

```
                        when an error is detected.  |v:errmsg| is still set.
                        When [!] is not used, an error message will cause
                        further messages to be displayed normally.
                        Redirection, started with |:redir|, will continue as
                        usual, although there might be small differences.
                        This will allow redirecting the output of a command
                        without seeing it on the screen.  Example: >
                            :redir >/tmp/foobar
                            :silent g/Aap/p
                            :redir END
<                       To execute a Normal mode command silently, use the
                        |:normal| command.  For example, to search for a
                        string without messages: >
                            :silent exe "normal /path\<CR>"
<                       ":silent!" is useful to execute a command that may
                        fail, but the failure is to be ignored.  Example: >
                            :let v:errmsg = ""
                            :silent! /^begin
                            :if v:errmsg != ""
                            : ... pattern was not found
<                       ":silent" will also avoid the hit-enter prompt.  When
                        using this for an external command, this may cause the
                        screen to be messed up.  Use |CTRL-L| to clean it up
                        then.
                        ":silent menu ..." defines a menu that will not echo a
                        Command-line command.  The command will still produce
                        messages though.  Use ":silent" in the command itself
                        to avoid that: ":silent menu .... :silent command".

                                                    *:uns* *:unsilent*
:uns[ilent] {command}   Execute {command} not silently.  Only makes a
                        difference when |:silent| was used to get to this
                        command.
                        Use this for giving a message even when |:silent| was
                        used.  In this example |:silent| is used to avoid the
                        message about reading the file and |:unsilent| to be
                        able to list the first line of each file. >
                :silent argdo unsilent echo expand('%') . ": " . getline(1)
<

                                                    *:verb* *:verbose*
:[count]verb[ose] {command}
                        Execute {command} with 'verbose' set to [count].  If
                        [count] is omitted one is used. ":0verbose" can be
                        used to set 'verbose' to zero.
                        The additional use of ":silent" makes messages
                        generated but not displayed.
                        The combination of ":silent" and ":verbose" can be
                        used to generate messages and check them with
                        |v:statusmsg| and friends.  For example: >
                                :let v:statusmsg = ""
                                :silent verbose runtime foobar.vim
                                :if v:statusmsg != ""
                                :  " foobar.vim could not be found
                                :endif
<                       When concatenating another command, the ":verbose"
                        only applies to the first one: >
                                :4verbose set verbose | set verbose
<                                 verbose=4 ~
                                  verbose=0 ~
                        For logging verbose messages in a file use the
                        'verbosefile' option.
```

:verbose-cmd
When 'verbose' is non-zero, listing the value of a Vim option or a key map or an abbreviation or a user-defined function or a command or a highlight group or an autocommand will also display where it was last defined. If it was defined manually then there will be no "Last set" message. When it was defined while executing a function, user command or autocommand, the script in which it was defined is reported.
{not available when compiled without the |+eval| feature}

K
K Run a program to lookup the keyword under the cursor. The name of the program is given with the 'keywordprg' (kp) option (default is "man"). The keyword is formed of letters, numbers and the characters in 'iskeyword'. The keyword under or right of the cursor is used. The same can be done with the command >
:!{program} {keyword}
< There is an example of a program to use in the tools directory of Vim. It is called "ref" and does a simple spelling check.
Special cases:
- If 'keywordprg' begins with ":" it is invoked as a Vim Ex command with [count].
- If 'keywordprg' is empty, the ":help" command is used. It's a good idea to include more characters in 'iskeyword' then, to be able to find more help.
- When 'keywordprg' is equal to "man" or starts with ":", a [count] before "K" is inserted after keywordprg and before the keyword. For example, using "2K" while the cursor is on "mkdir", results in: >
!man 2 mkdir
< - When 'keywordprg' is equal to "man -s", a count before "K" is inserted after the "-s". If there is no count, the "-s" is removed.
{not in Vi}

v_K
{Visual}K Like "K", but use the visually highlighted text for the keyword. Only works when the highlighted text is not more than one line. {not in Vi}

[N]gs *gs* *:sl* *:sleep*
:[N]sl[eep] [N] [m] Do nothing for [N] seconds. When [m] is included, sleep for [N] milliseconds. The count for "gs" always uses seconds. The default is one second. >
:sleep "sleep for one second
:5sleep "sleep for five seconds
:sleep 100m "sleep for a hundred milliseconds
10gs "sleep for ten seconds
< Can be interrupted with CTRL-C (CTRL-Break on MS-DOS).
"gs" stands for "goto sleep".
While sleeping the cursor is positioned in the text, if at a visible position. {not in Vi}
Also process the received netbeans messages. {only available when compiled with the |+netbeans_intg| feature}

g_CTRL-A

```
g CTRL-A                Only when Vim was compiled with MEM_PROFILING defined
                        (which is very rare): print memory usage statistics.
                        Only useful for debugging Vim.
                        For incrementing in Visual mode see |v_g_CTRL-A|.

==============================================================================
2. Using Vim like less or more                                  *less*

If you use the less or more program to view a file, you don't get syntax
highlighting.  Thus you would like to use Vim instead.  You can do this by
using the shell script "$VIMRUNTIME/macros/less.sh".

This shell script uses the Vim script "$VIMRUNTIME/macros/less.vim".  It sets
up mappings to simulate the commands that less supports.  Otherwise, you can
still use the Vim commands.

This isn't perfect.  For example, when viewing a short file Vim will still use
the whole screen.  But it works good enough for most uses, and you get syntax
highlighting.

The "h" key will give you a short overview of the available commands.

If you want to set options differently when using less, define the
LessInitFunc in your vimrc, for example: >

        func LessInitFunc()
          set nocursorcolumn nocursorline
        endfunc
<

 vim:tw=78:ts=8:ft=help:norl:
*recover.txt*   For Vim version 8.0.  Last change: 2014 Mar 27


                  VIM REFERENCE MANUAL    by Bram Moolenaar


Recovery after a crash                                  *crash-recovery*

You have spent several hours typing in that text that has to be finished
next morning, and then disaster strikes: Your computer crashes.

                        DON'T PANIC!

You can recover most of your changes from the files that Vim uses to store
the contents of the file.  Mostly you can recover your work with one command:
        vim -r filename

1. The swap file        |swap-file|
2. Recovery             |recovery|

==============================================================================
1. The swap file                                        *swap-file*

Vim stores the things you changed in a swap file.  Using the original file
you started from plus the swap file you can mostly recover your work.

You can see the name of the current swap file being used with the command:

        :sw[apname]                                     *:sw* *:swapname*

The name of the swap file is normally the same as the file you are editing,
```

```
with the extension ".swp".
- On Unix, a '.' is prepended to swap file names in the same directory as the
  edited file.  This avoids that the swap file shows up in a directory
  listing.
- On MS-DOS machines and when the 'shortname' option is on, any '.' in the
  original file name is replaced with '_'.
- If this file already exists (e.g., when you are recovering from a crash) a
  warning is given and another extension is used, ".swo", ".swn", etc.
- An existing file will never be overwritten.
- The swap file is deleted as soon as Vim stops editing the file.

Technical: The replacement of '.' with '_' is done to avoid problems with
           MS-DOS compatible filesystems (e.g., crossdos, multidos).  If Vim
           is able to detect that the file is on an MS-DOS-like filesystem, a
           flag is set that has the same effect as the 'shortname' option.
           This flag is reset when you start editing another file.
                                                          *E326*
           If the ".swp" file name already exists, the last character is
           decremented until there is no file with that name or ".saa" is
           reached.  In the last case, no swap file is created.

By setting the 'directory' option you can place the swap file in another place
than where the edited file is.
Advantages:
- You will not pollute the directories with ".swp" files.
- When the 'directory' is on another partition, reduce the risk of damaging
  the file system where the file is (in a crash).
Disadvantages:
- You can get name collisions from files with the same name but in different
  directories (although Vim tries to avoid that by comparing the path name).
  This will result in bogus ATTENTION warning messages.
- When you use your home directory, and somebody else tries to edit the same
  file, he will not see your swap file and will not get the ATTENTION warning
  message.
On the Amiga you can also use a recoverable ram disk, but there is no 100%
guarantee that this works.  Putting swap files in a normal ram disk (like RAM:
on the Amiga) or in a place that is cleared when rebooting (like /tmp on Unix)
makes no sense, you will lose the swap file in a crash.

If you want to put swap files in a fixed place, put a command resembling the
following ones in your .vimrc:
        :set dir=dh2:tmp        (for Amiga)
        :set dir=~/tmp          (for Unix)
        :set dir=c:\\tmp        (for MS-DOS and Win32)
This is also very handy when editing files on floppy.  Of course you will have
to create that "tmp" directory for this to work!

For read-only files, a swap file is not used.  Unless the file is big, causing
the amount of memory used to be higher than given with 'maxmem' or
'maxmemtot'.  And when making a change to a read-only file, the swap file is
created anyway.

The 'swapfile' option can be reset to avoid creating a swapfile.  And the
|:noswapfile| modifier can be used to not create a swapfile for a new buffer.

:nos[wapfile]   {command}                       *:nos* *:noswapfile*
                Execute {command}. If it contains a command that loads a new
                buffer, it will be loaded without creating a swapfile and the
                'swapfile' option will be reset.  If a buffer already had a
                swapfile it is not removed and 'swapfile' is not reset.
```

Detecting an existing swap file ~

You can find this in the user manual, section |11.3|.

Updating the swapfile ~

The swap file is updated after typing 200 characters or when you have not typed anything for four seconds. This only happens if the buffer was changed, not when you only moved around. The reason why it is not kept up to date all the time is that this would slow down normal work too much. You can change the 200 character count with the 'updatecount' option. You can set the time with the 'updatetime' option. The time is given in milliseconds. After writing to the swap file Vim syncs the file to disk. This takes some time, especially on busy Unix systems. If you don't want this you can set the 'swapsync' option to an empty string. The risk of losing work becomes bigger though. On some non-Unix systems (MS-DOS, Amiga) the swap file won't be written at all.

If the writing to the swap file is not wanted, it can be switched off by setting the 'updatecount' option to 0. The same is done when starting Vim with the "-n" option. Writing can be switched back on by setting the 'updatecount' option to non-zero. Swap files will be created for all buffers when doing this. But when setting 'updatecount' to zero, the existing swap files will not be removed, it will only affect files that will be opened after this.

If you want to make sure that your changes are in the swap file use this command:

:pre *:preserve* *E313* *E314*

:pre[serve] Write all text for all buffers into swap file. The original file is no longer needed for recovery. This sets a flag in the current buffer. When the '&' flag is present in 'cpoptions' the swap file will not be deleted for this buffer when Vim exits and the buffer is still loaded |cpo-&|. {Vi: might also exit}

A Vim swap file can be recognized by the first six characters: "b0VIM ". After that comes the version number, e.g., "3.0".

Links and symbolic links ~

On Unix it is possible to have two names for the same file. This can be done with hard links and with symbolic links (symlinks).

For hard links Vim does not know the other name of the file. Therefore, the name of the swapfile will be based on the name you used to edit the file. There is no check for editing the same file by the other name too, because Vim cannot find the other swapfile (except for searching all of your harddisk, which would be very slow).

For symbolic links Vim resolves the links to find the name of the actual file. The swap file name is based on that name. Thus it doesn't matter by what name you edit the file, the swap file name will normally be the same. However, there are exceptions:
- When the directory of the actual file is not writable the swapfile is put elsewhere.
- When the symbolic links somehow create a loop you get an *E773* error message and the unmodified file name will be used. You won't be able to

save your file normally.

==
2. Recovery *recovery* *E308* *E311*

Basic file recovery is explained in the user manual: |usr_11.txt|.

Another way to do recovery is to start Vim and use the ":recover" command.
This is easy when you start Vim to edit a file and you get the "ATTENTION:
Found a swap file ..." message. In this case the single command ":recover"
will do the work. You can also give the name of the file or the swap file to
the recover command:

:rec *:recover* *E305* *E306* *E307*

:rec[over] [file] Try to recover [file] from the swap file. If [file] is not given use the file name for the current buffer. The current contents of the buffer are lost. This command fails if the buffer was modified.

:rec[over]! [file] Like ":recover", but any changes in the current buffer are lost.

E312 *E309* *E310*

Vim has some intelligence about what to do if the swap file is corrupt in
some way. If Vim has doubt about what it found, it will give an error
message and insert lines with "???" in the text. If you see an error message
while recovering, search in the file for "???" to see what is wrong. You may
want to cut and paste to get the text you need.

The most common remark is "???LINES MISSING". This means that Vim cannot read
the text from the original file. This can happen if the system crashed and
parts of the original file were not written to disk.

Be sure that the recovery was successful before overwriting the original
file or deleting the swap file. It is good practice to write the recovered
file elsewhere and run 'diff' to find out if the changes you want are in the
recovered file. Or use |:DiffOrig|.

Once you are sure the recovery is ok delete the swap file. Otherwise, you
will continue to get warning messages that the ".swp" file already exists.

{Vi: recovers in another way and sends mail if there is something to recover}

ENCRYPTION AND THE SWAP FILE *:recover-crypt*

When the text file is encrypted the swap file is encrypted as well. This
makes recovery a bit more complicated. When recovering from a swap file and
encryption has been used, you will be asked to enter one or two crypt keys.

If the text file does not exist you will only be asked to enter the crypt key
for the swap file.

If the text file does exist, it may be encrypted in a different way than the
swap file. You will be asked for the crypt key twice:

```
Need encryption key for "/tmp/tt" ~
Enter encryption key: ****** ~
"/tmp/tt" [crypted] 23200L, 522129C ~
Using swap file "/tmp/.tt.swp" ~
Original file "/tmp/tt" ~
Swap file is encrypted: "/tmp/.tt.swp" ~
If you entered a new crypt key but did not write the text file, ~
```

```
        enter the new crypt key. ~
        If you wrote the text file after changing the crypt key press enter ~
        to use the same key for text file and swap file ~
        Enter encryption key:  ~

You can be in one of these two situations:

1. The encryption key was not changed, or after changing the key the text file
   was written.  You will be prompted for the crypt key twice.  The second
   time you can simply press Enter.  That means the same key is used for the
   text file and the swap file.
2. You entered a new encryption key, but did not save the text file.  Vim will
   then use the new key for the swap file, and the text file will still be
   encrypted with the old key.  At the second prompt enter the new key.

Note that after recovery the key of the swap file will be used for the text
file.  Thus if you write the text file, you need to use that new key.


 vim:tw=78:ts=8:ft=help:norl:
```