Tuning Vim

==

usr_40.txt   For Vim version 8.0. Last change: 2013 Aug 05

VIM USER MANUAL - by Bram Moolenaar

Make new commands

Vim is an extensible editor. You can take a sequence of commands you use often and turn it into a new command. Or redefine an existing command. Autocommands make it possible to execute commands automatically.

     Next chapter: |usr_41.txt| Write a Vim script
 Previous chapter: |usr_32.txt| The undo tree
Table of contents: |usr_toc.txt|

==

40.1 Key mapping

A simple mapping was explained in section |05.3|. The principle is that one sequence of key strokes is translated into another sequence of key strokes. This is a simple, yet powerful mechanism.

The simplest form is that one key is mapped to a sequence of keys. Since the function keys, except <F1>, have no predefined meaning in Vim, these are good choices to map. Example: >

```
	:map <F2> GoDate: <Esc>:read !date<CR>kJ
```

This shows how three modes are used. After going to the last line with "G", the "o" command opens a new line and starts Insert mode. The text "Date: " is inserted and <Esc> takes you out of insert mode.

Notice the use of special keys inside <>. This is called angle bracket notation. You type these as separate characters, not by pressing the key itself. This makes the mappings better readable and you can copy and paste the text without problems.

The ":" character takes Vim to the command line. The ":read !date" command reads the output from the "date" command and appends it below the current line. The <CR> is required to execute the ":read" command.

At this point of execution the text looks like this:

```
	Date:  ~
	Fri Jun 15 12:54:34 CEST 2001 ~
```

Now "kJ" moves the cursor up and joins the lines together.

To decide which key or keys you use for mapping, see |map-which-keys|.

MAPPING AND MODES

The ":map" command defines remapping for keys in Normal mode. You can also define mappings for other modes. For example, ":imap" applies to Insert mode. You can use it to insert a date below the cursor: >

```
        :imap <F2> <CR>Date: <Esc>:read !date<CR>kJ
```

It looks a lot like the mapping for <F2> in Normal mode, only the start is different. The <F2> mapping for Normal mode is still there. Thus you can map the same key differently for each mode.

Notice that, although this mapping starts in Insert mode, it ends in Normal mode. If you want it to continue in Insert mode, append an "a" to the mapping.

Here is an overview of map commands and in which mode they work:

:map	Normal, Visual and Operator-pending
:vmap	Visual
:nmap	Normal
:omap	Operator-pending
:map!	Insert and Command-line
:imap	Insert
:cmap	Command-line

Operator-pending mode is when you typed an operator character, such as "d" or "y", and you are expected to type the motion command or a text object. Thus when you type "dw", the "w" is entered in operator-pending mode.

Suppose that you want to define <F7> so that the command d<F7> deletes a C program block (text enclosed in curly braces, {}). Similarly y<F7> would yank the program block into the unnamed register. Therefore, what you need to do is to define <F7> to select the current program block. You can do this with the following command: >

```
        :omap <F7> a{
```

This causes <F7> to perform a select block "a{" in operator-pending mode, just like you typed it. This mapping is useful if typing a { on your keyboard is a bit difficult.

LISTING MAPPINGS

To see the currently defined mappings, use ":map" without arguments. Or one of the variants that include the mode in which they work. The output could look like this:

```
           _g            :call MyGrep(1)<CR> ~
        v  <F2>          :s/^/> /<CR>:noh<CR>`` ~
        n  <F2>          :.,$s/^/> /<CR>:noh<CR>`` ~
           <xHome>       <Home>
           <xEnd>        <End>
```

The first column of the list shows in which mode the mapping is effective. This is "n" for Normal mode, "i" for Insert mode, etc. A blank is used for a mapping defined with ":map", thus effective in both Normal and Visual mode.

One useful purpose of listing the mapping is to check if special keys in <> form have been recognized (this only works when color is supported). For example, when <Esc> is displayed in color, it stands for the escape character. When it has the same color as the other text, it is five characters.

REMAPPING

The result of a mapping is inspected for other mappings in it. For example, the mappings for <F2> above could be shortened to: >

```
	:map <F2> G<F3>
	:imap <F2> <Esc><F3>
	:map <F3>  oDate: <Esc>:read !date<CR>kJ
```

For Normal mode <F2> is mapped to go to the last line, and then behave like <F3> was pressed. In Insert mode <F2> stops Insert mode with <Esc> and then also uses <F3>. Then <F3> is mapped to do the actual work.

Suppose you hardly ever use Ex mode, and want to use the "Q" command to format text (this was so in old versions of Vim). This mapping will do it: >

```
	:map Q gq
```

But, in rare cases you need to use Ex mode anyway. Let's map "gQ" to Q, so that you can still go to Ex mode: >

```
	:map gQ Q
```

What happens now is that when you type "gQ" it is mapped to "Q". So far so good. But then "Q" is mapped to "gq", thus typing "gQ" results in "gq", and you don't get to Ex mode at all.

To avoid keys to be mapped again, use the ":noremap" command: >

```
	:noremap gQ Q
```

Now Vim knows that the "Q" is not to be inspected for mappings that apply to it. There is a similar command for every mode:

:noremap	Normal, Visual and Operator-pending
:vnoremap	Visual
:nnoremap	Normal
:onoremap	Operator-pending
:noremap!	Insert and Command-line
:inoremap	Insert
:cnoremap	Command-line

RECURSIVE MAPPING

When a mapping triggers itself, it will run forever. This can be used to repeat an action an unlimited number of times.

For example, you have a list of files that contain a version number in the first line. You edit these files with "vim *.txt". You are now editing the first file. Define this mapping: >

```
	:map ,, :s/5.1/5.2/<CR>:wnext<CR>,,
```

Now you type ",,". This triggers the mapping. It replaces "5.1" with "5.2" in the first line. Then it does a ":wnext" to write the file and edit the next one. The mapping ends in ",,". This triggers the same mapping again, thus doing the substitution, etc.

This continues until there is an error. In this case it could be a file where the substitute command doesn't find a match for "5.1". You can then make a change to insert "5.1" and continue by typing ",," again. Or the ":wnext" fails, because you are in the last file in the list.

When a mapping runs into an error halfway, the rest of the mapping is discarded. CTRL-C interrupts the mapping (CTRL-Break on MS-Windows).

DELETE A MAPPING

To remove a mapping use the ":unmap" command. Again, the mode the unmapping applies to depends on the command used:

:unmap	Normal, Visual and Operator-pending
:vunmap	Visual
:nunmap	Normal
:ounmap	Operator-pending
:unmap!	Insert and Command-line
:iunmap	Insert
:cunmap	Command-line

There is a trick to define a mapping that works in Normal and Operator-pending mode, but not in Visual mode. First define it for all three modes, then delete it for Visual mode: >

```
        :map <C-A> /---><CR>
        :vunmap <C-A>
```

Notice that the five characters "<C-A>" stand for the single key CTRL-A.

To remove all mappings use the |:mapclear| command. You can guess the variations for different modes by now. Be careful with this command, it can't be undone.

SPECIAL CHARACTERS

The ":map" command can be followed by another command. A | character separates the two commands. This also means that a | character can't be used inside a map command. To include one, use <Bar> (five characters). Example:
>

```
        :map <F8> :write <Bar> !checkin %:S<CR>
```

The same problem applies to the ":unmap" command, with the addition that you have to watch out for trailing white space. These two commands are different:
>

```
        :unmap a | unmap b
        :unmap a| unmap b
```

The first command tries to unmap "a ", with a trailing space.

When using a space inside a mapping, use <Space> (seven characters): >

```
        :map <Space> W
```

This makes the spacebar move a blank-separated word forward.

It is not possible to put a comment directly after a mapping, because the " character is considered to be part of the mapping. You can use |", this starts a new, empty command with a comment. Example: >

```
        :map <Space> W|     " Use spacebar to move forward a word
```

MAPPINGS AND ABBREVIATIONS

Abbreviations are a lot like Insert mode mappings. The arguments are handled in the same way. The main difference is the way they are triggered. An abbreviation is triggered by typing a non-word character after the word. A mapping is triggered when typing the last character.

Another difference is that the characters you type for an abbreviation are inserted in the text while you type them. When the abbreviation is triggered these characters are deleted and replaced by what the abbreviation produces. When typing the characters for a mapping, nothing is inserted until you type the last character that triggers it. If the 'showcmd' option is set, the typed characters are displayed in the last line of the Vim window.

An exception is when a mapping is ambiguous. Suppose you have done two mappings: >

```
        :imap aa foo
        :imap aaa bar
```

Now, when you type "aa", Vim doesn't know if it should apply the first or the second mapping. It waits for another character to be typed. If it is an "a", the second mapping is applied and results in "bar". If it is a space, for example, the first mapping is applied, resulting in "foo", and then the space is inserted.

ADDITIONALLY...

The <script> keyword can be used to make a mapping local to a script. See |:map-<script>|.

The <buffer> keyword can be used to make a mapping local to a specific buffer. See |:map-<buffer>|

The <unique> keyword can be used to make defining a new mapping fail when it already exists. Otherwise a new mapping simply overwrites the old one. See |:map-<unique>|.

To make a key do nothing, map it to <Nop> (five characters). This will make the <F7> key do nothing at all: >

```
        :map <F7> <Nop>| map! <F7> <Nop>
```

There must be no space after <Nop>.

==

40.2 Defining command-line commands

The Vim editor enables you to define your own commands. You execute these commands just like any other Command-line mode command.

To define a command, use the ":command" command, as follows: >

```
        :command DeleteFirst 1delete
```

Now when you execute the command ":DeleteFirst" Vim executes ":1delete", which deletes the first line.

> Note:
> User-defined commands must start with a capital letter. You cannot use ":X", ":Next" and ":Print". The underscore cannot be used! You can use digits, but this is discouraged.

To list the user-defined commands, execute the following command: >

```
        :command
```

Just like with the builtin commands, the user defined commands can be abbreviated. You need to type just enough to distinguish the command from another. Command line completion can be used to get the full name.

NUMBER OF ARGUMENTS

User-defined commands can take a series of arguments. The number of arguments must be specified by the -nargs option. For instance, the example :DeleteFirst command takes no arguments, so you could have defined it as follows: >

```
	:command -nargs=0 DeleteFirst 1delete
```

However, because zero arguments is the default, you do not need to add "-nargs=0". The other values of -nargs are as follows:

-nargs=0	No arguments
-nargs=1	One argument
-nargs=*	Any number of arguments
-nargs=?	Zero or one argument
-nargs=+	One or more arguments

USING THE ARGUMENTS

Inside the command definition, the arguments are represented by the <args> keyword. For example: >

```
	:command -nargs=+ Say :echo "<args>"
```

Now when you type >

```
	:Say Hello World
```

Vim echoes "Hello World". However, if you add a double quote, it won't work. For example: >

```
	:Say he said "hello"
```

To get special characters turned into a string, properly escaped to use as an expression, use "<q-args>": >

```
	:command -nargs=+ Say :echo <q-args>
```

Now the above ":Say" command will result in this to be executed: >

```
	:echo "he said \"hello\""
```

The <f-args> keyword contains the same information as the <args> keyword, except in a format suitable for use as function call arguments. For example:
>

```
	:command -nargs=* DoIt :call AFunction(<f-args>)
	:DoIt a b c
```

Executes the following command: >

```
	:call AFunction("a", "b", "c")
```

LINE RANGE

Some commands take a range as their argument. To tell Vim that you are defining such a command, you need to specify a -range option. The values for this option are as follows:

```
	-range		Range is allowed; default is the current line.
	-range=%	Range is allowed; default is the whole file.
	-range={count}	Range is allowed; the last number in it is used as a
			single number whose default is {count}.
```

When a range is specified, the keywords <line1> and <line2> get the values of the first and last line in the range. For example, the following command defines the SaveIt command, which writes out the specified range to the file "save_file": >

```
	:command -range=% SaveIt :<line1>,<line2>write! save_file
```

OTHER OPTIONS

Some of the other options and keywords are as follows:

```
	-count={number}		The command can take a count whose default is
				{number}.  The resulting count can be used
				through the <count> keyword.
	-bang			You can use a !.  If present, using <bang> will
				result in a !.
	-register		You can specify a register.  (The default is
				the unnamed register.)
				The register specification is available as
				<reg> (a.k.a. <register>).
	-complete={type}	Type of command-line completion used.  See
				|:command-completion| for the list of possible
				values.
	-bar			The command can be followed by | and another
				command, or " and a comment.
	-buffer			The command is only available for the current
				buffer.
```

Finally, you have the <lt> keyword. It stands for the character <. Use this to escape the special meaning of the <> items mentioned.

REDEFINING AND DELETING

To redefine the same command use the ! argument: >

```
	:command -nargs=+ Say :echo "<args>"
	:command! -nargs=+ Say :echo <q-args>
```

To delete a user command use ":delcommand". It takes a single argument, which is the name of the command. Example: >

```
	:delcommand SaveIt
```

To delete all the user commands: >

```
	:comclear
```

Careful, this can't be undone!

More details about all this in the reference manual: |user-commands|.

==

40.3 Autocommands

An autocommand is a command that is executed automatically in response to some event, such as a file being read or written or a buffer change. Through the use of autocommands you can train Vim to edit compressed files, for example. That is used in the |gzip| plugin.

Autocommands are very powerful. Use them with care and they will help you avoid typing many commands. Use them carelessly and they will cause a lot of trouble.

Suppose you want to replace a datestamp on the end of a file every time it is written. First you define a function: >

```
	:function DateInsert()
	:  $delete
	:  read !date
	:endfunction
```

You want this function to be called each time, just before a buffer is written to a file. This will make that happen: >

```
	:autocmd BufWritePre *  call DateInsert()
```

"BufWritePre" is the event for which this autocommand is triggered: Just before (pre) writing a buffer to a file. The "*" is a pattern to match with the file name. In this case it matches all files.

With this command enabled, when you do a ":write", Vim checks for any matching BufWritePre autocommands and executes them, and then it performs the ":write".

The general form of the :autocmd command is as follows: >

```
	:autocmd [group] {events} {file_pattern} [nested] {command}
```

The [group] name is optional. It is used in managing and calling the commands (more on this later). The {events} parameter is a list of events (comma separated) that trigger the command.

{file_pattern} is a filename, usually with wildcards. For example, using "*.txt" makes the autocommand be used for all files whose name end in ".txt". The optional [nested] flag allows for nesting of autocommands (see below), and finally, {command} is the command to be executed.

EVENTS

One of the most useful events is BufReadPost. It is triggered after a new file is being edited. It is commonly used to set option values. For example, you know that "*.gsm" files are GNU assembly language. To get the syntax file right, define this autocommand: >

```
	:autocmd BufReadPost *.gsm  set filetype=asm
```

If Vim is able to detect the type of file, it will set the 'filetype' option for you. This triggers the Filetype event. Use this to do something when a certain type of file is edited. For example, to load a list of abbreviations for text files: >

```
	:autocmd Filetype text  source ~/.vim/abbrevs.vim
```

When starting to edit a new file, you could make Vim insert a skeleton: >

```
	:autocmd BufNewFile *.[ch]  0read ~/skeletons/skel.c
```

See |autocmd-events| for a complete list of events.

PATTERNS

The {file_pattern} argument can actually be a comma-separated list of file patterns. For example: "*.c,*.h" matches files ending in ".c" and ".h".

The usual file wildcards can be used. Here is a summary of the most often used ones:

*	Match any character any number of times
?	Match any character once
[abc]	Match the character a, b or c
.	Matches a dot
a{b,c}	Matches "ab" and "ac"

When the pattern includes a slash (/) Vim will compare directory names. Without the slash only the last part of a file name is used. For example, "*.txt" matches "/home/biep/readme.txt". The pattern "/home/biep/*" would also match it. But "home/foo/*.txt" wouldn't.

When including a slash, Vim matches the pattern against both the full path of the file ("/home/biep/readme.txt") and the relative path (e.g., "biep/readme.txt").

> Note:
> When working on a system that uses a backslash as file separator, such as MS-Windows, you still use forward slashes in autocommands. This makes it easier to write the pattern, since a backslash has a special meaning. It also makes the autocommands portable.

DELETING

To delete an autocommand, use the same command as what it was defined with, but leave out the {command} at the end and use a !. Example: >

```
        :autocmd! FileWritePre *
```

This will delete all autocommands for the "FileWritePre" event that use the "*" pattern.

LISTING

To list all the currently defined autocommands, use this: >

```
        :autocmd
```

The list can be very long, especially when filetype detection is used. To list only part of the commands, specify the group, event and/or pattern. For example, to list all BufNewFile autocommands: >

```
        :autocmd BufNewFile
```

To list all autocommands for the pattern "*.c": >

```
        :autocmd * *.c
```

Using "*" for the event will list all the events. To list all autocommands for the cprograms group: >

```
        :autocmd cprograms
```

GROUPS

The {group} item, used when defining an autocommand, groups related autocommands together. This can be used to delete all the autocommands in a certain group, for example.

When defining several autocommands for a certain group, use the ":augroup" command. For example, let's define autocommands for C programs: >

```
        :augroup cprograms
        :  autocmd BufReadPost *.c,*.h :set sw=4 sts=4
        :  autocmd BufReadPost *.cpp   :set sw=3 sts=3
        :augroup END
```

This will do the same as: >

```
        :autocmd cprograms BufReadPost *.c,*.h :set sw=4 sts=4
        :autocmd cprograms BufReadPost *.cpp   :set sw=3 sts=3
```

To delete all autocommands in the "cprograms" group: >

```
        :autocmd! cprograms
```

NESTING

Generally, commands executed as the result of an autocommand event will not trigger any new events. If you read a file in response to a FileChangedShell event, it will not trigger the autocommands that would set the syntax, for example. To make the events triggered, add the "nested" argument: >

```
        :autocmd FileChangedShell * nested  edit
```

EXECUTING AUTOCOMMANDS

It is possible to trigger an autocommand by pretending an event has occurred. This is useful to have one autocommand trigger another one. Example: >

```
        :autocmd BufReadPost *.new  execute "doautocmd BufReadPost " .
expand("<afile>:r")
```

This defines an autocommand that is triggered when a new file has been edited. The file name must end in ".new". The ":execute" command uses expression evaluation to form a new command and execute it. When editing the file "tryout.c.new" the executed command will be: >

```
        :doautocmd BufReadPost tryout.c
```

The expand() function takes the "<afile>" argument, which stands for the file name the autocommand was executed for, and takes the root of the file name with ":r".

":doautocmd" executes on the current buffer. The ":doautoall" command works like "doautocmd" except it executes on all the buffers.

USING NORMAL MODE COMMANDS

The commands executed by an autocommand are Command-line commands. If you want to use a Normal mode command, the ":normal" command can be used. Example: >

```
        :autocmd BufReadPost *.log normal G
```

This will make the cursor jump to the last line of *.log files when you start to edit it.

Using the ":normal" command is a bit tricky. First of all, make sure its argument is a complete command, including all the arguments. When you use "i" to go to Insert mode, there must also be a <Esc> to leave Insert mode again. If you use a "/" to start a search pattern, there must be a <CR> to execute it.

The ":normal" command uses all the text after it as commands. Thus there can be no | and another command following. To work around this, put the ":normal" command inside an ":execute" command. This also makes it possible to pass unprintable characters in a convenient way. Example: >

```
        :autocmd BufReadPost *.chg execute "normal ONew entry:\<Esc>" |
                \ 1read !date
```

This also shows the use of a backslash to break a long command into more lines. This can be used in Vim scripts (not at the command line).

When you want the autocommand do something complicated, which involves jumping around in the file and then returning to the original position, you may want to restore the view on the file. See |restore-position| for an example.

IGNORING EVENTS

At times, you will not want to trigger an autocommand. The 'eventignore' option contains a list of events that will be totally ignored. For example, the following causes events for entering and leaving a window to be ignored: >

```
        :set eventignore=WinEnter,WinLeave
```

To ignore all events, use the following command: >

```
        :set eventignore=all
```

To set it back to the normal behavior, make 'eventignore' empty: >

```
        :set eventignore=
```

==

Next chapter: |usr_41.txt| Write a Vim script

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_41.txt   For Vim version 8.0. Last change: 2017 Aug 22

VIM USER MANUAL - by Bram Moolenaar

Write a Vim script

The Vim script language is used for the startup vimrc file, syntax files, and many other things. This chapter explains the items that can be used in a Vim script. There are a lot of them, thus this is a long chapter.

```
     Next chapter: |usr_42.txt|  Add new menus
 Previous chapter: |usr_40.txt|  Make new commands
Table of contents: |usr_toc.txt|
```

==

41.1 Introduction   *vim-script-intro* *script*

Your first experience with Vim scripts is the vimrc file. Vim reads it when it starts up and executes the commands. You can set options to values you prefer. And you can use any colon command in it (commands that start with a ":"; these are sometimes referred to as Ex commands or command-line commands).
   Syntax files are also Vim scripts. As are files that set options for a specific file type. A complicated macro can be defined by a separate Vim script file. You can think of other uses yourself.

Let's start with a simple example: >

```
	:let i = 1
	:while i < 5
	:  echo "count is" i
	:  let i += 1
	:endwhile
<
```

Note:
The ":" characters are not really needed here. You only need to use them when you type a command. In a Vim script file they can be left out. We will use them here anyway to make clear these are colon commands and make them stand out from Normal mode commands.
Note:
You can try out the examples by yanking the lines from the text here and executing them with :@"

The output of the example code is:

```
	count is 1 ~
	count is 2 ~
	count is 3 ~
	count is 4 ~
```

In the first line the ":let" command assigns a value to a variable. The generic form is: >

```
	:let {variable} = {expression}
```

In this case the variable name is "i" and the expression is a simple value, the number one.
   The ":while" command starts a loop. The generic form is: >

```
	:while {condition}
```

```
        :  {statements}
        :endwhile
```

The statements until the matching ":endwhile" are executed for as long as the condition is true. The condition used here is the expression "i < 5". This is true when the variable i is smaller than five.

Note:
If you happen to write a while loop that keeps on running, you can interrupt it by pressing CTRL-C (CTRL-Break on MS-Windows).

The ":echo" command prints its arguments. In this case the string "count is" and the value of the variable i. Since i is one, this will print:

```
        count is 1 ~
```

Then there is the ":let i += 1" command. This does the same thing as ":let i = i + 1". This adds one to the variable i and assigns the new value to the same variable.

The example was given to explain the commands, but would you really want to make such a loop, it can be written much more compact: >

```
        :for i in range(1, 4)
        :  echo "count is" i
        :endfor
```

We won't explain how |:for| and |range()| work until later. Follow the links if you are impatient.

THREE KINDS OF NUMBERS

Numbers can be decimal, hexadecimal or octal. A hexadecimal number starts with "0x" or "0X". For example "0x1f" is decimal 31. An octal number starts with a zero. "017" is decimal 15. Careful: don't put a zero before a decimal number, it will be interpreted as an octal number!

The ":echo" command always prints decimal numbers. Example: >

```
        :echo 0x7f 036
<       127 30 ~
```

A number is made negative with a minus sign. This also works for hexadecimal and octal numbers. A minus sign is also used for subtraction. Compare this with the previous example: >

```
        :echo 0x7f -036
<       97 ~
```

White space in an expression is ignored. However, it's recommended to use it for separating items, to make the expression easier to read. For example, to avoid the confusion with a negative number above, put a space between the minus sign and the following number: >

```
        :echo 0x7f - 036
```

==

41.2 Variables

A variable name consists of ASCII letters, digits and the underscore. It cannot start with a digit. Valid variable names are:

```
        counter
        _aap3
        very_long_variable_name_with_underscores
        FuncLength
        LENGTH
```

Invalid names are "foo+bar" and "6var".

These variables are global. To see a list of currently defined variables use this command: >

```
        :let
```

You can use global variables everywhere. This also means that when the variable "count" is used in one script file, it might also be used in another file. This leads to confusion at least, and real problems at worst. To avoid this, you can use a variable local to a script file by prepending "s:". For example, one script contains this code: >

```
        :let s:count = 1
        :while s:count < 5
        :  source other.vim
        :  let s:count += 1
        :endwhile
```

Since "s:count" is local to this script, you can be sure that sourcing the "other.vim" script will not change this variable. If "other.vim" also uses an "s:count" variable, it will be a different copy, local to that script. More about script-local variables here: |script-variable|.

There are more kinds of variables, see |internal-variables|. The most often used ones are:

b:name	variable local to a buffer
w:name	variable local to a window
g:name	global variable (also in a function)
v:name	variable predefined by Vim

DELETING VARIABLES

Variables take up memory and show up in the output of the ":let" command. To delete a variable use the ":unlet" command. Example: >

```
        :unlet s:count
```

This deletes the script-local variable "s:count" to free up the memory it uses. If you are not sure if the variable exists, and don't want an error message when it doesn't, append !: >

```
        :unlet! s:count
```

When a script finishes, the local variables used there will not be automatically freed. The next time the script executes, it can still use the old value. Example: >

```
        :if !exists("s:call_count")
        :  let s:call_count = 0
        :endif
        :let s:call_count = s:call_count + 1
        :echo "called" s:call_count "times"
```

The "exists()" function checks if a variable has already been defined. Its argument is the name of the variable you want to check. Not the variable itself! If you would do this: >

```
	:if !exists(s:call_count)
```

Then the value of s:call_count will be used as the name of the variable that exists() checks. That's not what you want.

The exclamation mark ! negates a value. When the value was true, it becomes false. When it was false, it becomes true. You can read it as "not". Thus "if !exists()" can be read as "if not exists()".

What Vim calls true is anything that is not zero. Zero is false.

```
	Note:
	Vim automatically converts a string to a number when it is looking for
	a number.  When using a string that doesn't start with a digit the
	resulting number is zero.  Thus look out for this: >
		:if "true"
<	The "true" will be interpreted as a zero, thus as false!
```

STRING VARIABLES AND CONSTANTS

So far only numbers were used for the variable value. Strings can be used as well. Numbers and strings are the basic types of variables that Vim supports. The type is dynamic, it is set each time when assigning a value to the variable with ":let". More about types in |41.8|.

To assign a string value to a variable, you need to use a string constant. There are two types of these. First the string in double quotes: >

```
	:let name = "peter"
	:echo name
<	peter ~
```

If you want to include a double quote inside the string, put a backslash in front of it: >

```
	:let name = "\"peter\""
	:echo name
<	"peter" ~
```

To avoid the need for a backslash, you can use a string in single quotes: >

```
	:let name = '"peter"'
	:echo name
<	"peter" ~
```

Inside a single-quote string all the characters are as they are. Only the single quote itself is special: you need to use two to get one. A backslash is taken literally, thus you can't use it to change the meaning of the character after it.

In double-quote strings it is possible to use special characters. Here are a few useful ones:

```
	\t		<Tab>
	\n		<NL>, line break
	\r		<CR>, <Enter>
	\e		<Esc>
	\b		<BS>, backspace
	\"		"
	\\		\, backslash
	\<Esc>		<Esc>
	\<C-W>		CTRL-W
```

The last two are just examples. The "\<name>" form can be used to include

the special key "name".
See |expr-quote| for the full list of special items in a string.

==

41.3 Expressions

Vim has a rich, yet simple way to handle expressions. You can read the definition here: |expression-syntax|. Here we will show the most common items.

The numbers, strings and variables mentioned above are expressions by themselves. Thus everywhere an expression is expected, you can use a number, string or variable. Other basic items in an expression are:

$NAME	environment variable
&name	option
@r	register

Examples: >

```
	:echo "The value of 'tabstop' is" &ts
	:echo "Your home directory is" $HOME
	:if @a > 5
```

The &name form can be used to save an option value, set it to a new value, do something and restore the old value. Example: >

```
	:let save_ic = &ic
	:set noic
	:/The Start/,$delete
	:let &ic = save_ic
```

This makes sure the "The Start" pattern is used with the 'ignorecase' option off. Still, it keeps the value that the user had set. (Another way to do this would be to add "\C" to the pattern, see |/\C|.)

MATHEMATICS

It becomes more interesting if we combine these basic items. Let's start with mathematics on numbers:

a + b	add
a - b	subtract
a * b	multiply
a / b	divide
a % b	modulo

The usual precedence is used. Example: >

```
	:echo 10 + 5 * 2
<	20 ~
```

Grouping is done with parentheses. No surprises here. Example: >

```
	:echo (10 + 5) * 2
<	30 ~
```

Strings can be concatenated with ".". Example: >

```
	:echo "foo" . "bar"
<	foobar ~
```

When the ":echo" command gets multiple arguments, it separates them with a space. In the example the argument is a single expression, thus no space is inserted.

Borrowed from the C language is the conditional expression:

```
        a ? b : c
```

If "a" evaluates to true "b" is used, otherwise "c" is used. Example: >

```
        :let i = 4
        :echo i > 5 ? "i is big" : "i is small"
<       i is small ~
```

The three parts of the constructs are always evaluated first, thus you could see it work as:

```
        (a) ? (b) : (c)
```

==

41.4 Conditionals

The ":if" commands executes the following statements, until the matching ":endif", only when a condition is met. The generic form is:

```
        :if {condition}
           {statements}
        :endif
```

Only when the expression {condition} evaluates to true (non-zero) will the {statements} be executed. These must still be valid commands. If they contain garbage, Vim won't be able to find the ":endif".

You can also use ":else". The generic form for this is:

```
        :if {condition}
           {statements}
        :else
           {statements}
        :endif
```

The second {statements} is only executed if the first one isn't.

Finally, there is ":elseif":

```
        :if {condition}
           {statements}
        :elseif {condition}
           {statements}
        :endif
```

This works just like using ":else" and then "if", but without the need for an extra ":endif".

A useful example for your vimrc file is checking the 'term' option and doing something depending upon its value: >

```
        :if &term == "xterm"
        :  " Do stuff for xterm
        :elseif &term == "vt100"
        :  " Do stuff for a vt100 terminal
        :else
        :  " Do something for other terminals
        :endif
```

LOGIC OPERATIONS

We already used some of them in the examples. These are the most often used ones:

a == b	equal to
a != b	not equal to
a > b	greater than
a >= b	greater than or equal to
a < b	less than
a <= b	less than or equal to

The result is one if the condition is met and zero otherwise. An example: >

```
	:if v:version >= 700
	:  echo "congratulations"
	:else
	:  echo "you are using an old version, upgrade!"
	:endif
```

Here "v:version" is a variable defined by Vim, which has the value of the Vim version. 600 is for version 6.0. Version 6.1 has the value 601. This is very useful to write a script that works with multiple versions of Vim. |v:version|

The logic operators work both for numbers and strings. When comparing two strings, the mathematical difference is used. This compares byte values, which may not be right for some languages.

When comparing a string with a number, the string is first converted to a number. This is a bit tricky, because when a string doesn't look like a number, the number zero is used. Example: >

```
	:if 0 == "one"
	:  echo "yes"
	:endif
```

This will echo "yes", because "one" doesn't look like a number, thus it is converted to the number zero.

For strings there are two more items:

a =~ b	matches with
a !~ b	does not match with

The left item "a" is used as a string. The right item "b" is used as a pattern, like what's used for searching. Example: >

```
	:if str =~ " "
	:  echo "str contains a space"
	:endif
	:if str !~ '\.$'
	:  echo "str does not end in a full stop"
	:endif
```

Notice the use of a single-quote string for the pattern. This is useful, because backslashes would need to be doubled in a double-quote string and patterns tend to contain many backslashes.

The 'ignorecase' option is used when comparing strings. When you don't want that, append "#" to match case and "?" to ignore case. Thus "==?" compares two strings to be equal while ignoring case. And "!~#" checks if a pattern doesn't match, also checking the case of letters. For the full table see |expr-==|.

MORE LOOPING

The ":while" command was already mentioned. Two more statements can be used in between the ":while" and the ":endwhile":

```
        :continue               Jump back to the start of the while loop; the
                                loop continues.
        :break                  Jump forward to the ":endwhile"; the loop is
                                discontinued.
```

Example: >

```
        :while counter < 40
        :  call do_something()
        :  if skip_flag
        :    continue
        :  endif
        :  if finished_flag
        :    break
        :  endif
        :  sleep 50m
        :endwhile
```

The ":sleep" command makes Vim take a nap. The "50m" specifies fifty milliseconds. Another example is ":sleep 4", which sleeps for four seconds.

Even more looping can be done with the ":for" command, see below in |41.8|.

==

41.5 Executing an expression

So far the commands in the script were executed by Vim directly. The ":execute" command allows executing the result of an expression. This is a very powerful way to build commands and execute them.

An example is to jump to a tag, which is contained in a variable: >

```
        :execute "tag " . tag_name
```

The "." is used to concatenate the string "tag " with the value of variable "tag_name". Suppose "tag_name" has the value "get_cmd", then the command that will be executed is: >

```
        :tag get_cmd
```

The ":execute" command can only execute colon commands. The ":normal" command executes Normal mode commands. However, its argument is not an expression but the literal command characters. Example: >

```
        :normal gg=G
```

This jumps to the first line and formats all lines with the "=" operator.

To make ":normal" work with an expression, combine ":execute" with it. Example: >

```
        :execute "normal " . normal_commands
```

The variable "normal_commands" must contain the Normal mode commands.

Make sure that the argument for ":normal" is a complete command. Otherwise

Vim will run into the end of the argument and abort the command. For example, if you start Insert mode, you must leave Insert mode as well. This works: >

```
	:execute "normal Inew text \<Esc>"
```

This inserts "new text " in the current line. Notice the use of the special key "\<Esc>". This avoids having to enter a real <Esc> character in your script.

If you don't want to execute a string but evaluate it to get its expression value, you can use the eval() function: >

```
	:let optname = "path"
	:let optval = eval('&' . optname)
```

A "&" character is prepended to "path", thus the argument to eval() is "&path". The result will then be the value of the 'path' option.
   The same thing can be done with: >

```
	:exe 'let optval = &' . optname
```

==

41.6 Using functions

Vim defines many functions and provides a large amount of functionality that way. A few examples will be given in this section. You can find the whole list here: |functions|.

A function is called with the ":call" command. The parameters are passed in between parentheses separated by commas. Example: >

```
	:call search("Date: ", "W")
```

This calls the search() function, with arguments "Date: " and "W". The search() function uses its first argument as a search pattern and the second one as flags. The "W" flag means the search doesn't wrap around the end of the file.

A function can be called in an expression. Example: >

```
	:let line = getline(".")
	:let repl = substitute(line, '\a', "*", "g")
	:call setline(".", repl)
```

The getline() function obtains a line from the current buffer. Its argument is a specification of the line number. In this case "." is used, which means the line where the cursor is.
   The substitute() function does something similar to the ":substitute" command. The first argument is the string on which to perform the substitution. The second argument is the pattern, the third the replacement string. Finally, the last arguments are the flags.
   The setline() function sets the line, specified by the first argument, to a new string, the second argument. In this example the line under the cursor is replaced with the result of the substitute(). Thus the effect of the three statements is equal to: >

```
	:substitute/\a/*/g
```

Using the functions becomes more interesting when you do more work before and after the substitute() call.

FUNCTIONS						*function-list*

There are many functions. We will mention them here, grouped by what they are used for. You can find an alphabetical list here: |functions|. Use CTRL-] on the function name to jump to detailed help on it.

String manipulation: *string-functions*

nr2char()	get a character by its ASCII value
char2nr()	get ASCII value of a character
str2nr()	convert a string to a Number
str2float()	convert a string to a Float
printf()	format a string according to % items
escape()	escape characters in a string with a '\'
shellescape()	escape a string for use with a shell command
fnameescape()	escape a file name for use with a Vim command
tr()	translate characters from one set to another
strtrans()	translate a string to make it printable
tolower()	turn a string to lowercase
toupper()	turn a string to uppercase
match()	position where a pattern matches in a string
matchend()	position where a pattern match ends in a string
matchstr()	match of a pattern in a string
matchstrpos()	match and positions of a pattern in a string
matchlist()	like matchstr() and also return submatches
stridx()	first index of a short string in a long string
strridx()	last index of a short string in a long string
strlen()	length of a string in bytes
strchars()	length of a string in characters
strwidth()	size of string when displayed
strdisplaywidth()	size of string when displayed, deals with tabs
substitute()	substitute a pattern match with a string
submatch()	get a specific match in ":s" and substitute()
strpart()	get part of a string using byte index
strcharpart()	get part of a string using char index
strgetchar()	get character from a string using char index
expand()	expand special keywords
iconv()	convert text from one encoding to another
byteidx()	byte index of a character in a string
byteidxcomp()	like byteidx() but count composing characters
repeat()	repeat a string multiple times
eval()	evaluate a string expression
execute()	execute an Ex command and get the output

List manipulation: *list-functions*

get()	get an item without error for wrong index
len()	number of items in a List
empty()	check if List is empty
insert()	insert an item somewhere in a List
add()	append an item to a List
extend()	append a List to a List
remove()	remove one or more items from a List
copy()	make a shallow copy of a List
deepcopy()	make a full copy of a List
filter()	remove selected items from a List
map()	change each List item
sort()	sort a List
reverse()	reverse the order of a List
uniq()	remove copies of repeated adjacent items
split()	split a String into a List
join()	join List items into a String
range()	return a List with a sequence of numbers
string()	String representation of a List
call()	call a function with List as arguments

```
        index()                 index of a value in a List
        max()                   maximum value in a List
        min()                   minimum value in a List
        count()                 count number of times a value appears in a List
        repeat()                repeat a List multiple times

Dictionary manipulation:                                *dict-functions*
        get()                   get an entry without an error for a wrong key
        len()                   number of entries in a Dictionary
        has_key()               check whether a key appears in a Dictionary
        empty()                 check if Dictionary is empty
        remove()                remove an entry from a Dictionary
        extend()                add entries from one Dictionary to another
        filter()                remove selected entries from a Dictionary
        map()                   change each Dictionary entry
        keys()                  get List of Dictionary keys
        values()                get List of Dictionary values
        items()                 get List of Dictionary key-value pairs
        copy()                  make a shallow copy of a Dictionary
        deepcopy()              make a full copy of a Dictionary
        string()                String representation of a Dictionary
        max()                   maximum value in a Dictionary
        min()                   minimum value in a Dictionary
        count()                 count number of times a value appears

Floating point computation:                             *float-functions*
        float2nr()              convert Float to Number
        abs()                   absolute value (also works for Number)
        round()                 round off
        ceil()                  round up
        floor()                 round down
        trunc()                 remove value after decimal point
        fmod()                  remainder of division
        exp()                   exponential
        log()                   natural logarithm (logarithm to base e)
        log10()                 logarithm to base 10
        pow()                   value of x to the exponent y
        sqrt()                  square root
        sin()                   sine
        cos()                   cosine
        tan()                   tangent
        asin()                  arc sine
        acos()                  arc cosine
        atan()                  arc tangent
        atan2()                 arc tangent
        sinh()                  hyperbolic sine
        cosh()                  hyperbolic cosine
        tanh()                  hyperbolic tangent
        isnan()                 check for not a number

Other computation:                                      *bitwise-function*
        and()                   bitwise AND
        invert()                bitwise invert
        or()                    bitwise OR
        xor()                   bitwise XOR
        sha256()                SHA-256 hash

Variables:                                              *var-functions*
        type()                  type of a variable
        islocked()              check if a variable is locked
        funcref()               get a Funcref for a function reference
        function()              get a Funcref for a function name
        getbufvar()             get a variable value from a specific buffer
        setbufvar()             set a variable in a specific buffer
        getwinvar()             get a variable from specific window
        gettabvar()             get a variable from specific tab page
        gettabwinvar()          get a variable from specific window & tab page
        setwinvar()             set a variable in a specific window
        settabvar()             set a variable in a specific tab page
        settabwinvar()          set a variable in a specific window & tab page
        garbagecollect()        possibly free memory

Cursor and mark position:               *cursor-functions* *mark-functions*
        col()                   column number of the cursor or a mark
        virtcol()               screen column of the cursor or a mark
        line()                  line number of the cursor or mark
        wincol()                window column number of the cursor
        winline()               window line number of the cursor
        cursor()                position the cursor at a line/column
        screencol()             get screen column of the cursor
        screenrow()             get screen row of the cursor
        getcurpos()             get position of the cursor
        getpos()                get position of cursor, mark, etc.
        setpos()                set position of cursor, mark, etc.
        byte2line()             get line number at a specific byte count
        line2byte()             byte count at a specific line
        diff_filler()           get the number of filler lines above a line
        screenattr()            get attribute at a screen line/row
        screenchar()            get character code at a screen line/row

Working with text in the current buffer:                *text-functions*
        getline()               get a line or list of lines from the buffer
        setline()               replace a line in the buffer
        append()                append line or list of lines in the buffer
        indent()                indent of a specific line
        cindent()               indent according to C indenting
        lispindent()            indent according to Lisp indenting
        nextnonblank()          find next non-blank line
        prevnonblank()          find previous non-blank line
        search()                find a match for a pattern
        searchpos()             find a match for a pattern
        searchpair()            find the other end of a start/skip/end
        searchpairpos()         find the other end of a start/skip/end
        searchdecl()            search for the declaration of a name
        getcharsearch()         return character search information
        setcharsearch()         set character search information

                                        *system-functions* *file-functions*
System functions and manipulation of files:
        glob()                  expand wildcards
        globpath()              expand wildcards in a number of directories
        glob2regpat()           convert a glob pattern into a search pattern
        findfile()              find a file in a list of directories
        finddir()               find a directory in a list of directories
        resolve()               find out where a shortcut points to
        fnamemodify()           modify a file name
        pathshorten()           shorten directory names in a path
        simplify()              simplify a path without changing its meaning
        executable()            check if an executable program exists
        exepath()               full path of an executable program
        filereadable()          check if a file can be read
        filewritable()          check if a file can be written to
        getfperm()              get the permissions of a file
        setfperm()              set the permissions of a file
        getftype()              get the kind of a file
        isdirectory()           check if a directory exists
        getfsize()              get the size of a file
        getcwd()                get the current working directory
        haslocaldir()           check if current window used |:lcd|
        tempname()              get the name of a temporary file
        mkdir()                 create a new directory
        delete()                delete a file
        rename()                rename a file
        system()                get the result of a shell command as a string
        systemlist()            get the result of a shell command as a list
        hostname()              name of the system
        readfile()              read a file into a List of lines
        writefile()             write a List of lines into a file

Date and Time:                          *date-functions* *time-functions*
        getftime()              get last modification time of a file
        localtime()             get current time in seconds
        strftime()              convert time to a string
        reltime()               get the current or elapsed time accurately
        reltimestr()            convert reltime() result to a string
        reltimefloat()          convert reltime() result to a Float

                        *buffer-functions* *window-functions* *arg-functions*
Buffers, windows and the argument list:
        argc()                  number of entries in the argument list
        argidx()                current position in the argument list
        arglistid()             get id of the argument list
        argv()                  get one entry from the argument list
        bufexists()             check if a buffer exists
        buflisted()             check if a buffer exists and is listed
        bufloaded()             check if a buffer exists and is loaded
        bufname()               get the name of a specific buffer
        bufnr()                 get the buffer number of a specific buffer
        tabpagebuflist()        return List of buffers in a tab page
        tabpagenr()             get the number of a tab page
        tabpagewinnr()          like winnr() for a specified tab page
        winnr()                 get the window number for the current window
        bufwinid()              get the window ID of a specific buffer
        bufwinnr()              get the window number of a specific buffer
        winbufnr()              get the buffer number of a specific window
        getbufline()            get a list of lines from the specified buffer
        win_findbuf()           find windows containing a buffer
        win_getid()             get window ID of a window
        win_gotoid()            go to window with ID
        win_id2tabwin()         get tab and window nr from window ID
        win_id2win()            get window nr from window ID
        getbufinfo()            get a list with buffer information
        gettabinfo()            get a list with tab page information
        getwininfo()            get a list with window information

Command line:                                   *command-line-functions*
        getcmdline()            get the current command line
        getcmdpos()             get position of the cursor in the command line
        setcmdpos()             set position of the cursor in the command line
        getcmdtype()            return the current command-line type
        getcmdwintype()         return the current command-line window type
        getcompletion()         list of command-line completion matches

Quickfix and location lists:                    *quickfix-functions*
        getqflist()             list of quickfix errors
        setqflist()             modify a quickfix list
        getloclist()            list of location list items
        setloclist()            modify a location list

Insert mode completion:                         *completion-functions*
        complete()              set found matches
        complete_add()          add to found matches
        complete_check()        check if completion should be aborted
        pumvisible()            check if the popup menu is displayed

Folding:                                        *folding-functions*
        foldclosed()            check for a closed fold at a specific line
        foldclosedend()         like foldclosed() but return the last line
        foldlevel()             check for the fold level at a specific line
        foldtext()              generate the line displayed for a closed fold
        foldtextresult()        get the text displayed for a closed fold

Syntax and highlighting:          *syntax-functions* *highlighting-functions*
        clearmatches()          clear all matches defined by |matchadd()| and
                                the |:match| commands
        getmatches()            get all matches defined by |matchadd()| and
                                the |:match| commands
        hlexists()              check if a highlight group exists
        hlID()                  get ID of a highlight group
        synID()                 get syntax ID at a specific position
        synIDattr()             get a specific attribute of a syntax ID
        synIDtrans()            get translated syntax ID
        synstack()              get list of syntax IDs at a specific position
        synconcealed()          get info about concealing
        diff_hlID()             get highlight ID for diff mode at a position
        matchadd()              define a pattern to highlight (a "match")
        matchaddpos()           define a list of positions to highlight
        matcharg()              get info about |:match| arguments
        matchdelete()           delete a match defined by |matchadd()| or a
                                |:match| command
        setmatches()            restore a list of matches saved by
                                |getmatches()|

Spelling:                                       *spell-functions*
        spellbadword()          locate badly spelled word at or after cursor
        spellsuggest()          return suggested spelling corrections
        soundfold()             return the sound-a-like equivalent of a word

History:                                        *history-functions*
        histadd()               add an item to a history
        histdel()               delete an item from a history
        histget()               get an item from a history
        histnr()                get highest index of a history list

Interactive:                                    *interactive-functions*
        browse()                put up a file requester
        browsedir()             put up a directory requester
        confirm()               let the user make a choice
        getchar()               get a character from the user
        getcharmod()            get modifiers for the last typed character
        feedkeys()              put characters in the typeahead queue
        input()                 get a line from the user
        inputlist()             let the user pick an entry from a list
        inputsecret()           get a line from the user without showing it
        inputdialog()           get a line from the user in a dialog
        inputsave()             save and clear typeahead
        inputrestore()          restore typeahead
```

```
GUI:                                            *gui-functions*
        getfontname()           get name of current font being used
        getwinposx()            X position of the GUI Vim window
        getwinposy()            Y position of the GUI Vim window
        balloon_show()          set the balloon content

Vim server:                                     *server-functions*
        serverlist()            return the list of server names
        remote_startserve()     run a server
        remote_send()           send command characters to a Vim server
        remote_expr()           evaluate an expression in a Vim server
        server2client()         send a reply to a client of a Vim server
        remote_peek()           check if there is a reply from a Vim server
        remote_read()           read a reply from a Vim server
        foreground()            move the Vim window to the foreground
        remote_foreground()     move the Vim server window to the foreground

Window size and position:                       *window-size-functions*
        winheight()             get height of a specific window
        winwidth()              get width of a specific window
        winrestcmd()            return command to restore window sizes
        winsaveview()           get view of current window
        winrestview()           restore saved view of current window

Mappings:                                   *mapping-functions*
        hasmapto()              check if a mapping exists
        mapcheck()              check if a matching mapping exists
        maparg()                get rhs of a mapping
        wildmenumode()          check if the wildmode is active

Testing:                                    *test-functions*
        assert_equal()          assert that two expressions values are equal
        assert_notequal()       assert that two expressions values are not equal
        assert_inrange()        assert that an expression is inside a range
        assert_match()          assert that a pattern matches the value
        assert_notmatch()       assert that a pattern does not match the value
        assert_false()          assert that an expression is false
        assert_true()           assert that an expression is true
        assert_exception()      assert that a command throws an exception
        assert_fails()          assert that a function call fails
        assert_report()         report a test failure
        test_alloc_fail()       make memory allocation fail
        test_autochdir()        enable 'autochdir' during startup
        test_override()         test with Vim internal overrides
        test_garbagecollect_now()   free memory right now
        test_ignore_error()     ignore a specific error message
        test_null_channel()     return a null Channel
        test_null_dict()        return a null Dict
        test_null_job()         return a null Job
        test_null_list()        return a null List
        test_null_partial()     return a null Partial function
        test_null_string()      return a null String
        test_settime()          set the time Vim uses internally

Inter-process communication:                *channel-functions*
        ch_canread()            check if there is something to read
        ch_open()               open a channel
        ch_close()              close a channel
        ch_close_in()           close the in part of a channel
        ch_read()               read a message from a channel
        ch_readraw()            read a raw message from a channel
        ch_sendexpr()           send a JSON message over a channel
        ch_sendraw()            send a raw message over a channel
        ch_evalexpr()           evaluates an expression over channel
        ch_evalraw()            evaluates a raw string over channel
        ch_status()             get status of a channel
        ch_getbufnr()           get the buffer number of a channel
        ch_getjob()             get the job associated with a channel
        ch_info()               get channel information
        ch_log()                write a message in the channel log file
        ch_logfile()            set the channel log file
        ch_setoptions()         set the options for a channel
        json_encode()           encode an expression to a JSON string
        json_decode()           decode a JSON string to Vim types
        js_encode()             encode an expression to a JSON string
        js_decode()             decode a JSON string to Vim types

Jobs:                                           *job-functions*
        job_start()             start a job
        job_stop()              stop a job
        job_status()            get the status of a job
        job_getchannel()        get the channel used by a job
        job_info()              get information about a job
        job_setoptions()        set options for a job

Terminal window:                                *terminal-functions*
        term_start()            open a terminal window and run a job
        term_list()             get the list of terminal buffers
        term_sendkeys()         send keystrokes to a terminal
        term_wait()             wait for screen to be updated
        term_getjob()           get the job associated with a terminal
        term_scrape()           get row of a terminal screen
        term_getline()          get a line of text from a terminal
        term_getattr()          get the value of attribute {what}
        term_getcursor()        get the cursor position of a terminal
        term_getscrolled()      get the scroll count of a terminal
        term_getaltscreen()     get the alternate screen flag
        term_getsize()          get the size of a terminal
        term_getstatus()        get the status of a terminal
        term_gettitle()         get the title of a terminal
        term_gettty()           get the tty name of a terminal

Timers:                                         *timer-functions*
        timer_start()           create a timer
        timer_pause()           pause or unpause a timer
        timer_stop()            stop a timer
        timer_stopall()         stop all timers
        timer_info()            get information about timers

Various:                                        *various-functions*
        mode()                  get current editing mode
        visualmode()            last visual mode used
        exists()                check if a variable, function, etc. exists
        has()                   check if a feature is supported in Vim
        changenr()              return number of most recent change
        cscope_connection()     check if a cscope connection exists
        did_filetype()          check if a FileType autocommand was used
        eventhandler()          check if invoked by an event handler
        getpid()                get process ID of Vim

        libcall()               call a function in an external library
        libcallnr()             idem, returning a number

        undofile()              get the name of the undo file
        undotree()              return the state of the undo tree

        getreg()                get contents of a register
        getregtype()            get type of a register
        setreg()                set contents and type of a register

        shiftwidth()            effective value of 'shiftwidth'

        wordcount()             get byte/word/char count of buffer

        taglist()               get list of matching tags
        tagfiles()              get a list of tags files

        luaeval()               evaluate Lua expression
        mzeval()                evaluate |MzScheme| expression
        perleval()              evaluate Perl expression (|+perl|)
        py3eval()               evaluate Python expression (|+python3|)
        pyeval()                evaluate Python expression (|+python|)
        pyxeval()               evaluate |python_x| expression
```

==

41.7 Defining a function

Vim enables you to define your own functions. The basic function declaration
begins as follows: >

```
        :function {name}({var1}, {var2}, ...)
        :  {body}
        :endfunction
```

<

	Note:
	Function names must begin with a capital letter.

Let's define a short function to return the smaller of two numbers. It starts
with this line: >

```
        :function Min(num1, num2)
```

This tells Vim that the function is named "Min" and it takes two arguments:
"num1" and "num2".
   The first thing you need to do is to check to see which number is smaller:
   >

```
        :  if a:num1 < a:num2
```

The special prefix "a:" tells Vim that the variable is a function argument.
Let's assign the variable "smaller" the value of the smallest number: >

```
        :  if a:num1 < a:num2
        :    let smaller = a:num1
        :  else
        :    let smaller = a:num2
        :  endif
```

The variable "smaller" is a local variable. Variables used inside a function
are local unless prefixed by something like "g:", "a:", or "s:".

	Note:
	To access a global variable from inside a function you must prepend
	"g:" to it. Thus "g:today" inside a function is used for the global
	variable "today", and "today" is another variable, local to the
	function.

You now use the ":return" statement to return the smallest number to the user. Finally, you end the function: >

```
        :  return smaller
        :endfunction
```

The complete function definition is as follows: >

```
        :function Min(num1, num2)
        :  if a:num1 < a:num2
        :    let smaller = a:num1
        :  else
        :    let smaller = a:num2
        :  endif
        :  return smaller
        :endfunction
```

For people who like short functions, this does the same thing: >

```
        :function Min(num1, num2)
        :  if a:num1 < a:num2
        :    return a:num1
        :  endif
        :  return a:num2
        :endfunction
```

A user defined function is called in exactly the same way as a built-in function. Only the name is different. The Min function can be used like this: >

```
        :echo Min(5, 8)
```

Only now will the function be executed and the lines be interpreted by Vim. If there are mistakes, like using an undefined variable or function, you will now get an error message. When defining the function these errors are not detected.

When a function reaches ":endfunction" or ":return" is used without an argument, the function returns zero.

To redefine a function that already exists, use the ! for the ":function" command: >

```
        :function!  Min(num1, num2, num3)
```

USING A RANGE

The ":call" command can be given a line range. This can have one of two meanings. When a function has been defined with the "range" keyword, it will take care of the line range itself.
  The function will be passed the variables "a:firstline" and "a:lastline". These will have the line numbers from the range the function was called with. Example: >

```
        :function Count_words() range
        :  let lnum = a:firstline
        :  let n = 0
        :  while lnum <= a:lastline
        :    let n = n + len(split(getline(lnum)))
        :    let lnum = lnum + 1
        :  endwhile
```

```
	:  echo "found " . n . " words"
	:endfunction
```

You can call this function with: >

```
	:10,30call Count_words()
```

It will be executed once and echo the number of words.
   The other way to use a line range is by defining a function without the "range" keyword. The function will be called once for every line in the range, with the cursor in that line. Example: >

```
	:function  Number()
	:  echo "line " . line(".") . " contains: " . getline(".")
	:endfunction
```

If you call this function with: >

```
	:10,15call Number()
```

The function will be called six times.

VARIABLE NUMBER OF ARGUMENTS

Vim enables you to define functions that have a variable number of arguments. The following command, for instance, defines a function that must have 1 argument (start) and can have up to 20 additional arguments: >

```
	:function Show(start, ...)
```

The variable "a:1" contains the first optional argument, "a:2" the second, and so on. The variable "a:0" contains the number of extra arguments.
   For example: >

```
	:function Show(start, ...)
	:  echohl Title
	:  echo "start is " . a:start
	:  echohl None
	:  let index = 1
	:  while index <= a:0
	:    echo "  Arg " . index . " is " . a:{index}
	:    let index = index + 1
	:  endwhile
	:  echo ""
	:endfunction
```

This uses the ":echohl" command to specify the highlighting used for the following ":echo" command. ":echohl None" stops it again. The ":echon" command works like ":echo", but doesn't output a line break.

You can also use the a:000 variable, it is a List of all the "..." arguments. See |a:000|.

LISTING FUNCTIONS

The ":function" command lists the names and arguments of all user-defined functions: >

```
	:function
<	function Show(start, ...) ~
```

```
	function GetVimIndent() ~
	function SetSyn(name) ~
```

To see what a function does, use its name as an argument for ":function": >

```
	:function SetSyn
<	1     if &syntax == '' ~
	2       let &syntax = a:name ~
	3     endif ~
	   endfunction ~
```

DEBUGGING

The line number is useful for when you get an error message or when debugging. See |debug-scripts| about debugging mode.
   You can also set the 'verbose' option to 12 or higher to see all function calls. Set it to 15 or higher to see every executed line.

DELETING A FUNCTION

To delete the Show() function: >

```
	:delfunction Show
```

You get an error when the function doesn't exist.

FUNCTION REFERENCES

Sometimes it can be useful to have a variable point to one function or another. You can do it with the function() function. It turns the name of a function into a reference: >

```
	:let result = 0		" or 1
	:function! Right()
	:  return 'Right!'
	:endfunc
	:function! Wrong()
	:  return 'Wrong!'
	:endfunc
	:
	:if result == 1
	:  let Afunc = function('Right')
	:else
	:  let Afunc = function('Wrong')
	:endif
	:echo call(Afunc, [])
<	Wrong! ~
```

Note that the name of a variable that holds a function reference must start with a capital. Otherwise it could be confused with the name of a builtin function.
   The way to invoke a function that a variable refers to is with the call() function. Its first argument is the function reference, the second argument is a List with arguments.

Function references are most useful in combination with a Dictionary, as is explained in the next section.

==

41.8 Lists and Dictionaries

So far we have used the basic types String and Number. Vim also supports two composite types: List and Dictionary.

A List is an ordered sequence of things. The things can be any kind of value, thus you can make a List of numbers, a List of Lists and even a List of mixed items. To create a List with three strings: >

```
        :let alist = ['aap', 'mies', 'noot']
```

The List items are enclosed in square brackets and separated by commas. To create an empty List: >

```
        :let alist = []
```

You can add items to a List with the add() function: >

```
        :let alist = []
        :call add(alist, 'foo')
        :call add(alist, 'bar')
        :echo alist
<       ['foo', 'bar'] ~
```

List concatenation is done with +: >

```
        :echo alist + ['foo', 'bar']
<       ['foo', 'bar', 'foo', 'bar'] ~
```

Or, if you want to extend a List directly: >

```
        :let alist = ['one']
        :call extend(alist, ['two', 'three'])
        :echo alist
<       ['one', 'two', 'three'] ~
```

Notice that using add() will have a different effect: >

```
        :let alist = ['one']
        :call add(alist, ['two', 'three'])
        :echo alist
<       ['one', ['two', 'three']] ~
```

The second argument of add() is added as a single item.

FOR LOOP

One of the nice things you can do with a List is iterate over it: >

```
        :let alist = ['one', 'two', 'three']
        :for n in alist
        :  echo n
        :endfor
<       one ~
        two ~
        three ~
```

This will loop over each element in List "alist", assigning the value to variable "n". The generic form of a for loop is: >

```
        :for {varname} in {listexpression}
        :  {commands}
        :endfor
```

To loop a certain number of times you need a List of a specific length. The range() function creates one for you: >

```
        :for a in range(3)
        :  echo a
        :endfor
<       0 ~
        1 ~
        2 ~
```

Notice that the first item of the List that range() produces is zero, thus the last item is one less than the length of the list.
   You can also specify the maximum value, the stride and even go backwards: >

```
        :for a in range(8, 4, -2)
        :  echo a
        :endfor
<       8 ~
        6 ~
        4 ~
```

A more useful example, looping over lines in the buffer: >

```
        :for line in getline(1, 20)
        :  if line =~ "Date: "
        :    echo matchstr(line, 'Date: \zs.*')
        :  endif
        :endfor
```

This looks into lines 1 to 20 (inclusive) and echoes any date found in there.

DICTIONARIES

A Dictionary stores key-value pairs. You can quickly lookup a value if you know the key. A Dictionary is created with curly braces: >

```
        :let uk2nl = {'one': 'een', 'two': 'twee', 'three': 'drie'}
```

Now you can lookup words by putting the key in square brackets: >

```
        :echo uk2nl['two']
<       twee ~
```

The generic form for defining a Dictionary is: >

```
        {<key> : <value>, ...}
```

An empty Dictionary is one without any keys: >

```
        {}
```

The possibilities with Dictionaries are numerous. There are various functions for them as well. For example, you can obtain a list of the keys and loop over them: >

```
        :for key in keys(uk2nl)
        :  echo key
        :endfor
```

```
<       three ~
        one ~
        two ~
```

You will notice the keys are not ordered. You can sort the list to get a specific order: >

```
        :for key in sort(keys(uk2nl))
        :  echo key
        :endfor
<       one ~
        three ~
        two ~
```

But you can never get back the order in which items are defined. For that you need to use a List, it stores items in an ordered sequence.

DICTIONARY FUNCTIONS

The items in a Dictionary can normally be obtained with an index in square brackets: >

```
        :echo uk2nl['one']
<       een ~
```

A method that does the same, but without so many punctuation characters: >

```
        :echo uk2nl.one
<       een ~
```

This only works for a key that is made of ASCII letters, digits and the underscore. You can also assign a new value this way: >

```
        :let uk2nl.four = 'vier'
        :echo uk2nl
<       {'three': 'drie', 'four': 'vier', 'one': 'een', 'two': 'twee'} ~
```

And now for something special: you can directly define a function and store a reference to it in the dictionary: >

```
        :function uk2nl.translate(line) dict
        :  return join(map(split(a:line), 'get(self, v:val, "???")'))
        :endfunction
```

Let's first try it out: >

```
        :echo uk2nl.translate('three two five one')
<       drie twee ??? een ~
```

The first special thing you notice is the "dict" at the end of the ":function" line. This marks the function as being used from a Dictionary. The "self" local variable will then refer to that Dictionary.

Now let's break up the complicated return command: >

```
        split(a:line)
```

The split() function takes a string, chops it into whitespace separated words and returns a list with these words. Thus in the example it returns: >

```
        :echo split('three two five one')
<       ['three', 'two', 'five', 'one'] ~
```

This list is the first argument to the map() function. This will go through the list, evaluating its second argument with "v:val" set to the value of each item. This is a shortcut to using a for loop. This command: >

```
        :let alist = map(split(a:line), 'get(self, v:val, "???")')
```

Is equivalent to: >

```
        :let alist = split(a:line)
        :for idx in range(len(alist))
        :  let alist[idx] = get(self, alist[idx], "???")
        :endfor
```

The get() function checks if a key is present in a Dictionary. If it is, then the value is retrieved. If it isn't, then the default value is returned, in the example it's '???'. This is a convenient way to handle situations where a key may not be present and you don't want an error message.

The join() function does the opposite of split(): it joins together a list of words, putting a space in between.

This combination of split(), map() and join() is a nice way to filter a line of words in a very compact way.

OBJECT ORIENTED PROGRAMMING

Now that you can put both values and functions in a Dictionary, you can actually use a Dictionary like an object.

Above we used a Dictionary for translating Dutch to English. We might want to do the same for other languages. Let's first make an object (aka Dictionary) that has the translate function, but no words to translate: >

```
        :let transdict = {}
        :function transdict.translate(line) dict
        :  return join(map(split(a:line), 'get(self.words, v:val, "???")'))
        :endfunction
```

It's slightly different from the function above, using 'self.words' to lookup word translations. But we don't have a self.words. Thus you could call this an abstract class.

Now we can instantiate a Dutch translation object: >

```
        :let uk2nl = copy(transdict)
        :let uk2nl.words = {'one': 'een', 'two': 'twee', 'three': 'drie'}
        :echo uk2nl.translate('three one')
<       drie een ~
```

And a German translator: >

```
        :let uk2de = copy(transdict)
        :let uk2de.words = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}
        :echo uk2de.translate('three one')
<       drei eins ~
```

You see that the copy() function is used to make a copy of the "transdict" Dictionary and then the copy is changed to add the words. The original remains the same, of course.

Now you can go one step further, and use your preferred translator: >

```
        :if $LANG =~ "de"
        :  let trans = uk2de
        :else
        :  let trans = uk2nl
        :endif
        :echo trans.translate('one two three')
<       een twee drie ~
```

Here "trans" refers to one of the two objects (Dictionaries). No copy is made. More about List and Dictionary identity can be found at |list-identity| and |dict-identity|.

Now you might use a language that isn't supported. You can overrule the translate() function to do nothing: >

```
        :let uk2uk = copy(transdict)
        :function! uk2uk.translate(line)
        :  return a:line
        :endfunction
        :echo uk2uk.translate('three one wladiwostok')
<       three one wladiwostok ~
```

Notice that a ! was used to overwrite the existing function reference. Now use "uk2uk" when no recognized language is found: >

```
        :if $LANG =~ "de"
        :  let trans = uk2de
        :elseif $LANG =~ "nl"
        :  let trans = uk2nl
        :else
        :  let trans = uk2uk
        :endif
        :echo trans.translate('one two three')
<       one two three ~
```

For further reading see |Lists| and |Dictionaries|.

==

41.9 Exceptions

Let's start with an example: >

```
        :try
        :   read ~/templates/pascal.tmpl
        :catch /E484:/
        :   echo "Sorry, the Pascal template file cannot be found."
        :endtry
```

The ":read" command will fail if the file does not exist. Instead of generating an error message, this code catches the error and gives the user a nice message.

For the commands in between ":try" and ":endtry" errors are turned into exceptions. An exception is a string. In the case of an error the string contains the error message. And every error message has a number. In this case, the error we catch contains "E484:". This number is guaranteed to stay the same (the text may change, e.g., it may be translated).

When the ":read" command causes another error, the pattern "E484:" will not match in it. Thus this exception will not be caught and result in the usual error message.

You might be tempted to do this: >

```
	:try
	:   read ~/templates/pascal.tmpl
	:catch
	:   echo "Sorry, the Pascal template file cannot be found."
	:endtry
```

This means all errors are caught. But then you will not see errors that are useful, such as "E21: Cannot make changes, 'modifiable' is off".

Another useful mechanism is the ":finally" command: >

```
	:let tmp = tempname()
	:try
	:   exe ".,$write " . tmp
	:   exe "!filter " . tmp
	:   .,$delete
	:   exe "$read " . tmp
	:finally
	:   call delete(tmp)
	:endtry
```

This filters the lines from the cursor until the end of the file through the "filter" command, which takes a file name argument. No matter if the filtering works, something goes wrong in between ":try" and ":finally" or the user cancels the filtering by pressing CTRL-C, the "call delete(tmp)" is always executed. This makes sure you don't leave the temporary file behind.

More information about exception handling can be found in the reference manual: |exception-handling|.

==

41.10 Various remarks

Here is a summary of items that apply to Vim scripts. They are also mentioned elsewhere, but form a nice checklist.

The end-of-line character depends on the system. For Unix a single <NL> character is used. For MS-DOS, Windows, OS/2 and the like, <CR><LF> is used. This is important when using mappings that end in a <CR>. See |:source_crnl|.

WHITE SPACE

Blank lines are allowed and ignored.

Leading whitespace characters (blanks and TABs) are always ignored. The whitespaces between parameters (e.g. between the "set" and the "cpoptions" in the example below) are reduced to one blank character and plays the role of a separator, the whitespaces after the last (visible) character may or may not be ignored depending on the situation, see below.

For a ":set" command involving the "=" (equal) sign, such as in: >

```
	:set cpoptions    =aABceFst
```

the whitespace immediately before the "=" sign is ignored. But there can be no whitespace after the "=" sign!

To include a whitespace character in the value of an option, it must be escaped by a "\" (backslash) as in the following example: >

```
        :set tags=my\ nice\ file
```

The same example written as: >

```
        :set tags=my nice file
```

will issue an error, because it is interpreted as: >

```
        :set tags=my
        :set nice
        :set file
```

COMMENTS

The character " (the double quote mark) starts a comment. Everything after and including this character until the end-of-line is considered a comment and is ignored, except for commands that don't consider comments, as shown in examples below. A comment can start on any character position on the line.

There is a little "catch" with comments for some commands. Examples: >

```
        :abbrev dev development         " shorthand
        :map <F3> o#include             " insert include
        :execute cmd                    " do it
        :!ls *.c                        " list C files
```

The abbreviation 'dev' will be expanded to 'development   " shorthand'. The mapping of <F3> will actually be the whole line after the 'o# ....' including the '" insert include'. The "execute" command will give an error. The "!" command will send everything after it to the shell, causing an error for an unmatched '"' character.

There can be no comment after ":map", ":abbreviate", ":execute" and "!" commands (there are a few more commands with this restriction). For the ":map", ":abbreviate" and ":execute" commands there is a trick: >

```
        :abbrev dev development|" shorthand
        :map <F3> o#include|" insert include
        :execute cmd                    |" do it
```

With the '|' character the command is separated from the next one. And that next command is only a comment. For the last command you need to do two things: |:execute| and use '|': >

```
        :exe '!ls *.c'                  |" list C files
```

Notice that there is no white space before the '|' in the abbreviation and mapping. For these commands, any character until the end-of-line or '|' is included. As a consequence of this behavior, you don't always see that trailing whitespace is included: >

```
        :map <F4> o#include
```

To spot these problems, you can set the 'list' option when editing vimrc files.

For Unix there is one special way to comment a line, that allows making a Vim script executable: >

```
        #!/usr/bin/env vim -S
        echo "this is a Vim script"
        quit
```

The "#" command by itself lists a line with the line number. Adding an exclamation mark changes it into doing nothing, so that you can add the shell command to execute the rest of the file. |:#!| |-S|

PITFALLS

Even bigger problem arises in the following example: >

```
        :map ,ab o#include
        :unmap ,ab 
```

Here the unmap command will not work, because it tries to unmap ",ab ". This does not exist as a mapped sequence. An error will be issued, which is very hard to identify, because the ending whitespace character in ":unmap ,ab " is not visible.

And this is the same as what happens when one uses a comment after an 'unmap' command: >

```
        :unmap ,ab     " comment
```

Here the comment part will be ignored. However, Vim will try to unmap ',ab   ', which does not exist. Rewrite it as: >

```
        :unmap ,ab|    " comment
```

RESTORING THE VIEW

Sometimes you want to make a change and go back to where the cursor was. Restoring the relative position would also be nice, so that the same line appears at the top of the window.

   This example yanks the current line, puts it above the first line in the file and then restores the view: >

```
        map ,p ma"aYHmbgg"aP`bzt`a
```

What this does: >

```
        ma"aYHmbgg"aP`bzt`a
<       ma                      set mark a at cursor position
          "aY                   yank current line into register a
             Hmb                go to top line in window and set mark b there
                gg              go to first line in file
                  "aP           put the yanked line above it
                     `b         go back to top line in display
                       zt       position the text in the window as before
                         `a     go back to saved cursor position
```

PACKAGING

To avoid your function names to interfere with functions that you get from others, use this scheme:

- Prepend a unique string before each function name. I often use an abbreviation. For example, "OW_" is used for the option window functions.
- Put the definition of your functions together in a file. Set a global variable to indicate that the functions have been loaded. When sourcing the file again, first unload the functions.

Example: >

```
        " This is the XXX package
```

```
	if exists("XXX_loaded")
	  delfun XXX_one
	  delfun XXX_two
	endif

	function XXX_one(a)
		... body of function ...
	endfun

	function XXX_two(b)
		... body of function ...
	endfun

	let XXX_loaded = 1
```

==

## *41.11* Writing a plugin				*write-plugin*

You can write a Vim script in such a way that many people can use it. This is called a plugin. Vim users can drop your script in their plugin directory and use its features right away |add-plugin|.

There are actually two types of plugins:

```
  global plugins: For all types of files.
filetype plugins: Only for files of a specific type.
```

In this section the first type is explained. Most items are also relevant for writing filetype plugins. The specifics for filetype plugins are in the next section |write-filetype-plugin|.

NAME

First of all you must choose a name for your plugin. The features provided by the plugin should be clear from its name. And it should be unlikely that someone else writes a plugin with the same name but which does something different. And please limit the name to 8 characters, to avoid problems on old Windows systems.

A script that corrects typing mistakes could be called "typecorr.vim". We will use it here as an example.

For the plugin to work for everybody, it should follow a few guidelines. This will be explained step-by-step. The complete example plugin is at the end.

BODY

Let's start with the body of the plugin, the lines that do the actual work: >

```
 14	iabbrev teh the
 15	iabbrev otehr other
 16	iabbrev wnat want
 17	iabbrev synchronisation
 18		\ synchronization
 19	let s:count = 4
```

The actual list should be much longer, of course.

The line numbers have only been added to explain a few things, don't put them in your plugin file!

HEADER

You will probably add new corrections to the plugin and soon have several versions lying around. And when distributing this file, people will want to know who wrote this wonderful plugin and where they can send remarks. Therefore, put a header at the top of your plugin: >

```
  1     " Vim global plugin for correcting typing mistakes
  2     " Last Change:  2000 Oct 15
  3     " Maintainer:   Bram Moolenaar <Bram@vim.org>
```

About copyright and licensing: Since plugins are very useful and it's hardly worth restricting their distribution, please consider making your plugin either public domain or use the Vim |license|. A short note about this near the top of the plugin should be sufficient. Example: >

```
  4     " License:      This file is placed in the public domain.
```

LINE CONTINUATION, AVOIDING SIDE EFFECTS   *use-cpo-save*

In line 18 above, the line-continuation mechanism is used |line-continuation|. Users with 'compatible' set will run into trouble here, they will get an error message. We can't just reset 'compatible', because that has a lot of side effects. To avoid this, we will set the 'cpoptions' option to its Vim default value and restore it later. That will allow the use of line-continuation and make the script work for most people. It is done like this: >

```
 11     let s:save_cpo = &cpo
 12     set cpo&vim
 ..
 42     let &cpo = s:save_cpo
 43     unlet s:save_cpo
```

We first store the old value of 'cpoptions' in the s:save_cpo variable. At the end of the plugin this value is restored.

Notice that a script-local variable is used |s:var|. A global variable could already be in use for something else. Always use script-local variables for things that are only used in the script.

NOT LOADING

It's possible that a user doesn't always want to load this plugin. Or the system administrator has dropped it in the system-wide plugin directory, but a user has his own plugin he wants to use. Then the user must have a chance to disable loading this specific plugin. This will make it possible: >

```
  6     if exists("g:loaded_typecorr")
  7       finish
  8     endif
  9     let g:loaded_typecorr = 1
```

This also avoids that when the script is loaded twice it would cause error messages for redefining functions and cause trouble for autocommands that are added twice.

The name is recommended to start with "loaded_" and then the file name of the plugin, literally. The "g:" is prepended just to avoid mistakes when using the variable in a function (without "g:" it would be a variable local to the function).

Using "finish" stops Vim from reading the rest of the file, it's much quicker than using if-endif around the whole file.

MAPPING

Now let's make the plugin more interesting: We will add a mapping that adds a correction for the word under the cursor. We could just pick a key sequence for this mapping, but the user might already use it for something else. To allow the user to define which keys a mapping in a plugin uses, the <Leader> item can be used: >

```
  map <unique> <Leader>a  <Plug>TypecorrAdd
```

The "<Plug>TypecorrAdd" thing will do the work, more about that further on.

The user can set the "mapleader" variable to the key sequence that he wants this mapping to start with. Thus if the user has done: >

```
let mapleader = "_"
```

the mapping will define "_a". If the user didn't do this, the default value will be used, which is a backslash. Then a map for "\a" will be defined.

Note that <unique> is used, this will cause an error message if the mapping already happened to exist. |:map-<unique>|

But what if the user wants to define his own key sequence? We can allow that with this mechanism: >

```
if !hasmapto('<Plug>TypecorrAdd')
  map <unique> <Leader>a  <Plug>TypecorrAdd
endif
```

This checks if a mapping to "<Plug>TypecorrAdd" already exists, and only defines the mapping from "<Leader>a" if it doesn't. The user then has a chance of putting this in his vimrc file: >

```
map ,c  <Plug>TypecorrAdd
```

Then the mapped key sequence will be ",c" instead of "_a" or "\a".

PIECES

If a script gets longer, you often want to break up the work in pieces. You can use functions or mappings for this. But you don't want these functions and mappings to interfere with the ones from other scripts. For example, you could define a function Add(), but another script could try to define the same function. To avoid this, we define the function local to the script by prepending it with "s:".

We will define a function that adds a new typing correction: >

```
function s:Add(from, correct)
  let to = input("type the correction for " . a:from . ": ")
  exe ":iabbrev " . a:from . " " . to
..
```

```
 36     endfunction
```

Now we can call the function s:Add() from within this script. If another script also defines s:Add(), it will be local to that script and can only be called from the script it was defined in. There can also be a global Add() function (without the "s:"), which is again another function.

<SID> can be used with mappings. It generates a script ID, which identifies the current script. In our typing correction plugin we use it like this: >

```
 24     noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
 ..
 28     noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
```

Thus when a user types "\a", this sequence is invoked: >

```
        \a  ->  <Plug>TypecorrAdd  ->  <SID>Add  ->  :call <SID>Add()
```

If another script would also map <SID>Add, it would get another script ID and thus define another mapping.

Note that instead of s:Add() we use <SID>Add() here. That is because the mapping is typed by the user, thus outside of the script. The <SID> is translated to the script ID, so that Vim knows in which script to look for the Add() function.

This is a bit complicated, but it's required for the plugin to work together with other plugins. The basic rule is that you use <SID>Add() in mappings and s:Add() in other places (the script itself, autocommands, user commands).

We can also add a menu entry to do the same as the mapping: >

```
 26     noremenu <script> Plugin.Add\ Correction      <SID>Add
```

The "Plugin" menu is recommended for adding menu items for plugins. In this case only one item is used. When adding more items, creating a submenu is recommended. For example, "Plugin.CVS" could be used for a plugin that offers CVS operations "Plugin.CVS.checkin", "Plugin.CVS.checkout", etc.

Note that in line 28 ":noremap" is used to avoid that any other mappings cause trouble. Someone may have remapped ":call", for example. In line 24 we also use ":noremap", but we do want "<SID>Add" to be remapped. This is why "<script>" is used here. This only allows mappings which are local to the script. |:map-<script>| The same is done in line 26 for ":noremenu". |:menu-<script>|

<SID> AND <Plug>   *using-<Plug>*

Both <SID> and <Plug> are used to avoid that mappings of typed keys interfere with mappings that are only to be used from other mappings. Note the difference between using <SID> and <Plug>:

<Plug> is visible outside of the script. It is used for mappings which the user might want to map a key sequence to. <Plug> is a special code that a typed key will never produce.
        To make it very unlikely that other plugins use the same sequence of characters, use this structure: <Plug> scriptname mapname
        In our example the scriptname is "Typecorr" and the mapname is "Add". This results in "<Plug>TypecorrAdd". Only the first character of scriptname and mapname is uppercase, so that we can see where mapname starts.

<SID> is the script ID, a unique identifier for a script. Internally Vim translates <SID> to "<SNR>123_", where "123" can be any number. Thus a function "<SID>Add()" will have a name "<SNR>11_Add()" in one script, and "<SNR>22_Add()" in another. You can see this if you use the ":function" command to get a list of functions. The translation of <SID> in mappings is exactly the same, that's how you can call a script-local function from a mapping.

USER COMMAND

Now let's add a user command to add a correction: >

```
if !exists(":Correct")
  command -nargs=1  Correct  :call s:Add(<q-args>, 0)
endif
```

The user command is defined only if no command with the same name already exists. Otherwise we would get an error here. Overriding the existing user command with ":command!" is not a good idea, this would probably make the user wonder why the command he defined himself doesn't work. |:command|

SCRIPT VARIABLES

When a variable starts with "s:" it is a script variable. It can only be used inside a script. Outside the script it's not visible. This avoids trouble with using the same variable name in different scripts. The variables will be kept as long as Vim is running. And the same variables are used when sourcing the same script again. |s:var|

The fun is that these variables can also be used in functions, autocommands and user commands that are defined in the script. In our example we can add a few lines to count the number of corrections: >

```
let s:count = 4
..
function s:Add(from, correct)
..
  let s:count = s:count + 1
  echo s:count . " corrections now"
endfunction
```

First s:count is initialized to 4 in the script itself. When later the s:Add() function is called, it increments s:count. It doesn't matter from where the function was called, since it has been defined in the script, it will use the local variables from this script.

THE RESULT

Here is the resulting complete example: >

```
" Vim global plugin for correcting typing mistakes
" Last Change:  2000 Oct 15
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" License:      This file is placed in the public domain.

if exists("g:loaded_typecorr")
  finish
endif
```

```
  9     let g:loaded_typecorr = 1
 10
 11     let s:save_cpo = &cpo
 12     set cpo&vim
 13
 14     iabbrev teh the
 15     iabbrev otehr other
 16     iabbrev wnat want
 17     iabbrev synchronisation
 18             \ synchronization
 19     let s:count = 4
 20
 21     if !hasmapto('<Plug>TypecorrAdd')
 22       map <unique> <Leader>a  <Plug>TypecorrAdd
 23     endif
 24     noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
 25
 26     noremenu <script> Plugin.Add\ Correction      <SID>Add
 27
 28     noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
 29
 30     function s:Add(from, correct)
 31       let to = input("type the correction for " . a:from . ": ")
 32       exe ":iabbrev " . a:from . " " . to
 33       if a:correct | exe "normal viws\<C-R>\" \b\e" | endif
 34       let s:count = s:count + 1
 35       echo s:count . " corrections now"
 36     endfunction
 37
 38     if !exists(":Correct")
 39       command -nargs=1  Correct  :call s:Add(<q-args>, 0)
 40     endif
 41
 42     let &cpo = s:save_cpo
 43     unlet s:save_cpo
```

Line 33 wasn't explained yet. It applies the new correction to the word under the cursor. The |:normal| command is used to use the new abbreviation. Note that mappings and abbreviations are expanded here, even though the function was called from a mapping defined with ":noremap".

Using "unix" for the 'fileformat' option is recommended. The Vim scripts will then work everywhere. Scripts with 'fileformat' set to "dos" do not work on Unix. Also see |:source_crnl|. To be sure it is set right, do this before writing the file: >

```
	:set fileformat=unix
```

DOCUMENTATION   *write-local-help*

It's a good idea to also write some documentation for your plugin. Especially when its behavior can be changed by the user. See |add-local-help| for how they are installed.

Here is a simple example for a plugin help file, called "typecorr.txt": >

```
	*typecorr.txt*	Plugin for correcting typing mistakes

	If you make typing mistakes, this plugin will have them corrected
	automatically.

```

```
There are currently only a few corrections.  Add your own if you like.

Mappings:
<Leader>a   or   <Plug>TypecorrAdd
        Add a correction for the word under the cursor.

Commands:
:Correct {word}
        Add a correction for {word}.

                                                *typecorr-settings*
This plugin doesn't have any settings.
```

The first line is actually the only one for which the format matters. It will be extracted from the help file to be put in the "LOCAL ADDITIONS:" section of help.txt |local-additions|. The first "*" must be in the first column of the first line. After adding your help file do ":help" and check that the entries line up nicely.

You can add more tags inside ** in your help file. But be careful not to use existing help tags. You would probably use the name of your plugin in most of them, like "typecorr-settings" in the example.

Using references to other parts of the help in || is recommended. This makes it easy for the user to find associated help.

FILETYPE DETECTION *plugin-filetype*

If your filetype is not already detected by Vim, you should create a filetype detection snippet in a separate file. It is usually in the form of an autocommand that sets the filetype when the file name matches a pattern. Example: >

```
        au BufNewFile,BufRead *.foo                     set filetype=foofoo
```

Write this single-line file as "ftdetect/foofoo.vim" in the first directory that appears in 'runtimepath'. For Unix that would be "~/.vim/ftdetect/foofoo.vim". The convention is to use the name of the filetype for the script name.

You can make more complicated checks if you like, for example to inspect the contents of the file to recognize the language. Also see |new-filetype|.

SUMMARY *plugin-special*

Summary of special things to use in a plugin:

s:name	Variables local to the script.
<SID>	Script-ID, used for mappings and functions local to the script.
hasmapto()	Function to test if the user already defined a mapping for functionality the script offers.
<Leader>	Value of "mapleader", which the user defines as the keys that plugin mappings start with.
:map <unique>	Give a warning if a mapping already exists.

:noremap <script>	Use only mappings local to the script, not global mappings.
exists(":Cmd")	Check if a user command already exists.

==

41.12 Writing a filetype plugin *write-filetype-plugin* *ftplugin*

A filetype plugin is like a global plugin, except that it sets options and defines mappings for the current buffer only. See |add-filetype-plugin| for how this type of plugin is used.

First read the section on global plugins above |41.11|. All that is said there also applies to filetype plugins. There are a few extras, which are explained here. The essential thing is that a filetype plugin should only have an effect on the current buffer.

DISABLING

If you are writing a filetype plugin to be used by many people, they need a chance to disable loading it. Put this at the top of the plugin: >

```
	" Only do this when not done yet for this buffer
	if exists("b:did_ftplugin")
	  finish
	endif
	let b:did_ftplugin = 1
```

This also needs to be used to avoid that the same plugin is executed twice for the same buffer (happens when using an ":edit" command without arguments).

Now users can disable loading the default plugin completely by making a filetype plugin with only this line: >

```
	let b:did_ftplugin = 1
```

This does require that the filetype plugin directory comes before $VIMRUNTIME in 'runtimepath'!

If you do want to use the default plugin, but overrule one of the settings, you can write the different setting in a script: >

```
	setlocal textwidth=70
```

Now write this in the "after" directory, so that it gets sourced after the distributed "vim.vim" ftplugin |after-directory|. For Unix this would be "~/.vim/after/ftplugin/vim.vim". Note that the default plugin will have set "b:did_ftplugin", but it is ignored here.

OPTIONS

To make sure the filetype plugin only affects the current buffer use the >

```
	:setlocal
```

command to set options. And only set options which are local to a buffer (see the help for the option to check that). When using |:setlocal| for global options or options local to a window, the value will change for many buffers, and that is not what a filetype plugin should do.

When an option has a value that is a list of flags or items, consider using "+=" and "-=" to keep the existing value. Be aware that the user may have changed an option value already. First resetting to the default value and then changing it is often a good idea. Example: >

```
	:setlocal formatoptions& formatoptions+=ro
```

MAPPINGS

To make sure mappings will only work in the current buffer use the >

```
	:map <buffer>
```

command. This needs to be combined with the two-step mapping explained above. An example of how to define functionality in a filetype plugin: >

```
	if !hasmapto('<Plug>JavaImport')
	  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport
	endif
	noremap <buffer> <unique> <Plug>JavaImport oimport ""<Left><Esc>
```

|hasmapto()| is used to check if the user has already defined a map to <Plug>JavaImport. If not, then the filetype plugin defines the default mapping. This starts with |<LocalLeader>|, which allows the user to select the key(s) he wants filetype plugin mappings to start with. The default is a backslash.
"<unique>" is used to give an error message if the mapping already exists or overlaps with an existing mapping.
|:noremap| is used to avoid that any other mappings that the user has defined interferes. You might want to use ":noremap <script>" to allow remapping mappings defined in this script that start with <SID>.

The user must have a chance to disable the mappings in a filetype plugin, without disabling everything. Here is an example of how this is done for a plugin for the mail filetype: >

```
	" Add mappings, unless the user didn't want this.
	if !exists("no_plugin_maps") && !exists("no_mail_maps")
	  " Quote text by inserting "> "
	  if !hasmapto('<Plug>MailQuote')
	    vmap <buffer> <LocalLeader>q <Plug>MailQuote
	    nmap <buffer> <LocalLeader>q <Plug>MailQuote
	  endif
	  vnoremap <buffer> <Plug>MailQuote :s/^/> /<CR>
	  nnoremap <buffer> <Plug>MailQuote :.,$s/^/> /<CR>
	endif
```

Two global variables are used:

|no_plugin_maps|	disables mappings for all filetype plugins
|no_mail_maps|	disables mappings for the "mail" filetype

USER COMMANDS

To add a user command for a specific file type, so that it can only be used in one buffer, use the "-buffer" argument to |:command|. Example: >

```
	:command -buffer  Make  make %:r.s
```

VARIABLES

A filetype plugin will be sourced for each buffer of the type it's for. Local script variables |s:var| will be shared between all invocations. Use local buffer variables |b:var| if you want a variable specifically for one buffer.

FUNCTIONS

When defining a function, this only needs to be done once. But the filetype plugin will be sourced every time a file with this filetype will be opened. This construct makes sure the function is only defined once: >

```
        :if !exists("*s:Func")
        :  function s:Func(arg)
        :    ...
        :  endfunction
        :endif
<
```

UNDO   *undo_indent* *undo_ftplugin*

When the user does ":setfiletype xyz" the effect of the previous filetype should be undone. Set the b:undo_ftplugin variable to the commands that will undo the settings in your filetype plugin. Example: >

```
        let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
                \ . "| unlet b:match_ignorecase b:match_words b:match_skip"
```

Using ":setlocal" with "<" after the option name resets the option to its global value. That is mostly the best way to reset the option value.

This does require removing the "C" flag from 'cpoptions' to allow line continuation, as mentioned above |use-cpo-save|.

For undoing the effect of an indent script, the b:undo_indent variable should be set accordingly.

FILE NAME

The filetype must be included in the file name |ftplugin-name|. Use one of these three forms:

```
        .../ftplugin/stuff.vim
        .../ftplugin/stuff_foo.vim
        .../ftplugin/stuff/bar.vim
```

"stuff" is the filetype, "foo" and "bar" are arbitrary names.

SUMMARY   *ftplugin-special*

Summary of special things to use in a filetype plugin:

<LocalLeader>   Value of "maplocalleader", which the user defines as
                       the keys that filetype plugin mappings start with.

:map <buffer>   Define a mapping local to the buffer.

:noremap <script>   Only remap mappings defined in this script that start
                       with <SID>.

:setlocal	Set an option for the current buffer only.
:command -buffer	Define a user command local to the buffer.
exists("*s:Func")	Check if a function was already defined.

Also see |plugin-special|, the special things used for all plugins.

==

41.13 Writing a compiler plugin *write-compiler-plugin*

A compiler plugin sets options for use with a specific compiler. The user can load it with the |:compiler| command. The main use is to set the 'errorformat' and 'makeprg' options.

Easiest is to have a look at examples. This command will edit all the default compiler plugins: >

```
	:next $VIMRUNTIME/compiler/*.vim
```

Use |:next| to go to the next plugin file.

There are two special items about these files. First is a mechanism to allow a user to overrule or add to the default file. The default files start with: >

```
	:if exists("current_compiler")
	:  finish
	:endif
	:let current_compiler = "mine"
```

When you write a compiler file and put it in your personal runtime directory (e.g., ~/.vim/compiler for Unix), you set the "current_compiler" variable to make the default file skip the settings.

:CompilerSet

The second mechanism is to use ":set" for ":compiler!" and ":setlocal" for ":compiler". Vim defines the ":CompilerSet" user command for this. However, older Vim versions don't, thus your plugin should define it then. This is an example: >

```
  if exists(":CompilerSet") != 2
    command -nargs=* CompilerSet setlocal <args>
  endif
  CompilerSet errorformat&		" use the default 'errorformat'
  CompilerSet makeprg=nmake
```

When you write a compiler plugin for the Vim distribution or for a system-wide runtime directory, use the mechanism mentioned above. When "current_compiler" was already set by a user plugin nothing will be done.

When you write a compiler plugin to overrule settings from a default plugin, don't check "current_compiler". This plugin is supposed to be loaded last, thus it should be in a directory at the end of 'runtimepath'. For Unix that could be ~/.vim/after/compiler.

==

41.14 Writing a plugin that loads quickly *write-plugin-quickload*

A plugin may grow and become quite long. The startup delay may become noticeable, while you hardly ever use the plugin. Then it's time for a quickload plugin.

The basic idea is that the plugin is loaded twice. The first time user commands and mappings are defined that offer the functionality. The second time the functions that implement the functionality are defined.

It may sound surprising that quickload means loading a script twice. What we mean is that it loads quickly the first time, postponing the bulk of the script to the second time, which only happens when you actually use it. When you always use the functionality it actually gets slower!

Note that since Vim 7 there is an alternative: use the |autoload| functionality |41.15|.

The following example shows how it's done: >

```
	" Vim global plugin for demonstrating quick loading
	" Last Change:	2005 Feb 25
	" Maintainer:	Bram Moolenaar <Bram@vim.org>
	" License:	This file is placed in the public domain.

	if !exists("s:did_load")
		command -nargs=* BNRead  call BufNetRead(<f-args>)
		map <F19> :call BufNetWrite('something')<CR>

		let s:did_load = 1
		exe 'au FuncUndefined BufNet* source ' . expand('<sfile>')
		finish
	endif

	function BufNetRead(...)
		echo 'BufNetRead(' . string(a:000) . ')'
		" read functionality here
	endfunction

	function BufNetWrite(...)
		echo 'BufNetWrite(' . string(a:000) . ')'
		" write functionality here
	endfunction
```

When the script is first loaded "s:did_load" is not set. The commands between the "if" and "endif" will be executed. This ends in a |:finish| command, thus the rest of the script is not executed.

The second time the script is loaded "s:did_load" exists and the commands after the "endif" are executed. This defines the (possible long) BufNetRead() and BufNetWrite() functions.

If you drop this script in your plugin directory Vim will execute it on startup. This is the sequence of events that happens:

1. The "BNRead" command is defined and the <F19> key is mapped when the script is sourced at startup. A |FuncUndefined| autocommand is defined. The ":finish" command causes the script to terminate early.

2. The user types the BNRead command or presses the <F19> key. The BufNetRead() or BufNetWrite() function will be called.

3. Vim can't find the function and triggers the |FuncUndefined| autocommand event. Since the pattern "BufNet*" matches the invoked function, the command "source fname" will be executed. "fname" will be equal to the name of the script, no matter where it is located, because it comes from expanding "<sfile>" (see |expand()|).

4. The script is sourced again, the "s:did_load" variable exists and the

functions are defined.

Notice that the functions that are loaded afterwards match the pattern in the |FuncUndefined| autocommand. You must make sure that no other plugin defines functions that match this pattern.

41.15 Writing library scripts *write-library-script*

Some functionality will be required in several places. When this becomes more than a few lines you will want to put it in one script and use it from many scripts. We will call that one script a library script.

Manually loading a library script is possible, so long as you avoid loading it when it's already done. You can do this with the |exists()| function. Example: >

```
	if !exists('*MyLibFunction')
	   runtime library/mylibscript.vim
	endif
	call MyLibFunction(arg)
```

Here you need to know that MyLibFunction() is defined in a script "library/mylibscript.vim" in one of the directories in 'runtimepath'.

To make this a bit simpler Vim offers the autoload mechanism. Then the example looks like this: >

```
	call mylib#myfunction(arg)
```

That's a lot simpler, isn't it? Vim will recognize the function name and when it's not defined search for the script "autoload/mylib.vim" in 'runtimepath'. That script must define the "mylib#myfunction()" function.

You can put many other functions in the mylib.vim script, you are free to organize your functions in library scripts. But you must use function names where the part before the '#' matches the script name. Otherwise Vim would not know what script to load.

If you get really enthusiastic and write lots of library scripts, you may want to use subdirectories. Example: >

```
	call netlib#ftp#read('somefile')
```

For Unix the library script used for this could be:

```
	~/.vim/autoload/netlib/ftp.vim
```

Where the function is defined like this: >

```
	function netlib#ftp#read(fname)
		"  Read the file fname through ftp
	endfunction
```

Notice that the name the function is defined with is exactly the same as the name used for calling the function. And the part before the last '#' exactly matches the subdirectory and script name.

You can use the same mechanism for variables: >

```
	let weekdays = dutch#weekdays
```

This will load the script "autoload/dutch.vim", which should contain something like: >

```
	let dutch#weekdays = ['zondag', 'maandag', 'dinsdag', 'woensdag',
		\ 'donderdag', 'vrijdag', 'zaterdag']
```

Further reading: |autoload|.

==

41.16 Distributing Vim scripts			*distribute-script*

Vim users will look for scripts on the Vim website: http://www.vim.org. If you made something that is useful for others, share it!

Vim scripts can be used on any system. There might not be a tar or gzip command. If you want to pack files together and/or compress them the "zip" utility is recommended.

For utmost portability use Vim itself to pack scripts together. This can be done with the Vimball utility. See |vimball|.

It's good if you add a line to allow automatic updating. See |glvs-plugins|.

==

Next chapter: |usr_42.txt| Add new menus

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_42.txt	For Vim version 8.0. Last change: 2008 May 05

VIM USER MANUAL - by Bram Moolenaar

Add new menus

By now you know that Vim is very flexible. This includes the menus used in the GUI. You can define your own menu entries to make certain commands easily accessible. This is for mouse-happy users only.

```
     Next chapter: |usr_43.txt|  Using filetypes
 Previous chapter: |usr_41.txt|  Write a Vim script
Table of contents: |usr_toc.txt|
```

==

42.1 Introduction

The menus that Vim uses are defined in the file "$VIMRUNTIME/menu.vim". If you want to write your own menus, you might first want to look through that file.
   To define a menu item, use the ":menu" command. The basic form of this command is as follows: >

```
	:menu {menu-item} {keys}
```

The {menu-item} describes where on the menu to put the item. A typical {menu-item} is "File.Save", which represents the item "Save" under the "File" menu. A dot is used to separate the names. Example: >

```
	:menu File.Save  :update<CR>
```

The ":update" command writes the file when it was modified.

You can add another level: "Edit.Settings.Shiftwidth" defines a submenu "Settings" under the "Edit" menu, with an item "Shiftwidth". You could use even deeper levels. Don't use this too much, you need to move the mouse quite a bit to use such an item.

The ":menu" command is very similar to the ":map" command: the left side specifies how the item is triggered and the right hand side defines the characters that are executed. {keys} are characters, they are used just like you would have typed them. Thus in Insert mode, when {keys} is plain text, that text is inserted.

ACCELERATORS

The ampersand character (&) is used to indicate an accelerator. For instance, you can use Alt-F to select "File" and S to select "Save". (The 'winaltkeys' option may disable this though!). Therefore, the {menu-item} looks like "&File.&Save". The accelerator characters will be underlined in the menu.

You must take care that each key is used only once in each menu. Otherwise you will not know which of the two will actually be used. Vim doesn't warn you for this.

PRIORITIES

The actual definition of the File.Save menu item is as follows: >

```
	:menu 10.340 &File.&Save<Tab>:w  :confirm w<CR>
```

The number 10.340 is called the priority number. It is used by the editor to decide where it places the menu item. The first number (10) indicates the position on the menu bar. Lower numbered menus are positioned to the left, higher numbers to the right.

These are the priorities used for the standard menus:

```
	  10	20     40     50      60       70		9999

	+------------------------------------------------------------+
	| File	Edit  Tools  Syntax  Buffers  Window		Help |
	+------------------------------------------------------------+
```

Notice that the Help menu is given a very high number, to make it appear on the far right.

The second number (340) determines the location of the item within the pull-down menu. Lower numbers go on top, higher number on the bottom. These are the priorities in the File menu:

```
			+-----------------+
	    10.310	|Open...	  |
	    10.320	|Split-Open...	  |
	    10.325	|New		  |
	    10.330	|Close		  |
	    10.335	|---------------- |
	    10.340	|Save		  |
	    10.350	|Save As...	  |
	    10.400	|---------------- |
	    10.410	|Split Diff with  |
	    10.420	|Split Patched By |
	    10.500	|---------------- |
            10.510      |Print           |
            10.600      |--------------- |
            10.610      |Save-Exit       |
            10.620      |Exit            |
                        +----------------+
```

Notice that there is room in between the numbers. This is where you can insert your own items, if you really want to (it's often better to leave the standard menus alone and add a new menu for your own items).

When you create a submenu, you can add another ".number" to the priority. Thus each name in {menu-item} has its priority number.

SPECIAL CHARACTERS

The {menu-item} in this example is "&File.&Save<Tab>:w". This brings up an important point: {menu-item} must be one word. If you want to put a dot, space or tabs in the name, you either use the <> notation (<Space> and <Tab>, for instance) or use the backslash (\) escape. >

```
        :menu 10.305 &File.&Do\ It\.\.\. :exit<CR>
```

In this example, the name of the menu item "Do It..." contains a space and the command is ":exit<CR>".

The <Tab> character in a menu name is used to separate the part that defines the menu name from the part that gives a hint to the user. The part after the <Tab> is displayed right aligned in the menu. In the File.Save menu the name used is "&File.&Save<Tab>:w". Thus the menu name is "File.Save" and the hint is ":w".

SEPARATORS

The separator lines, used to group related menu items together, can be defined by using a name that starts and ends in a '-'. For example "-sep-". When using several separators the names must be different. Otherwise the names don't matter.

The command from a separator will never be executed, but you have to define one anyway. A single colon will do. Example: >

```
        :amenu 20.510 Edit.-sep3- :
```

==

42.2 Menu commands

You can define menu items that exist for only certain modes. This works just like the variations on the ":map" command:

:menu	Normal, Visual and Operator-pending mode
:nmenu	Normal mode
:vmenu	Visual mode
:omenu	Operator-pending mode
:menu!	Insert and Command-line mode
:imenu	Insert mode
:cmenu	Command-line mode
:amenu	All modes

To avoid that the commands of a menu item are being mapped, use the command ":noremenu", ":nnoremenu", ":anoremenu", etc.

USING :AMENU

The ":amenu" command is a bit different. It assumes that the {keys} you give are to be executed in Normal mode. When Vim is in Visual or Insert mode when the menu is used, Vim first has to go back to Normal mode. ":amenu" inserts a CTRL-C or CTRL-O for you. For example, if you use this command:
>

```
        :amenu  90.100 Mine.Find\ Word  *
```

Then the resulting menu commands will be:

```
        Normal mode:            *
        Visual mode:            CTRL-C *
        Operator-pending mode:  CTRL-C *
        Insert mode:            CTRL-O *
        Command-line mode:      CTRL-C *
```

When in Command-line mode the CTRL-C will abandon the command typed so far. In Visual and Operator-pending mode CTRL-C will stop the mode. The CTRL-O in Insert mode will execute the command and then return to Insert mode.
   CTRL-O only works for one command. If you need to use two or more commands, put them in a function and call that function. Example: >

```
        :amenu  Mine.Next\ File  :call <SID>NextFile()<CR>
        :function <SID>NextFile()
        :  next
        :  1/^Code
        :endfunction
```

This menu entry goes to the next file in the argument list with ":next". Then it searches for the line that starts with "Code".
   The <SID> before the function name is the script ID. This makes the function local to the current Vim script file. This avoids problems when a function with the same name is defined in another script file. See |<SID>|.

SILENT MENUS

The menu executes the {keys} as if you typed them. For a ":" command this means you will see the command being echoed on the command line. If it's a long command, the hit-Enter prompt will appear. That can be very annoying!
   To avoid this, make the menu silent. This is done with the <silent> argument. For example, take the call to NextFile() in the previous example. When you use this menu, you will see this on the command line:

```
        :call <SNR>34_NextFile() ~
```

To avoid this text on the command line, insert "<silent>" as the first argument: >

```
        :amenu <silent> Mine.Next\ File :call <SID>NextFile()<CR>
```

Don't use "<silent>" too often. It is not needed for short commands. If you make a menu for someone else, being able the see the executed command will give him a hint about what he could have typed, instead of using the mouse.

LISTING MENUS

When a menu command is used without a {keys} part, it lists the already defined menus. You can specify a {menu-item}, or part of it, to list specific menus. Example: >

```
	:amenu
```

This lists all menus. That's a long list! Better specify the name of a menu to get a shorter list: >

```
	:amenu Edit
```

This lists only the "Edit" menu items for all modes. To list only one specific menu item for Insert mode: >

```
	:imenu Edit.Undo
```

Take care that you type exactly the right name. Case matters here. But the '&' for accelerators can be omitted. The <Tab> and what comes after it can be left out as well.

DELETING MENUS

To delete a menu, the same command is used as for listing, but with "menu" changed to "unmenu". Thus ":menu" becomes, ":unmenu", ":nmenu" becomes ":nunmenu", etc. To delete the "Tools.Make" item for Insert mode: >

```
	:iunmenu Tools.Make
```

You can delete a whole menu, with all its items, by using the menu name. Example: >

```
	:aunmenu Syntax
```

This deletes the Syntax menu and all the items in it.

==

42.3 Various

You can change the appearance of the menus with flags in 'guioptions'. In the default value they are all included, except "M". You can remove a flag with a command like: >

```
	:set guioptions-=m
<
	m		When removed the menubar is not displayed.

	M		When added the default menus are not loaded.

	g		When removed the inactive menu items are not made grey
			but are completely removed.  (Does not work on all
			systems.)

	t		When removed the tearoff feature is not enabled.
```

The dotted line at the top of a menu is not a separator line. When you select this item, the menu is "teared-off": It is displayed in a separate window. This is called a tearoff menu. This is useful when you use the same menu often.

For translating menu items, see |:menutrans|.

Since the mouse has to be used to select a menu item, it is a good idea to use the ":browse" command for selecting a file. And ":confirm" to get a dialog instead of an error message, e.g., when the current buffer contains changes.

These two can be combined: >

```
        :amenu File.Open  :browse confirm edit<CR>
```

The ":browse" makes a file browser appear to select the file to edit. The ":confirm" will pop up a dialog when the current buffer has changes. You can then select to save the changes, throw them away or cancel the command.

For more complicated items, the confirm() and inputdialog() functions can be used. The default menus contain a few examples.

==

42.4 Toolbar and popup menus

There are two special menus: ToolBar and PopUp. Items that start with these names do not appear in the normal menu bar.

TOOLBAR

The toolbar appears only when the "T" flag is included in the 'guioptions' option.

The toolbar uses icons rather than text to represent the command. For example, the {menu-item} named "ToolBar.New" causes the "New" icon to appear on the toolbar.

The Vim editor has 28 built-in icons. You can find a table here: |builtin-tools|. Most of them are used in the default toolbar. You can redefine what these items do (after the default menus are setup).

You can add another bitmap for a toolbar item. Or define a new toolbar item with a bitmap. For example, define a new toolbar item with: >

```
        :tmenu ToolBar.Compile  Compile the current file
        :amenu ToolBar.Compile  :!cc %:S -o %:r:S<CR>
```

Now you need to create the icon. For MS-Windows it must be in bitmap format, with the name "Compile.bmp". For Unix XPM format is used, the file name is "Compile.xpm". The size must be 18 by 18 pixels. On MS-Windows other sizes can be used as well, but it will look ugly.

Put the bitmap in the directory "bitmaps" in one of the directories from 'runtimepath'. E.g., for Unix "~/.vim/bitmaps/Compile.xpm".

You can define tooltips for the items in the toolbar. A tooltip is a short text that explains what a toolbar item will do. For example "Open file". It appears when the mouse pointer is on the item, without moving for a moment. This is very useful if the meaning of the picture isn't that obvious.
Example: >

```
        :tmenu ToolBar.Make  Run make in the current directory
<
```

> Note:
> Pay attention to the case used. "Toolbar" and "toolbar" are different from "ToolBar"!

To remove a tooltip, use the |:tunmenu| command.

The 'toolbar' option can be used to display text instead of a bitmap, or both text and a bitmap. Most people use just the bitmap, since the text takes quite a bit of space.

POPUP MENU

The popup menu pops up where the mouse pointer is. On MS-Windows you activate it by clicking the right mouse button. Then you can select an item with the left mouse button. On Unix the popup menu is used by pressing and holding the right mouse button.

The popup menu only appears when the 'mousemodel' has been set to "popup" or "popup_setpos". The difference between the two is that "popup_setpos" moves the cursor to the mouse pointer position. When clicking inside a selection, the selection will be used unmodified. When there is a selection but you click outside of it, the selection is removed.

There is a separate popup menu for each mode. Thus there are never grey items like in the normal menus.

What is the meaning of life, the universe and everything? *42*
Douglas Adams, the only person who knew what this question really was about is now dead, unfortunately. So now you might wonder what the meaning of death is...

==

Next chapter: |usr_43.txt| Using filetypes

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_43.txt For Vim version 8.0. Last change: 2015 Oct 23

VIM USER MANUAL - by Bram Moolenaar

Using filetypes

When you are editing a file of a certain type, for example a C program or a shell script, you often use the same option settings and mappings. You quickly get tired of manually setting these each time. This chapter explains how to do it automatically.

```
     Next chapter: |usr_44.txt|  Your own syntax highlighted
 Previous chapter: |usr_42.txt|  Add new menus
Table of contents: |usr_toc.txt|
```

==

43.1 Plugins for a filetype *filetype-plugin*

How to start using filetype plugins has already been discussed here: |add-filetype-plugin|. But you probably are not satisfied with the default settings, because they have been kept minimal. Suppose that for C files you want to set the 'softtabstop' option to 4 and define a mapping to insert a three-line comment. You do this with only two steps:

your-runtime-dir

1. Create your own runtime directory. On Unix this usually is "~/.vim". In this directory create the "ftplugin" directory: >

```
	mkdir ~/.vim
	mkdir ~/.vim/ftplugin
```

<

When you are not on Unix, check the value of the 'runtimepath' option to see where Vim will look for the "ftplugin" directory: >

```
	set runtimepath
```

< You would normally use the first directory name (before the first comma).

You might want to prepend a directory name to the 'runtimepath' option in your |vimrc| file if you don't like the default value.

2. Create the file "~/.vim/ftplugin/c.vim", with the contents: >

```
	setlocal softtabstop=4
	noremap <buffer> <LocalLeader>c o/**************<CR><CR>/<Esc>
	let b:undo_ftplugin = "setl softtabstop< | unmap <buffer> <LocalLeader>c"
```

Try editing a C file. You should notice that the 'softtabstop' option is set to 4. But when you edit another file it's reset to the default zero. That is because the ":setlocal" command was used. This sets the 'softtabstop' option only locally to the buffer. As soon as you edit another buffer, it will be set to the value set for that buffer. For a new buffer it will get the default value or the value from the last ":set" command.

Likewise, the mapping for "\c" will disappear when editing another buffer. The ":map <buffer>" command creates a mapping that is local to the current buffer. This works with any mapping command: ":map!", ":vmap", etc. The |<LocalLeader>| in the mapping is replaced with the value of the "maplocalleader" variable.

The line to set b:undo_ftplugin is for when the filetype is set to another value. In that case you will want to undo your preferences. The b:undo_ftplugin variable is executed as a command. Watch out for characters with a special meaning inside a string, such as a backslash.

You can find examples for filetype plugins in this directory: >

```
	$VIMRUNTIME/ftplugin/
```

More details about writing a filetype plugin can be found here: |write-plugin|.

==

43.2 Adding a filetype

If you are using a type of file that is not recognized by Vim, this is how to get it recognized. You need a runtime directory of your own. See |your-runtime-dir| above.

Create a file "filetype.vim" which contains an autocommand for your filetype. (Autocommands were explained in section |40.3|.) Example: >

```
	augroup filetypedetect
	au BufNewFile,BufRead *.xyz     setf xyz
	augroup END
```

This will recognize all files that end in ".xyz" as the "xyz" filetype. The ":augroup" commands put this autocommand in the "filetypedetect" group. This allows removing all autocommands for filetype detection when doing ":filetype off". The "setf" command will set the 'filetype' option to its argument, unless it was set already. This will make sure that 'filetype' isn't set twice.

You can use many different patterns to match the name of your file. Directory names can also be included. See |autocmd-patterns|. For example, the files under "/usr/share/scripts/" are all "ruby" files, but don't have the expected file name extension. Adding this to the example above: >

```
	augroup filetypedetect
	au BufNewFile,BufRead *.xyz			setf xyz
```

```
        au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
        augroup END
```

However, if you now edit a file /usr/share/scripts/README.txt, this is not a ruby file. The danger of a pattern ending in "*" is that it quickly matches too many files. To avoid trouble with this, put the filetype.vim file in another directory, one that is at the end of 'runtimepath'. For Unix for example, you could use "~/.vim/after/filetype.vim".

You now put the detection of text files in ~/.vim/filetype.vim: >

```
        augroup filetypedetect
        au BufNewFile,BufRead *.txt                     setf text
        augroup END
```

That file is found in 'runtimepath' first. Then use this in ~/.vim/after/filetype.vim, which is found last: >

```
        augroup filetypedetect
        au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
        augroup END
```

What will happen now is that Vim searches for "filetype.vim" files in each directory in 'runtimepath'. First ~/.vim/filetype.vim is found. The autocommand to catch *.txt files is defined there. Then Vim finds the filetype.vim file in $VIMRUNTIME, which is halfway 'runtimepath'. Finally ~/.vim/after/filetype.vim is found and the autocommand for detecting ruby files in /usr/share/scripts is added.

When you now edit /usr/share/scripts/README.txt, the autocommands are checked in the order in which they were defined. The *.txt pattern matches, thus "setf text" is executed to set the filetype to "text". The pattern for ruby matches too, and the "setf ruby" is executed. But since 'filetype' was already set to "text", nothing happens here.

When you edit the file /usr/share/scripts/foobar the same autocommands are checked. Only the one for ruby matches and "setf ruby" sets 'filetype' to ruby.

RECOGNIZING BY CONTENTS

If your file cannot be recognized by its file name, you might be able to recognize it by its contents. For example, many script files start with a line like:

```
        #!/bin/xyz ~
```

To recognize this script create a file "scripts.vim" in your runtime directory (same place where filetype.vim goes). It might look like this: >

```
        if did_filetype()
          finish
        endif
        if getline(1) =~ '^#!.*[/\\]xyz\>'
          setf xyz
        endif
```

The first check with did_filetype() is to avoid that you will check the contents of files for which the filetype was already detected by the file name. That avoids wasting time on checking the file when the "setf" command won't do anything.

The scripts.vim file is sourced by an autocommand in the default filetype.vim file. Therefore, the order of checks is:

```
	1. filetype.vim files before $VIMRUNTIME in 'runtimepath'
	2. first part of $VIMRUNTIME/filetype.vim
	3. all scripts.vim files in 'runtimepath'
	4. remainder of $VIMRUNTIME/filetype.vim
	5. filetype.vim files after $VIMRUNTIME in 'runtimepath'
```

If this is not sufficient for you, add an autocommand that matches all files
and sources a script or executes a function to check the contents of the file.

==

Next chapter: |usr_44.txt| Your own syntax highlighted

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_44.txt   For Vim version 8.0. Last change: 2017 May 06

VIM USER MANUAL - by Bram Moolenaar

Your own syntax highlighted

Vim comes with highlighting for a couple of hundred different file types. If
the file you are editing isn't included, read this chapter to find out how to
get this type of file highlighted. Also see |:syn-define| in the reference
manual.

     Next chapter: |usr_45.txt| Select your language
 Previous chapter: |usr_43.txt| Using filetypes
Table of contents: |usr_toc.txt|

==

44.1 Basic syntax commands

Using an existing syntax file to start with will save you a lot of time. Try
finding a syntax file in $VIMRUNTIME/syntax for a language that is similar.
These files will also show you the normal layout of a syntax file. To
understand it, you need to read the following.

Let's start with the basic arguments. Before we start defining any new
syntax, we need to clear out any old definitions: >

```
	:syntax clear
```

This isn't required in the final syntax file, but very useful when
experimenting.

There are more simplifications in this chapter. If you are writing a syntax
file to be used by others, read all the way through the end to find out the
details.

LISTING DEFINED ITEMS

To check which syntax items are currently defined, use this command: >

```
	:syntax
```

You can use this to check which items have actually been defined. Quite useful when you are experimenting with a new syntax file. It also shows the colors used for each item, which helps to find out what is what.

To list the items in a specific syntax group use: >

```
	:syntax list {group-name}
```

This also can be used to list clusters (explained in |44.8|). Just include the @ in the name.

MATCHING CASE

Some languages are not case sensitive, such as Pascal. Others, such as C, are case sensitive. You need to tell which type you have with the following commands: >

```
	:syntax case match
	:syntax case ignore
```

The "match" argument means that Vim will match the case of syntax elements. Therefore, "int" differs from "Int" and "INT". If the "ignore" argument is used, the following are equivalent: "Procedure", "PROCEDURE" and "procedure".

The ":syntax case" commands can appear anywhere in a syntax file and affect the syntax definitions that follow. In most cases, you have only one ":syntax case" command in your syntax file; if you work with an unusual language that contains both case-sensitive and non-case-sensitive elements, however, you can scatter the ":syntax case" command throughout the file.

==

44.2 Keywords

The most basic syntax elements are keywords. To define a keyword, use the following form: >

```
	:syntax keyword {group} {keyword} ...
```

The {group} is the name of a syntax group. With the ":highlight" command you can assign colors to a {group}. The {keyword} argument is an actual keyword. Here are a few examples: >

```
	:syntax keyword xType int long char
	:syntax keyword xStatement if then else endif
```

This example uses the group names "xType" and "xStatement". By convention, each group name is prefixed by the filetype for the language being defined. This example defines syntax for the x language (eXample language without an interesting name). In a syntax file for "csh" scripts the name "cshType" would be used. Thus the prefix is equal to the value of 'filetype'.

These commands cause the words "int", "long" and "char" to be highlighted one way and the words "if", "then", "else" and "endif" to be highlighted another way. Now you need to connect the x group names to standard Vim names. You do this with the following commands: >

```
	:highlight link xType Type
	:highlight link xStatement Statement
```

This tells Vim to highlight "xType" like "Type" and "xStatement" like "Statement". See |group-name| for the standard names.

UNUSUAL KEYWORDS

The characters used in a keyword must be in the 'iskeyword' option. If you use another character, the word will never match. Vim doesn't give a warning message for this.
   The x language uses the '-' character in keywords. This is how it's done:
>

```
	:setlocal iskeyword+=-
	:syntax keyword xStatement when-not
```

The ":setlocal" command is used to change 'iskeyword' only for the current buffer. Still it does change the behavior of commands like "w" and "*". If that is not wanted, don't define a keyword but use a match (explained in the next section).

The x language allows for abbreviations. For example, "next" can be abbreviated to "n", "ne" or "nex". You can define them by using this command:
>

```
	:syntax keyword xStatement n[ext]
```

This doesn't match "nextone", keywords always match whole words only.

==
44.3 Matches

Consider defining something a bit more complex. You want to match ordinary identifiers. To do this, you define a match syntax item. This one matches any word consisting of only lowercase letters: >

```
	:syntax match xIdentifier /\<\l\+\>/
```

<

	Note:
	Keywords overrule any other syntax item. Thus the keywords "if",
	"then", etc., will be keywords, as defined with the ":syntax keyword"
	commands above, even though they also match the pattern for
	xIdentifier.

The part at the end is a pattern, like it's used for searching. The // is used to surround the pattern (like how it's done in a ":substitute" command). You can use any other character, like a plus or a quote.

Now define a match for a comment. In the x language it is anything from # to the end of a line: >

```
	:syntax match xComment /#.*/
```

Since you can use any search pattern, you can highlight very complex things with a match item. See |pattern| for help on search patterns.

==
44.4 Regions

In the example x language, strings are enclosed in double quotation marks ("). To highlight strings you define a region. You need a region start (double quote) and a region end (double quote). The definition is as follows: >

```
	:syntax region xString start=/"/ end=/"/
```

The "start" and "end" directives define the patterns used to find the start and end of the region. But what about strings that look like this?

```
	"A string with a double quote (\") in it" ~
```

This creates a problem: The double quotation marks in the middle of the string will end the region. You need to tell Vim to skip over any escaped double quotes in the string. Do this with the skip keyword: >

```
	:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

The double backslash matches a single backslash, since the backslash is a special character in search patterns.

When to use a region instead of a match? The main difference is that a match item is a single pattern, which must match as a whole. A region starts as soon as the "start" pattern matches. Whether the "end" pattern is found or not doesn't matter. Thus when the item depends on the "end" pattern to match, you cannot use a region. Otherwise, regions are often simpler to define. And it is easier to use nested items, as is explained in the next section.

==

44.5 Nested items

Take a look at this comment:

```
	%Get input  TODO: Skip white space ~
```

You want to highlight TODO in big yellow letters, even though it is in a comment that is highlighted blue. To let Vim know about this, you define the following syntax groups: >

```
	:syntax keyword xTodo TODO contained
	:syntax match xComment /%.*/ contains=xTodo
```

In the first line, the "contained" argument tells Vim that this keyword can exist only inside another syntax item. The next line has "contains=xTodo". This indicates that the xTodo syntax element is inside it. The result is that the comment line as a whole is matched with "xComment" and made blue. The word TODO inside it is matched by xTodo and highlighted yellow (highlighting for xTodo was setup for this).

RECURSIVE NESTING

The x language defines code blocks in curly braces. And a code block may contain other code blocks. This can be defined this way: >

```
	:syntax region xBlock start=/{/ end=/}/ contains=xBlock
```

Suppose you have this text:

```
	while i < b { ~
		if a { ~
			b = c; ~
		} ~
	} ~
```

First a xBlock starts at the { in the first line. In the second line another { is found. Since we are inside a xBlock item, and it contains itself, a nested xBlock item will start here. Thus the "b = c" line is inside the second level xBlock region. Then a } is found in the next line, which matches with the end pattern of the region. This ends the nested xBlock. Because the } is included in the nested region, it is hidden from the first xBlock region. Then at the last } the first xBlock region ends.

KEEPING THE END

Consider the following two syntax items: >

```
	:syntax region xComment start=/%/ end=/$/ contained
	:syntax region xPreProc start=/#/ end=/$/ contains=xComment
```

You define a comment as anything from % to the end of the line. A preprocessor directive is anything from # to the end of the line. Because you can have a comment on a preprocessor line, the preprocessor definition includes a "contains=xComment" argument. Now look what happens with this text:

```
	#define X = Y  % Comment text ~
	int foo = 1; ~
```

What you see is that the second line is also highlighted as xPreProc. The preprocessor directive should end at the end of the line. That is why you have used "end=/$/". So what is going wrong?

The problem is the contained comment. The comment starts with % and ends at the end of the line. After the comment ends, the preprocessor syntax continues. This is after the end of the line has been seen, so the next line is included as well.

To avoid this problem and to avoid a contained syntax item eating a needed end of line, use the "keepend" argument. This takes care of the double end-of-line matching: >

```
	:syntax region xComment start=/%/ end=/$/ contained
	:syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

CONTAINING MANY ITEMS

You can use the contains argument to specify that everything can be contained. For example: >

```
	:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

All syntax items will be contained in this one. It also contains itself, but not at the same position (that would cause an endless loop).

You can specify that some groups are not contained. Thus contain all groups but the ones that are listed:
>

```
	:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

With the "TOP" item you can include all items that don't have a "contained" argument. "CONTAINED" is used to only include items with a "contained" argument. See |:syn-contains| for the details.

==

44.6 Following groups

The x language has statements in this form:

```
	if (condition) then ~
```

You want to highlight the three items differently. But "(condition)" and "then" might also appear in other places, where they get different highlighting. This is how you can do this: >

```
	:syntax match xIf /if/ nextgroup=xIfCondition skipwhite
	:syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
	:syntax match xThen /then/ contained
```

The "nextgroup" argument specifies which item can come next. This is not required. If none of the items that are specified are found, nothing happens. For example, in this text:

```
	if not (condition) then ~
```

The "if" is matched by xIf. "not" doesn't match the specified nextgroup xIfCondition, thus only the "if" is highlighted.

The "skipwhite" argument tells Vim that white space (spaces and tabs) may appear in between the items. Similar arguments are "skipnl", which allows a line break in between the items, and "skipempty", which allows empty lines. Notice that "skipnl" doesn't skip an empty line, something must match after the line break.

==

44.7 Other arguments

MATCHGROUP

When you define a region, the entire region is highlighted according to the group name specified. To highlight the text enclosed in parentheses () with the group xInside, for example, use the following command: >

```
	:syntax region xInside start=/(/ end=/)/
```

Suppose, that you want to highlight the parentheses differently. You can do this with a lot of convoluted region statements, or you can use the "matchgroup" argument. This tells Vim to highlight the start and end of a region with a different highlight group (in this case, the xParen group): >

```
	:syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

The "matchgroup" argument applies to the start or end match that comes after it. In the previous example both start and end are highlighted with xParen. To highlight the end with xParenEnd: >

```
	:syntax region xInside matchgroup=xParen start=/(/
		\ matchgroup=xParenEnd end=/)/
```

A side effect of using "matchgroup" is that contained items will not match in the start or end of the region. The example for "transparent" uses this.

TRANSPARENT

In a C language file you would like to highlight the () text after a "while" differently from the () text after a "for". In both of these there can be nested () items, which should be highlighted in the same way. You must make sure the () highlighting stops at the matching). This is one way to do this:
>

```
	:syntax region cWhile matchgroup=cWhile start=/while\s*(/ end=/)/
		\ contains=cCondNest
```

```
        :syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
                \ contains=cCondNest
        :syntax region cCondNest start=/(/ end=/)/ contained transparent
```

Now you can give cWhile and cFor different highlighting. The cCondNest item can appear in either of them, but take over the highlighting of the item it is contained in. The "transparent" argument causes this.

Notice that the "matchgroup" argument has the same group as the item itself. Why define it then? Well, the side effect of using a matchgroup is that contained items are not found in the match with the start item then. This avoids that the cCondNest group matches the (just after the "while" or "for". If this would happen, it would span the whole text until the matching) and the region would continue after it. Now cCondNest only matches after the match with the start pattern, thus after the first (.

OFFSETS

Suppose you want to define a region for the text between (and) after an "if". But you don't want to include the "if" or the (and). You can do this by specifying offsets for the patterns. Example: >

```
        :syntax region xCond start=/if\s*(/ms=e+1 end=/)/me=s-1
```

The offset for the start pattern is "ms=e+1". "ms" stands for Match Start. This defines an offset for the start of the match. Normally the match starts where the pattern matches. "e+1" means that the match now starts at the end of the pattern match, and then one character further.

The offset for the end pattern is "me=s-1". "me" stands for Match End. "s-1" means the start of the pattern match and then one character back. The result is that in this text:

```
        if (foo == bar) ~
```

Only the text "foo == bar" will be highlighted as xCond.

More about offsets here: |:syn-pattern-offset|.

ONELINE

The "oneline" argument indicates that the region does not cross a line boundary. For example: >

```
        :syntax region xIfThen start=/if/ end=/then/ oneline
```

This defines a region that starts at "if" and ends at "then". But if there is no "then" after the "if", the region doesn't match.

> Note:
> When using "oneline" the region doesn't start if the end pattern doesn't match in the same line. Without "oneline" Vim does _not_ check if there is a match for the end pattern. The region starts even when the end pattern doesn't match in the rest of the file.

CONTINUATION LINES AND AVOIDING THEM

Things now become a little more complex. Let's define a preprocessor line. This starts with a # in the first column and continues until the end of the line. A line that ends with \ makes the next line a continuation line. The way you handle this is to allow the syntax item to contain a continuation pattern: >

```
        :syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
        :syntax match xLineContinue "\\$" contained
```

In this case, although xPreProc normally matches a single line, the group contained in it (namely xLineContinue) lets it go on for more than one line. For example, it would match both of these lines:

```
        #define SPAM  spam spam spam \ ~
                        bacon and spam ~
```

In this case, this is what you want. If it is not what you want, you can call for the region to be on a single line by adding "excludenl" to the contained pattern. For example, you want to highlight "end" in xPreProc, but only at the end of the line. To avoid making the xPreProc continue on the next line, like xLineContinue does, use "excludenl" like this: >

```
        :syntax region xPreProc start=/^#/ end=/$/
                \ contains=xLineContinue,xPreProcEnd
        :syntax match xPreProcEnd excludenl /end$/ contained
        :syntax match xLineContinue "\\$" contained
```

"excludenl" must be placed before the pattern. Since "xLineContinue" doesn't have "excludenl", a match with it will extend xPreProc to the next line as before.

==

44.8 Clusters

One of the things you will notice as you start to write a syntax file is that you wind up generating a lot of syntax groups. Vim enables you to define a collection of syntax groups called a cluster.

Suppose you have a language that contains for loops, if statements, while loops, and functions. Each of them contains the same syntax elements: numbers and identifiers. You define them like this: >

```
        :syntax match xFor /^for.*/ contains=xNumber,xIdent
        :syntax match xIf /^if.*/ contains=xNumber,xIdent
        :syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

You have to repeat the same "contains=" every time. If you want to add another contained item, you have to add it three times. Syntax clusters simplify these definitions by enabling you to have one cluster stand for several syntax groups.

To define a cluster for the two items that the three groups contain, use the following command: >

```
        :syntax cluster xState contains=xNumber,xIdent
```

Clusters are used inside other syntax items just like any syntax group. Their names start with @. Thus, you can define the three groups like this: >

```
        :syntax match xFor /^for.*/ contains=@xState
        :syntax match xIf /^if.*/ contains=@xState
        :syntax match xWhile /^while.*/ contains=@xState
```

You can add new group names to this cluster with the "add" argument: >

```
        :syntax cluster xState add=xString
```

You can remove syntax groups from this list as well: >

```
	:syntax cluster xState remove=xNumber
```

==

44.9 Including another syntax file

The C++ language syntax is a superset of the C language. Because you do not want to write two syntax files, you can have the C++ syntax file read in the one for C by using the following command: >

```
	:runtime! syntax/c.vim
```

The ":runtime!" command searches 'runtimepath' for all "syntax/c.vim" files. This makes the C parts of the C++ syntax be defined like for C files. If you have replaced the c.vim syntax file, or added items with an extra file, these will be loaded as well.
   After loading the C syntax items the specific C++ items can be defined. For example, add keywords that are not used in C: >

```
	:syntax keyword cppStatement	new delete this friend using
```

This works just like in any other syntax file.

Now consider the Perl language. A Perl script consists of two distinct parts: a documentation section in POD format, and a program written in Perl itself. The POD section starts with "=head" and ends with "=cut".
   You want to define the POD syntax in one file, and use it from the Perl syntax file. The ":syntax include" command reads in a syntax file and stores the elements it defined in a syntax cluster. For Perl, the statements are as follows: >

```
	:syntax include @Pod <sfile>:p:h/pod.vim
	:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

When "=head" is found in a Perl file, the perlPOD region starts. In this region the @Pod cluster is contained. All the items defined as top-level items in the pod.vim syntax files will match here. When "=cut" is found, the region ends and we go back to the items defined in the Perl file.
   The ":syntax include" command is clever enough to ignore a ":syntax clear" command in the included file. And an argument such as "contains=ALL" will only contain items defined in the included file, not in the file that includes it.
   The "<sfile>:p:h/" part uses the name of the current file (<sfile>), expands it to a full path (:p) and then takes the head (:h). This results in the directory name of the file. This causes the pod.vim file in the same directory to be included.

==

44.10 Synchronizing

Compilers have it easy. They start at the beginning of a file and parse it straight through. Vim does not have it so easy. It must start in the middle, where the editing is being done. So how does it tell where it is?
   The secret is the ":syntax sync" command. This tells Vim how to figure out where it is. For example, the following command tells Vim to scan backward for the beginning or end of a C-style comment and begin syntax coloring from there: >

```
	:syntax sync ccomment
```

You can tune this processing with some arguments. The "minlines" argument tells Vim the minimum number of lines to look backward, and "maxlines" tells the editor the maximum number of lines to scan.

For example, the following command tells Vim to look at least 10 lines before the top of the screen: >

```
	:syntax sync ccomment minlines=10 maxlines=500
```

If it cannot figure out where it is in that space, it starts looking farther and farther back until it figures out what to do. But it looks no farther back than 500 lines. (A large "maxlines" slows down processing. A small one might cause synchronization to fail.)

To make synchronizing go a bit faster, tell Vim which syntax items can be skipped. Every match and region that only needs to be used when actually displaying text can be given the "display" argument.

By default, the comment to be found will be colored as part of the Comment syntax group. If you want to color things another way, you can specify a different syntax group: >

```
	:syntax sync ccomment xAltComment
```

If your programming language does not have C-style comments in it, you can try another method of synchronization. The simplest way is to tell Vim to space back a number of lines and try to figure out things from there. The following command tells Vim to go back 150 lines and start parsing from there: >

```
	:syntax sync minlines=150
```

A large "minlines" value can make Vim slower, especially when scrolling backwards in the file.

Finally, you can specify a syntax group to look for by using this command: >

```
	:syntax sync match {sync-group-name}
		\ grouphere {group-name} {pattern}
```

This tells Vim that when it sees {pattern} the syntax group named {group-name} begins just after the pattern given. The {sync-group-name} is used to give a name to this synchronization specification. For example, the sh scripting language begins an if statement with "if" and ends it with "fi":

```
	if [ --f file.txt ] ; then ~
		echo "File exists" ~
	fi ~
```

To define a "grouphere" directive for this syntax, you use the following command: >

```
	:syntax sync match shIfSync grouphere shIf "\<if\>"
```

The "groupthere" argument tells Vim that the pattern ends a group. For example, the end of the if/fi group is as follows: >

```
	:syntax sync match shIfSync groupthere NONE "\<fi\>"
```

In this example, the NONE tells Vim that you are not in any special syntax region. In particular, you are not inside an if block.

You also can define matches and regions that are with no "grouphere" or "groupthere" arguments. These groups are for syntax groups skipped during synchronization. For example, the following skips over anything inside {}, even if it would normally match another synchronization method: >

```
	:syntax sync match xSpecial /{.*}/
```

More about synchronizing in the reference manual: |:syn-sync|.

44.11 Installing a syntax file

When your new syntax file is ready to be used, drop it in a "syntax" directory in 'runtimepath'. For Unix that would be "~/.vim/syntax".
The name of the syntax file must be equal to the file type, with ".vim" added. Thus for the x language, the full path of the file would be:

```
	~/.vim/syntax/x.vim ~
```

You must also make the file type be recognized. See |43.2|.

If your file works well, you might want to make it available to other Vim users. First read the next section to make sure your file works well for others. Then e-mail it to the Vim maintainer: <maintainer@vim.org>. Also explain how the filetype can be detected. With a bit of luck your file will be included in the next Vim version!

ADDING TO AN EXISTING SYNTAX FILE

We were assuming you were adding a completely new syntax file. When an existing syntax file works, but is missing some items, you can add items in a separate file. That avoids changing the distributed syntax file, which will be lost when installing a new version of Vim.
Write syntax commands in your file, possibly using group names from the existing syntax. For example, to add new variable types to the C syntax file:
>

```
	:syntax keyword cType off_t uint
```

Write the file with the same name as the original syntax file. In this case "c.vim". Place it in a directory near the end of 'runtimepath'. This makes it loaded after the original syntax file. For Unix this would be:

```
	~/.vim/after/syntax/c.vim ~
```

44.12 Portable syntax file layout

Wouldn't it be nice if all Vim users exchange syntax files? To make this possible, the syntax file must follow a few guidelines.

Start with a header that explains what the syntax file is for, who maintains it and when it was last updated. Don't include too much information about changes history, not many people will read it. Example: >

```
	" Vim syntax file
	" Language:	C
	" Maintainer:	Bram Moolenaar <Bram@vim.org>
	" Last Change:	2001 Jun 18
	" Remark:	Included by the C++ syntax.
```

Use the same layout as the other syntax files. Using an existing syntax file as an example will save you a lot of time.

Choose a good, descriptive name for your syntax file. Use lowercase letters and digits. Don't make it too long, it is used in many places: The name of the syntax file "name.vim", 'filetype', b:current_syntax and the start of each syntax group (nameType, nameStatement, nameString, etc).

Start with a check for "b:current_syntax". If it is defined, some other syntax file, earlier in 'runtimepath' was already loaded: >

```
        if exists("b:current_syntax")
          finish
        endif
```

To be compatible with Vim 5.8 use: >

```
        if version < 600
          syntax clear
        elseif exists("b:current_syntax")
          finish
        endif
```

Set "b:current_syntax" to the name of the syntax at the end. Don't forget that included files do this too, you might have to reset "b:current_syntax" if you include two files.

If you want your syntax file to work with Vim 5.x, add a check for v:version. Find an syntax file in the Vim 7.2 distribution for an example.

Do not include anything that is a user preference. Don't set 'tabstop', 'expandtab', etc. These belong in a filetype plugin.

Do not include mappings or abbreviations. Only include setting 'iskeyword' if it is really necessary for recognizing keywords.

To allow users select their own preferred colors, make a different group name for every kind of highlighted item. Then link each of them to one of the standard highlight groups. That will make it work with every color scheme. If you select specific colors it will look bad with some color schemes. And don't forget that some people use a different background color, or have only eight colors available.

For the linking use "hi def link", so that the user can select different highlighting before your syntax file is loaded. Example: >

```
          hi def link nameString        String
          hi def link nameNumber        Number
          hi def link nameCommand       Statement
          ... etc ...
```

Add the "display" argument to items that are not used when syncing, to speed up scrolling backwards and CTRL-L.

==

Next chapter: |usr_45.txt| Select your language

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_45.txt   For Vim version 8.0. Last change: 2008 Nov 15

VIM USER MANUAL - by Bram Moolenaar

Select your language

The messages in Vim can be given in several languages. This chapter explains how to change which one is used. Also, the different ways to work with files in various languages is explained.

Next chapter: |usr_90.txt| Installing Vim
Previous chapter: |usr_44.txt| Your own syntax highlighted
Table of contents: |usr_toc.txt|

==

45.1 Language for Messages

When you start Vim, it checks the environment to find out what language you are using. Mostly this should work fine, and you get the messages in your language (if they are available). To see what the current language is, use this command: >

```
        :language
```

If it replies with "C", this means the default is being used, which is English.

> Note:
> Using different languages only works when Vim was compiled to handle it. To find out if it works, use the ":version" command and check the output for "+gettext" and "+multi_lang". If they are there, you are OK. If you see "-gettext" or "-multi_lang" you will have to find another Vim.

What if you would like your messages in a different language? There are several ways. Which one you should use depends on the capabilities of your system.

The first way is to set the environment to the desired language before starting Vim. Example for Unix: >

```
        env LANG=de_DE.ISO_8859-1  vim
```

This only works if the language is available on your system. The advantage is that all the GUI messages and things in libraries will use the right language as well. A disadvantage is that you must do this before starting Vim. If you want to change language while Vim is running, you can use the second method: >

```
        :language fr_FR.ISO_8859-1
```

This way you can try out several names for your language. You will get an error message when it's not supported on your system. You don't get an error when translated messages are not available. Vim will silently fall back to using English.

To find out which languages are supported on your system, find the directory where they are listed. On my system it is "/usr/share/locale". On some systems it's in "/usr/lib/locale". The manual page for "setlocale" should give you a hint where it is found on your system.

Be careful to type the name exactly as it should be. Upper and lowercase matter, and the '-' and '_' characters are easily confused.

You can also set the language separately for messages, edited text and the time format. See |:language|.

DO-IT-YOURSELF MESSAGE TRANSLATION

If translated messages are not available for your language, you could write them yourself. To do this, get the source code for Vim and the GNU gettext package. After unpacking the sources, instructions can be found in the directory src/po/README.txt.

It's not too difficult to do the translation. You don't need to be a programmer. You must know both English and the language you are translating to, of course.

When you are satisfied with the translation, consider making it available to others. Upload it at vim-online (http://vim.sf.net) or e-mail it to the Vim maintainer <maintainer@vim.org>. Or both.

==

45.2 Language for Menus

The default menus are in English. To be able to use your local language, they must be translated. Normally this is automatically done for you if the environment is set for your language, just like with messages. You don't need to do anything extra for this. But it only works if translations for the language are available.

Suppose you are in Germany, with the language set to German, but prefer to use "File" instead of "Datei". You can switch back to using the English menus this way: >

```
        :set langmenu=none
```

It is also possible to specify a language: >

```
        :set langmenu=nl_NL.ISO_8859-1
```

Like above, differences between "-" and "_" matter. However, upper/lowercase differences are ignored here.

The 'langmenu' option must be set before the menus are loaded. Once the menus have been defined changing 'langmenu' has no direct effect. Therefore, put the command to set 'langmenu' in your vimrc file.

If you really want to switch menu language while running Vim, you can do it this way: >

```
        :source $VIMRUNTIME/delmenu.vim
        :set langmenu=de_DE.ISO_8859-1
        :source $VIMRUNTIME/menu.vim
```

There is one drawback: All menus that you defined yourself will be gone. You will need to redefine them as well.

DO-IT-YOURSELF MENU TRANSLATION

To see which menu translations are available, look in this directory:

```
        $VIMRUNTIME/lang ~
```

The files are called menu_{language}.vim. If you don't see the language you want to use, you can do your own translations. The simplest way to do this is by copying one of the existing language files, and change it.

First find out the name of your language with the ":language" command. Use this name, but with all letters made lowercase. Then copy the file to your own runtime directory, as found early in 'runtimepath'. For example, for Unix you would do: >

```
        :!cp $VIMRUNTIME/lang/menu_ko_kr.euckr.vim ~/.vim/lang/
menu_nl_be.iso_8859-1.vim
```

You will find hints for the translation in "$VIMRUNTIME/lang/README.txt".

==

45.3 Using another encoding

Vim guesses that the files you are going to edit are encoded for your language. For many European languages this is "latin1". Then each byte is one character. That means there are 256 different characters possible. For Asian languages this is not sufficient. These mostly use a double-byte encoding, providing for over ten thousand possible characters. This still isn't enough when a text is to contain several different languages. This is where Unicode comes in. It was designed to include all characters used in commonly used languages. This is the "Super encoding that replaces all others". But it isn't used that much yet.

Fortunately, Vim supports these three kinds of encodings. And, with some restrictions, you can use them even when your environment uses another language than the text.

Nevertheless, when you only edit files that are in the encoding of your language, the default should work fine and you don't need to do anything. The following is only relevant when you want to edit different languages.

> Note:
> Using different encodings only works when Vim was compiled to handle it. To find out if it works, use the ":version" command and check the output for "+multi_byte". If it's there, you are OK. If you see "-multi_byte" you will have to find another Vim.

USING UNICODE IN THE GUI

The nice thing about Unicode is that other encodings can be converted to it and back without losing information. When you make Vim use Unicode internally, you will be able to edit files in any encoding.

Unfortunately, the number of systems supporting Unicode is still limited. Thus it's unlikely that your language uses it. You need to tell Vim you want to use Unicode, and how to handle interfacing with the rest of the system.

Let's start with the GUI version of Vim, which is able to display Unicode characters. This should work: >

```
        :set encoding=utf-8
        :set guifont=-misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

The 'encoding' option tells Vim the encoding of the characters that you use. This applies to the text in buffers (files you are editing), registers, Vim script files, etc. You can regard 'encoding' as the setting for the internals of Vim.

This example assumes you have this font on your system. The name in the example is for the X Window System. This font is in a package that is used to enhance xterm with Unicode support. If you don't have this font, you might find it here:

```
        http://www.cl.cam.ac.uk/~mgk25/download/ucs-fonts.tar.gz ~
```

For MS-Windows, some fonts have a limited number of Unicode characters. Try using the "Courier New" font. You can use the Edit/Select Font... menu to select and try out the fonts available. Only fixed-width fonts can be used though. Example: >

```
        :set guifont=courier_new:h12
```

If it doesn't work well, try getting a fontpack. If Microsoft didn't move it, you can find it here:

```
	http://www.microsoft.com/typography/fonts/default.aspx ~
```

Now you have told Vim to use Unicode internally and display text with a Unicode font. Typed characters still arrive in the encoding of your original language. This requires converting them to Unicode. Tell Vim the language from which to convert with the 'termencoding' option. You can do it like this: >

```
	:let &termencoding = &encoding
	:set encoding=utf-8
```

This assigns the old value of 'encoding' to 'termencoding' before setting 'encoding' to utf-8. You will have to try out if this really works for your setup. It should work especially well when using an input method for an Asian language, and you want to edit Unicode text.

USING UNICODE IN A UNICODE TERMINAL

There are terminals that support Unicode directly. The standard xterm that comes with XFree86 is one of them. Let's use that as an example.

First of all, the xterm must have been compiled with Unicode support. See |UTF8-xterm| how to check that and how to compile it when needed.

Start the xterm with the "-u8" argument. You might also need so specify a font. Example: >

```
   xterm -u8 -fn -misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

Now you can run Vim inside this terminal. Set 'encoding' to "utf-8" as before. That's all.

USING UNICODE IN AN ORDINARY TERMINAL

Suppose you want to work with Unicode files, but don't have a terminal with Unicode support. You can do this with Vim, although characters that are not supported by the terminal will not be displayed. The layout of the text will be preserved. >

```
	:let &termencoding = &encoding
	:set encoding=utf-8
```

This is the same as what was used for the GUI. But it works differently: Vim will convert the displayed text before sending it to the terminal. That avoids that the display is messed up with strange characters.

For this to work the conversion between 'termencoding' and 'encoding' must be possible. Vim will convert from latin1 to Unicode, thus that always works. For other conversions the |+iconv| feature is required.

Try editing a file with Unicode characters in it. You will notice that Vim will put a question mark (or underscore or some other character) in places where a character should be that the terminal can't display. Move the cursor to a question mark and use this command: >

```
	ga
```

Vim will display a line with the code of the character. This gives you a hint about what character it is. You can look it up in a Unicode table. You could actually view a file that way, if you have lots of time at hand.

	Note:
	Since 'encoding' is used for all text inside Vim, changing it makes

all non-ASCII text invalid. You will notice this when using registers and the 'viminfo' file (e.g., a remembered search pattern). It's recommended to set 'encoding' in your vimrc file, and leave it alone.

==

45.4 Editing files with a different encoding

Suppose you have setup Vim to use Unicode, and you want to edit a file that is in 16-bit Unicode. Sounds simple, right? Well, Vim actually uses utf-8 encoding internally, thus the 16-bit encoding must be converted, since there is a difference between the character set (Unicode) and the encoding (utf-8 or 16-bit).

Vim will try to detect what kind of file you are editing. It uses the encoding names in the 'fileencodings' option. When using Unicode, the default value is: "ucs-bom,utf-8,latin1". This means that Vim checks the file to see if it's one of these encodings:

ucs-bom	File must start with a Byte Order Mark (BOM). This allows detection of 16-bit, 32-bit and utf-8 Unicode encodings.
utf-8	utf-8 Unicode. This is rejected when a sequence of bytes is illegal in utf-8.
latin1	The good old 8-bit encoding. Always works.

When you start editing that 16-bit Unicode file, and it has a BOM, Vim will detect this and convert the file to utf-8 when reading it. The 'fileencoding' option (without s at the end) is set to the detected value. In this case it is "utf-16le". That means it's Unicode, 16-bit and little-endian. This file format is common on MS-Windows (e.g., for registry files).

When writing the file, Vim will compare 'fileencoding' with 'encoding'. If they are different, the text will be converted.

An empty value for 'fileencoding' means that no conversion is to be done. Thus the text is assumed to be encoded with 'encoding'.

If the default 'fileencodings' value is not good for you, set it to the encodings you want Vim to try. Only when a value is found to be invalid will the next one be used. Putting "latin1" first doesn't work, because it is never illegal. An example, to fall back to Japanese when the file doesn't have a BOM and isn't utf-8: >

```
        :set fileencodings=ucs-bom,utf-8,sjis
```

See |encoding-values| for suggested values. Other values may work as well. This depends on the conversion available.

FORCING AN ENCODING

If the automatic detection doesn't work you must tell Vim what encoding the file is. Example: >

```
        :edit ++enc=koi8-r russian.txt
```

The "++enc" part specifies the name of the encoding to be used for this file only. Vim will convert the file from the specified encoding, Russian in this example, to 'encoding'. 'fileencoding' will also be set to the specified encoding, so that the reverse conversion can be done when writing the file.

The same argument can be used when writing the file. This way you can actually use Vim to convert a file. Example: >

```
        :write ++enc=utf-8 russian.txt
<
```

Note:
Conversion may result in lost characters. Conversion from an encoding to Unicode and back is mostly free of this problem, unless there are illegal characters. Conversion from Unicode to other encodings often loses information when there was more than one language in the file.

==

45.5 Entering language text

Computer keyboards don't have much more than a hundred keys. Some languages have thousands of characters, Unicode has over hundred thousand. So how do you type these characters?

First of all, when you don't use too many of the special characters, you can use digraphs. This was already explained in |24.9|.

When you use a language that uses many more characters than keys on your keyboard, you will want to use an Input Method (IM). This requires learning the translation from typed keys to resulting character. When you need an IM you probably already have one on your system. It should work with Vim like with other programs. For details see |mbyte-XIM| for the X Window system and |mbyte-IME| for MS-Windows.

KEYMAPS

For some languages the character set is different from latin, but uses a similar number of characters. It's possible to map keys to characters. Vim uses keymaps for this.

Suppose you want to type Hebrew. You can load the keymap like this: >

```
        :set keymap=hebrew
```

Vim will try to find a keymap file for you. This depends on the value of 'encoding'. If no matching file was found, you will get an error message.

Now you can type Hebrew in Insert mode. In Normal mode, and when typing a ":" command, Vim automatically switches to English. You can use this command to switch between Hebrew and English: >

```
        CTRL-^
```

This only works in Insert mode and Command-line mode. In Normal mode it does something completely different (jumps to alternate file).

The usage of the keymap is indicated in the mode message, if you have the 'showmode' option set. In the GUI Vim will indicate the usage of keymaps with a different cursor color.

You can also change the usage of the keymap with the 'iminsert' and 'imsearch' options.

To see the list of mappings, use this command: >

```
        :lmap
```

To find out which keymap files are available, in the GUI you can use the Edit/Keymap menu. Otherwise you can use this command: >

```
        :echo globpath(&rtp, "keymap/*.vim")
```

DO-IT-YOURSELF KEYMAPS

You can create your own keymap file. It's not very difficult. Start with a keymap file that is similar to the language you want to use. Copy it to the "keymap" directory in your runtime directory. For example, for Unix, you would use the directory "~/.vim/keymap".

The name of the keymap file must look like this:

```
	keymap/{name}.vim ~
```

or

```
	keymap/{name}_{encoding}.vim ~
```

{name} is the name of the keymap. Chose a name that is obvious, but different from existing keymaps (unless you want to replace an existing keymap file). {name} cannot contain an underscore. Optionally, add the encoding used after an underscore. Examples:

```
	keymap/hebrew.vim ~
	keymap/hebrew_utf-8.vim ~
```

The contents of the file should be self-explanatory. Look at a few of the keymaps that are distributed with Vim. For the details, see |mbyte-keymap|.

LAST RESORT

If all other methods fail, you can enter any character with CTRL-V:

encoding	type	range ~
8-bit	CTRL-V 123	decimal 0-255
8-bit	CTRL-V x a1	hexadecimal 00-ff
16-bit	CTRL-V u 013b	hexadecimal 0000-ffff
31-bit	CTRL-V U 001303a4	hexadecimal 00000000-7fffffff

Don't type the spaces. See |i_CTRL-V_digit| for the details.

==

Next chapter: |usr_90.txt| Installing Vim

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl: