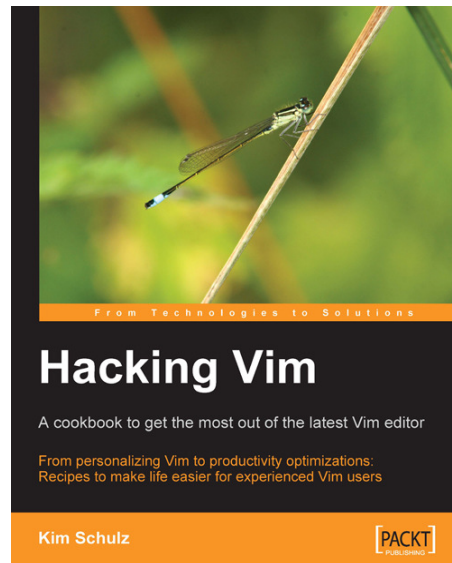




# Hacking Vim

A Cookbook to get the Most out of the Latest Vim Editor

Kim Schulz



## Chapter No. 2

### "Personalizing Vim"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter No. 2 “Personalizing Vim”

A synopsis of the book’s content

Information on where to buy this book

## About the Author

Kim Schulz has an M.Sc. in Software Engineering from Aalborg University in Denmark. He has been an active developer in the Linux and Open Source communities since 1997 and has worked with everything from translation and bug fixing to producing full-blown software systems.

This entire time, Vim has been Kim's editor of choice and it has been the first thing he installs whenever he sits at a new computer.

Today Kim works as a full-time software engineer at CSR Plc. developing software for the next generation wireless technologies.

A lot of Kim's spare time has been spent on developing the open-source CMS Fundament. This has lead to him now owning the web-hosting company Devteam Denmark that specializes in hosting and development of Fundament-based websites.

<b>For More Information: <a href="http://www.PacktPub.com/Vim/book">www.PacktPub.com/Vim/book</a></b>
---

# Hacking Vim: A Cookbook to get the Most out of the Latest Vim Editor

Back in the early days of the computer revolution, system resources were limited and developers had to figure out new ways to optimize their applications. This was also the case with the text editors of that time. One of the most popular editors of that time was an editor called Vim. It was optimized to near-perfection for the limited system resources on which it ran.

The world has come a long way since then, and even though the system resources have grown, many still stick with the Vim editor.

At first sight, the Vim editor might not look like much. However, if you look beneath the simple user-interface, you will discover why this editor is still the favorite editor for so many people, even today!

This editor has nearly every feature you would ever want, and if it's not in the editor, it is possible to add it by creating plugins and scripts. This high level of flexibility makes it ideal for many purposes, and it is also why Vim is still one of the most advanced editors.

New users join the Vim user community every day and want to use this editor in their daily work, and even though Vim sometimes can be complex to use, they still favor it above other editors. This is a book for these Vim users.

With this book, Vim users can make their daily work in the editor more comfortable and thereby optimize their productivity. In this way they will not only have an optimized editor, but also an optimized work-flow. The book will help them move from just using Vim as a simple text editor to a situation where they feel at home and can use it for many of their daily tasks.

For More Information: <a href="http://www.PacktPub.com/Vim/book">www.PacktPub.com/Vim/book</a>
--

## What This Book Covers

*Chapter 1* introduces Vim and a few well-known relatives; their history and relation to vi is briefly described.

*Chapter 2* introduces how to make Vim a better editor for you by modifying it for your personal needs. It shows you ways of modifying fonts, the color scheme, the status line, menus, and toolbar.

*Chapter 3* introduces some of the ways in which Vim helps us to navigate through files easily. It explains an alternative way for boosting navigation through files and buffers in Vim.

*Chapter 4* introduces you to features in Vim. It describes how to use templates, auto-completion, folding, sessions, and working with registers.

*Chapter 5* introduces simple tricks to format text and code. It also discusses how external tool can be used to give Vim just that extra edge it needs to be the perfect editor.

*Chapter 6* is especially for those who want to learn how to extend Vim with scripts. The chapter introduces scripting basics, how to use and install/uninstall scripts, debugging scripts, and lots more.

*Appendix A* has a listing of games that have been implemented with Vim scripting; it also provides an overview of chat and mail scripts and has a section on using Vim as an IDE.

*Appendix B* shows how to keep your Vim configuration files well organized and retain your Vim configuration across computers by storing a copy of it online

For More Information: <a href="http://www.PacktPub.com/Vim/book">www.PacktPub.com/Vim/book</a>
--

# 2

## Personalizing Vim

If you tend to use your computer a lot for editing files, you soon realize that having a good editor is of paramount importance. A good editor will be your best friend and help you with your daily tasks. But what makes an editor good?

Looking at the different editors available, we see that some of them try to be the best editor by developers adding features they think the users need. Others have accepted that they are not the best editor and instead try to be the simplest most, user-friendly, or fastest-loading editor around.

With the Vim editor, no one has decided what's best for you. Instead you are given the opportunity to modify a large range of settings to make Vim fit your needs. This means that the power is in the hands of the user, rather than the hands of the developers of the editor.

Some settings have to do with the actual layout of Vim (e.g. colors and menus), while others change areas that affect how we work with Vim—like key bindings that map certain key combinations to specific tasks.

In this chapter we will introduce a list of recipes that will help you personalize Vim in such a way that it becomes your personal favorite.

You will find recipes for the following personalization tasks:

1. Changing the fonts
2. Changing the color scheme
3. Personal highlighting
4. A more informative status line
5. Toggle menu and toolbar
6. Adding your own menu and toolbar buttons
7. Work area personalization

**For More Information:** [www.PacktPub.com/Vim/book](http://www.PacktPub.com/Vim/book)

Some of these tasks contain more than one recipe because there are different aspects to personalizing Vim for that particular task. It is you, the reader, who decides which recipes (or parts thereof) you would like to use.

Before we start working with Vim, there are some things that you need to know about your Vim installation – where to find the configuration files.

## Where are the Config Files?

When working with Vim, you need to know a range of different configuration files. The location of these files is very dependent on where you have installed Vim, and the operating system that you are using.

In general, there are three configuration files that you must know where to find.

### vimrc

This is the main configuration file for Vim. It exists in two versions – global and personal.

The global **vimrc** file is placed in the folder where all your Vim system files are installed. You can find out the location of this folder by opening Vim and executing the following command in normal mode:

```
:echo $VIM
```

Examples could be:

Linux: **/usr/share/vim/vimrc**

Windows: **c:\program files\vim\vimrc**

The personal vimrc file is placed in your home directory. The location of the home directory is dependent on your operating system. Vim originally was meant for UNIXes, so the personal vimrc file is set to be hidden by adding a dot as the first character in the filename. This normally hides files on UNIXes but not on Microsoft Windows. Instead, the vimrc file is prepended with an underscore on these systems. So, examples would be:

Linux: **/home/kim/.vimrc**

Windows: **c:\documents and settings\kim\\_vimrc**

Whatever you change in the personal vimrc file will overrule any previous setting made in the global vimrc file. This way you can modify the entire configuration without having to ever have access to the global vimrc file.

You can find out what Vim considers as the home directory on your system, by executing the following command in normal mode:

```
:echo $HOME
```

The **vimrc** file contains **ex** (vi predecessor) commands, one on each line, and is the default place to add modifications to the Vim setup. In the rest of the book, this file is just called vimrc.

Your vimrc can use other files as an external source for configurations. In the vimrc file, you use the command `source` like this:

```
source /path/to/external/file
```

Use this to keep the vimrc file clean, and your settings more structured (more about how to keep your vimrc clean in Appendix B).

## gvimrc

The **gvimrc** file is a configuration file specifically for Gvim. It resembles the vimrc file described above, and is placed in the same locations – as a personal version as well as a global version. For example:

Linux: **/home/kim/.gvimrc** and **/usr/share/vim/gvimrc**

Windows: **c:\documents and settings\kim\\_gvimrc**, and  
**c:\program files\vim\gvimrc**

This file is used for GUI-specific settings that only Gvim will be able to use. In the rest of the book, this file is called **gvimrc**

## exrc

This is a configuration file that is only used for backwards compatibility with the old vi/ex editor. It is placed at the same location (both global and local) as vimrc and is used the same way. However, it is almost never used anymore except if you want to use Vim in vi-compatible mode.

## [GVim]<sup>6+</sup> Changing the Fonts

In regular Vim there is not much to do when it comes to changing the font because the font follows the one of the terminal. In Gvim however, you are given the ability to change the font as much as you like.

The main command for changing the font in Linux is:

```
:set guifont=Courier\ 14
```

Where *Courier* can be exchanged with the name of any font that you have, and 14 with any font size you like (size in points—pt).

For changing the font in Windows, use the following command:

```
:set guifont=Courier:14
```

If you are not sure about whether a particular font is available on the computer or not, you can add another font at the end of the command by adding a comma between the two fonts. For example:

```
:set guifont=Courier\ New\ 12, Arial\ 10
```

If the font name contains a whitespace or a comma, you will need to escape it with a backslash. For example:

```
:set guifont=Courier\ New\ 12
```

This command sets the font to Courier New size 12—but only for this session. If you want to have this font every time you edit a file, the same command has to be added to your gvimrc file (without the ':' in front of **set**).



In Gvim on Windows, Linux (using GTK+), Mac OS, or Photon, you can get a font selection window shown if you use the command:

```
:set guifont=*
```

If you tend to use a lot of different fonts depending on what you are currently working with (code, text, log-files, etc.), you can set up Vim to use the correct font according to the filetype. For example, if you want to set the font to Arial size 12 every time a normal text file (.txt) is opened, this can be achieved by adding the following line to your vimrc file:

```
autocmd BufEnter *.txt set guifont=Arial\ 12
```



The window of Gvim will resize itself every time the font is changed. This means that if you use a smaller font you will also (as a default) have a smaller window. You will notice this right away if you add several different filetype commands like the one above, and then open some files of different types. Whenever you switch to a buffer with another filetype, the font will change, and hence the window size too.



You can find more information about changing fonts in the Vim help system under: `:help 'guifont'`

## [Vim]<sup>5+</sup> Changing Color Scheme

Often, when working in a console environment you have only a black background and white text in the foreground. This is, however, both dull and dark to look at. Some colors would be desirable.

As a default, you have the same colors in console Vim as in the console you opened it from. However, Vim has given its users the opportunity to change the colors it uses. This is mostly done with a color scheme file. These files are usually placed in a directory called *colors* wherever you have installed Vim.

You can easily change among the installed color schemes with the command:

```
:colorscheme mycolors
```

where *mycolors* is the name of one of the installed color schemes. If you don't know the names of the installed color schemes, you can place the cursor after writing:

```
:colorscheme
```

and shift through the names by pressing the tab-key. When you find the color scheme you want, you can press the enter key to apply it.

The color scheme does not apply only to foreground and background color, but also to the way code is highlighted, how errors are marked, and other visual markings in the text.

You will find that some color schemes are very alike and only minor things have changed. The reason for this is that the color schemes are user supplied. If some user did not like one of the color settings in a scheme, he or she could just change that single setting and re-release the color scheme under a different name.

Play around with the different color schemes and find the one you like. Now, test it in the situations where you would normally use it, and see if you still like all the color settings. In Chapter 6, we will get back to how you can change a color scheme to fit your needs perfectly.

```
/**
 * Main algorithm to keep processing while there are workunits
 */
private void run() {
    // Only invoke this once even though we receive multiple requests
    if (running) {
        System.out.println("Client is already running");
        return;
    } else {
        running = true;

        // Only do actual work if there are any workunits
        while (currentWorkUnit != null) {
            System.out.println("Acquiring workunits");
            if (!queueWorkunits())
                continue;
            System.out.println("Queueing threads for execution");
            queueWorkers();
            System.out.println("Awaiting results from workers");
            commitResults();
        }
        System.out.println("Ending connection with server");
    }
}
```

## [Vim]<sup>6+</sup> [Gvim]<sup>6+</sup> Personal Highlighting

In Vim, the feature of highlighting things is called **matching**.

With matching, you can make Vim mark almost any combination of letters, words, numbers, sentences, and lines. You can even select how it should mark it (errors in red, important words in green, etc).

Matching is done with the following command:

```
:match Group /pattern/
```

The command has two arguments. The first one is the name of the color group that you will use in the highlight.



Compared to a color scheme, which affects the entire color setup, a color group is a rather small combination of background (or foreground) colors that you can use for things like matches. When Vim is started, a wide range of color groups are set to default colors, depending on the color scheme you have selected.



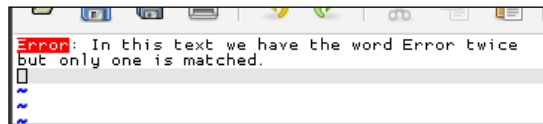
To see a complete list of color groups, use the command :

```
:so $VIMRUNTIME/syntax/hitest.vim
```

The second argument is the actual pattern you want to match. This pattern is a regular expression, and can vary from being very simple to extremely complex, depending on what you want to match. A simple example of the match command in use would be:

```
:match ErrorMsg /^Error/
```

This command looks for the word **Error** (marked with a `^`) at the beginning of all lines. If a match is found, it will be marked with the colors in the *ErrorMsg* color group (typically white text on red background).



If you don't like any of the color groups available, you can always define your own. The command to do this is as follows:

```
:highlight MyGroup ctermbg=red guibg=red gctermfg=yellow  
          guifg=yellow term=bold
```

This command creates a color group called "MyGroup" with a red background and yellow text, in both console (Vim) and GUI (Gvim). You can change the following options according to your preferences:

- **ctermbg** : Background color in console
- **guibg** : Background color in Gvim
- **ctermfg** : Text color in console
- **guifg** : Text color in Gvim
- **gui** : Font formatting in Gvim
- **term** : Font formatting in console (for example, bold)

If you use the name of an existing color group, you will alter that group for the rest of the session.

When using the match command, the given pattern will be matched until you perform a new match or execute the following command:

```
:match NONE
```

The match command can only match one pattern at a time; so Vim has provided you with two extra commands to match up to three patterns at a time. The commands are easy to remember because their names resemble that of the match command:

```
:2match
:3match
```

You might wonder what all this matching is good for, as it can often seem quite useless. To show the strength of matching, here are a few examples:

### Example 1:

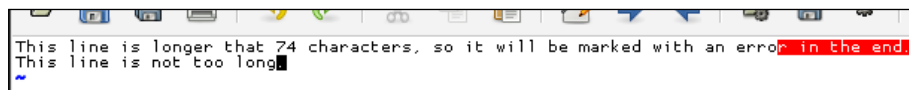
In mails, it is a common rule that you do not write lines more than 74 characters long (a rule that also applies to some older programming languages like for example Fortran-77). In a case like this, it would be nice if Vim could warn you when you reached this specific number of characters.

This can simply be done with the following command:

```
:match ErrorMessage /\%>73v.\+ /
```

Here, every character after the 73rd character will be marked as an error. This match is a regular expression that when broken down consists of:

- `\%>` : Match after column with the number right after this
- `73` : The column number
- `v` : Combined with the previous command, this means that the next part is very **magic**. See `:help magic` for more info.
- `.\+` : Match one or more of any character.



### Example 2:

When coding, it is generally a good rule of thumb that tabs are only to be used to indent code, and not anywhere else. However, for some it can be hard to obey this rule. Now, with the help of a simple match command this can easily be prevented.

The following command will mark any tabs that are not at the beginning of the line (indentation) as an error:

```
:match errorMessage /^[^t]\zs\t\+ /
```

Now you can check if you have forgotten the rule and used the tab key inside the code. Broken down, the match consists of the following parts:

- [^ : Begin a group of characters that should not be matched
- \t : The tab-character
- ] : End of character group.
- \zs : A zero-width match that places the 'matching' at the beginning of the line ignoring any whitespaces
- \t\+ : One or more tabs in a row.

This command says: don't match all the tab-characters, match only the ones that are not used at the beginning of the line (ignoring any whitespaces around it).

```
int Main(int count{
    int number = 4;                 //some number
}
```

If instead of using tabs if you want to use the space character for indentation, then you can change the command to:

```
:match errorMsg /\t/
```


This command just says: match all the tab-characters.

### Example 3:

If you write a lot of IP addresses in your text, sometimes you tend to enter a wrong value in one (like 123.123.123.**256**). To help you prevent this kind of an error, you can add the following match to your vimrc file:

```
match errorMsg /\(2[5][6-9]\|2[6-9][0-9]\|[3-9][0-9][0-9]\)\.[.]\
\ [0-9]\{1,3\}\.[.][0-9]\{1,3\}\.[.][0-9]\{1,3\}\|
\ [0-9]\{1,3\}\.[.]\(2[5][6-9]\|2[6-9][0-9]\|
\ \ [3-9][0-9][0-9]\)\.[.][0-9]\{1,3\}\.[.][0-9]\
\ \{1,3\}\|\ [0-9]\{1,3\}\.[.][0-9]\{1,3\}\.[.]\(2[5]
\ \ [6-9]\|\2[6-9][0-9]\|[3-9][0-9][0-9]\)\.[.]\
\ [0-9]\{1,3\}\
\ \ [0-9]\{1,3\}\.[.][0-9]\{1,3\}\.[.][0-9]\{1,3\}\.[.]\
\ \ (2[5][6-9]\|2[6-9][0-9]\|\ [3-9][0-9][0-9])\)/
```

Even though this seems a bit too complex for solving a small possible error, you have to remember that even if it helps you just once, it has already been worth adding.



If you want to match valid IP addresses, you can use this much simpler command:

```
match todo /\(\(25[0-5]\|2[0-4][0-9]\|[01]\)?[0-9][0-9]\?\)\.\.\)\{3\}\(25[0-5]\|2[0-4][0-9]\|[01]\)?[0-9][0-9]\?\)/
```

## [Vim]<sup>6+</sup> [GVim]<sup>6+</sup> A More Informative Status Line

At the bottom of the Vim editor, you will find two things: the command-line buffer (where you can input commands), and the status line. In the default configuration, Vim has a simple and quite non-informative status line. To the right it shows the number of the current row and column and to the left it shows name of the file currently open (if any).

Whenever you execute a Vim command, the status line will disappear and the command buffer will be shown in that line instead. If the command you execute writes any messages, then those will be shown on the right of the status line.

For simple and fast file editing, this status line is adequate. But if you use Vim every day and for a lot of different file formats, it would be nice to have a more informative statusline.

In this recipe, we see some examples of how the status line can be made a lot more informative with simple methods.

The command that sets how the status line should look is simply called:

```
:set statusline format
```

where **format** is a **printf**-like **string** (known from C programming) that describes how the status line should look.

If you look in the Vim help system by typing `:help 'statusline'`, you will see that the status line can contain a wide variety of pieces of information, some more useful in your daily work than others.

My status line always contains information about:

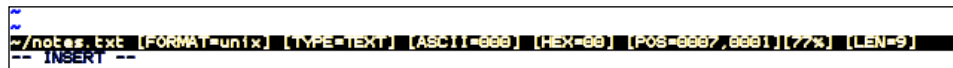
- Name of the file that I am editing
- Format of the file that I am editing (DOS, Unix)
- Filetype as recognized by Vim for the current file
- ASCII and hex value of the character under the cursor

- Position in the document as row and column number
- Length of the file (line count)

The following command will turn your status line into a true information bar with all the above information:

```
:set statusline=%F%m%r%h%w\ [FORMAT=%{&ff}]\ [TYPE=%Y]\ [ASCII=%03.3b]\ [HEX=%02.2B]\ [POS=%04l,%04v] [%p%%]\ [LEN=%L]
```

I have added a '[' around each of the pieces of information, so that it is easier to distinguish them from each other. This is purely to give a visual effect and can be left out if necessary.



However, we now see that the status line still shows the old non-informative status line, as in the default installation. This problem occurs because Vim, by default, does not show the status line at all. Instead, it just shows the command buffer with a little bit of information in it. To tell Vim that you would like to have a real status line shown, you will have to add the following setting to your vimrc file. This command will make sure that your status line is always shown as the second last line in the editor window:

```
:set laststatus=2
```

You will now see that the command buffer gets a place of its own in the last line of the editor window. This way there's always room for the status line and you will always have information about the file right in front of you. The status line does of course take up some of the editing area, but it is then up to you to decide whether it should be shown or not. You can always remove it for the rest of the editing session by executing the following command from within Vim:

```
:set laststatus=0
```

## [Gvim]<sup>6+</sup> Toggle Menu and Toolbar

If you are used to working with Vim in the console mode, you are also quite used to having no menus and toolbars in the top of the window. However, when you move to Gvim, you will soon realize that both the menu and the toolbar are there, by default, in the GUI.

Many believe that extra room for text is far more important than the menu and the toolbar. If you are one of those persons, you might like to remove the menu and toolbar while working in Gvim. However, some scripts add useful functionality in the menu and it is therefore important to have the menus. The solution for this could be toggling if the menu and toolbar is shown or not.

The following code maps the key combination *Ctrl-F2* to toggle the menu and toolbar in Gvim. You can add it to your vimrc file if you want this functionality.

```
map <silent> <C-F2> :if &guioptions =~# 'T' <Bar>
                        \set guioptions-=T <Bar>
                        \set guioptions-=m <bar>
                    \else <Bar>
                        \set guioptions+=T <Bar>
                        \set guioptions+=m <bar>
                    \endif<CR>
```

Now, whenever you don't need the menu and toolbar, you can just press *Ctrl-F2* and you will get the full space for your text.

If you want either the menu or the toolbar to be hidden all the time, add one of the following lines to your vimrc file:

To remove the menu completely:

```
:set guioptions-=m
```

To remove the toolbar completely:

```
:set guioptions-=T
```



Other parts of the GUI can be modified with the `set guioptions` command. To find out what you can modify, look in `:help 'guioptions'`

## [Gvim]<sup>7+</sup> Adding Your Own Menu and Toolbar Buttons

If you are in Gvim, you can make a handy menu with all the functionality you use the most. You might not always need to use it from the menu, but whenever you forget how to use it, you can always just find it there. If you need to get to the functionality really fast, you can even add it directly in the toolbar of Gvim.

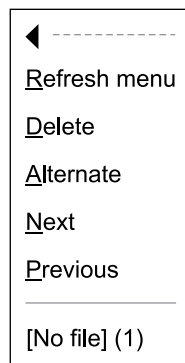


In this recipe, we look at both how to make your own menu and, later, how to add extra buttons to the toolbar in Gvim. Let us start with the menu construction.

## Adding a Menu

Building a menu is basically just executing a command for each item you want in the menu. As long as you follow the right naming convention, you will see a nice little menu with all your items in it.

Let us start with a simple example. Say you want to add a menu like the buffers menu, but for tabs.



The command you will need to use is:

```
:menu menupath command
```

This command works much like the map command, except that instead of mapping a command to a key combination, here the mapping is done to a menu item.

The command has two arguments. The first is the actual path in the menu where the item should be placed, and the second argument is the command that the menu item should execute. If for instance, you want to add a menu item called **Next** to the menu item **Tabs**, then you would need to use a command like this:

```
:menu Tabs.Next <ESC>:tabnext<cr>
```

So now you have a menu called **Tabs** with one menu item called **Next**. What the **Next** menu item does is execute the following command:

```
:tabnext
```

This command is prepended with `<Esc>` to get into the normal mode, and then `<cr>` to actually execute the command. If you haven't added `<Esc>` this command won't work. Another way to get around this is by adding specific menu items according to the current mode. For this Vim has a range of alternatives to the `:menu` command:

- `:nmenu` – for **N**ormal mode.
- `:imenu` – for **I**nsert mode. `^o` is prepended.
- `:vmenu` – for **V**isual mode. `^C` is prepended and `^\^G` is appended.
- `:cmenu` – for **C**ommand-line mode. `^C` is prepended and `^\^G` is appended.
- `:omenu` – for **O**P-pending mode. `^C` is prepended and `^\^G` is appended.



The prepended parts (`^o` and `^C`) are to get into normal mode.

The `^o` (`Ctrl-O`) is especially for insert mode because it gets you back into insert mode after executing the command.

`^\^G` (`Ctrl-\`, `Ctrl-G`) is to handle the special case wherein the global insert mode setting is set to true and Vim has insert mode as the default mode (Vim is mode-less). In this case, it will get you back into insert mode and in the rest of the cases it will get you back in the mode you just came from.

Instead of setting the same menu item for each and every mode, you can just replace the commands with this single command:

```
:amenu menu-path command
```

According to the current mode, this command prepends and appends the right things.

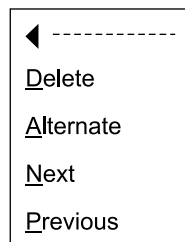
So let's go to our new **Tabs** menu, and add some more items and functionality to it. With the following, it should look similar to the **Buffers** menu:

```
:amenu Tabs.&Delete :confirm tabclose<cr>
:amenu Tabs.&Alternate :confirm tabn #<cr>
:amenu <silent> Tabs.&Next :tabnext<cr>
:amenu <silent>Tabs.&Previous :tabprevious<cr>
```

The observant reader might have noticed that some new things have been added in the commands.

The first thing is the `<silent>` tag in the last two commands. By adding this we can avoid the command being echoed in the command-line buffer during execution.

While this is a purely cosmetic functionality, the `'&'` in the menu path is a more functional extension. By adding an `'&'` in front of one of the letters in the last part of the menu path, you can define a keyboard shortcut for an item. This makes it easy to navigate to that particular item in the menu and execute it.



Let's say that you want to go to the next tab by executing the **Tabs > Next** menu item; now you can do so by simply pressing *Alt-t n*. This is *Alt-t* for opening **Tabs**, and *n* to call the **Next** item—*n* because the '*&*' is in front of the *N* in **Next**. If another menu item uses the same character for a shortcut, you can cycle through them by pressing the *Alt* key repeatedly.



If you would like to have a line that separates some of the items in your menu drop down, you can use the name '**SEP**' for the item and ':' for the command: `:amenu Tabs.-SEP- :`



The menu that we have created will only exist as long as Vim is open in this session, so in order to get it into your menu all the time, you need to add it to your vimrc file (without the ':' in front of the commands).

So now we have a simple tabs menu that looks a bit like the **Buffers** menu. It does not, however, have the functionality that lists active buffers in the **Buffers** menu. This does not make much of a difference when you realize that buffers can be hidden for the user, but tabs cannot. You can, in other words, always see exactly how many tabs you have and what they are called by just looking at the tab bar.

A personal menu can be used for a lot of other interesting things. If you work with many types of files you can even start having menus for specific file types or sub-menus for the different types in the same menu.

A sub-menu is constructed by following the naming convention in the menu path. So if you want to have **Tabs > Navigation > Next**, you will simply have to add the **Next** menu item with the menu path `Tabs.Navigation.&Next`

## [Gvim]<sup>6+</sup> Adding Toolbar Icons

So now that we know how to make our menus, adding our own icons to the toolbar isn't that difficult. Actually, Vim is constructed in such a way that the toolbar is just another menu with a special name. Hence, adding an icon to the toolbar is just like adding an item to a menu.

In the case of a 'toolbar menu', you will be able to add items to it by using a menu-path that starts with the name `ToolBar`. To add an item to the toolbar that gives access for executing the command `:buffers` (show list of open buffers), all you have to do is to execute the following command:

```
:amenu icon=/path/to/icon/myicon.png ToolBar.Bufferlist :buffers<cr>
```

Of course, you will need to have an icon placed somewhere that can be shown in the toolbar.

The path to the icon is given with the argument `icon` to the `amenu` command. If you do not give a path to the file, but only the filename, then Vim will look for the icon in a folder called `bitmaps/` in the Vim runtimepath (execute `:echo $VIMRUNTIME` to see where it is). The type of icons supported is dependant on the system you use it on.

And that's really it! After executing the command, you will see your icon in the toolbar as the last one on the right. If you press it, it will execute the command, `:buffers`, and show you a buffer list.

As with the menus, you can add toolbar buttons that are only shown in specific modes using the mode-specific menu commands `imenu`, `vmenu`, `cmenu`, etc.



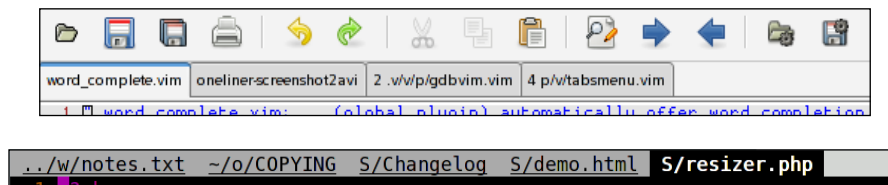
If you want your menu or toolbar icon placed elsewhere than to the right of the others, then you can use priorities. Read more about how in: `:help menu-priority` and `:help sub-menu-priority`

## [Vim]<sup>7+</sup> Modifying Tabs

Ever since the release of Vim version 7.0, there has been support for tabs or tab pages as it is called. Tab pages are not like the normal tabs in other applications; rather they are a way to group your open files. Each tab can contain several open buffers and even several windows at the same time.

What makes tabs special is that the commands you would normally execute on all open buffers/windows (like `:bufdo`, `:windo`, `:all`, `:ball`) are limited to only the windows and buffers in the current tab page.

Normally, tab pages are shown as a list of tabs in the top of the window (just above the editing area). Each tab has a label, which as a default shows the name of the file in the currently active buffer. If more windows are open, at the same time, in the tab page, then the tab label will also show a number telling how many windows.



Sometimes you might like to have the label on the tabs telling you something different. For instance, if you often have one tab for each project, then it would be nice to name the tab according to the name of the project in it.

The label on the tabs is set in a way very much similar to the one used for the status line (see section *A More Informative Status Line*). But here, instead of setting the property `status line`, you set the property `tabline`:

```
:set tabline=tabline-layout
```

or if you are in Gvim:

```
:set guitablabel
```

Even though setting the `tabline` resembles the way you set the status line, it is a bit more troublesome. This is mainly because you need to take care of whether the tab is the active one or not. So let's start with a little example for Vim.

When we have a lot of tabs, they tend to take up too much space in the tab page, especially if they contain the entire name of the file in the currently active buffer. We want to have only the first 6 letters of the name of the active buffer in the tab label. The active tab should also be easy to distinguish from the other tabs; so let's make its colors white on red like error messages.

The following script in Vim script does just that (learn more about how to create Vim scripts in Chapter 6).

```
function ShortTabLine()
  let ret = ''
  for i in range(tabpagenr('$'))
    " select the color group for highlighting active tab
    if i + 1 == tabpagenr()
      let ret .= '%#errorMsg#'
    else
```

```
        let ret .= '%#TabLine#'
    endif

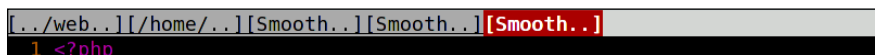
    " find the buffername for the tablabel
    let buflist = tabpagebuflist(i+1)
    let winnr = tabpagewinnr(i+1)
    let buffername = bufname(buflist[winnr - 1])
    let filename = fnamemodify(buffername, ':t')
    " check if there is no name
    if filename == ''
        let filename = 'noname'
    endif
    " only show the first 6 letters of the name and
    " .. if the filename is more than 8 letters long
    if strlen(filename) >= 8
        let ret .= '['. filename[0:5]. '...]'
    else
        let ret .= '['. filename. ']'
    endif
    endif
endfor

" after the last tab fill with TabLineFill and reset tab page #
let ret .= '%#TabLineFill#%T'
return ret
endfunction
```

Now, we have the function and just need to add it to our vimrc file, along with a line that sets the tabline to the output of our function. This can be done with the following command:

```
:set tabline=%!ShortTabLine()
```

The result is a more compact tablist as shown in the following screenshot:



Changing the tabline in Gvim is a bit different, but still follows almost the same basic ideas. However, when in the GUI, you do not have to consider things like the color of the active tab, or whether it is actually active or not because this is all a part of the GUI design itself.

So let's simplify the ShortTabLine() function a bit so that it only sets the tab label:

```
function ShortTabLabel()
    let bufnrlist = tabpagebuflist(v:lnum)
```

```

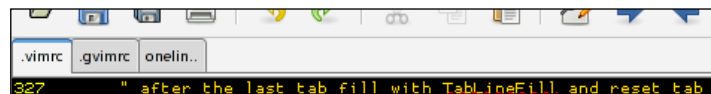
" show only the first 6 letters of the name + ..
let label = bufname(bufnrlist[tabpagewinnr(v:lnum) - 1])
let filename = fnamemodify(label, ':h')
" only add .. if string is more than 8 letters
if strlen(filename) >=8
    let ret=filename[0:5].'..'
else
    let ret = filename
endif
return ret
endfunction

```

So now we just have to set the `guitablabel` property to the output of our function:

```
:set guitablabel=%{ShortTabLabel() }
```

The result will be fine small tabs as shown in the following figure.



If you want to remove the tabs bar completely from Gvim, then you can use the command: **:set showtabline=0** (set to 1 to get it shown again).

So now we have limited the information in the tabs, but we would still like to have the information somewhere. For that we have a nice little tip—use the tool tips.

The nice thing about tool tips is that when you don't activate them (hold your cursor over some area, e.g., a tab) you don't see them. This way you can have the information without it filling up the entire editor.

To set the tool tip for a tab you will need to use the following command:

```
:set guitabtooltip
```

This property should be set to the value you want to show, when the mouse cursor hovers over the tab.

To test it you can try with a simple execution like:

```
:set guitabtooltip='my tooltip'
```

Now, this only shows a static text in the tool tip. We need some more information there. We removed the path from the filenames on the tabs, but sometimes it is actually nice to have this information available. With the tool tips this is easily shown with the following command:

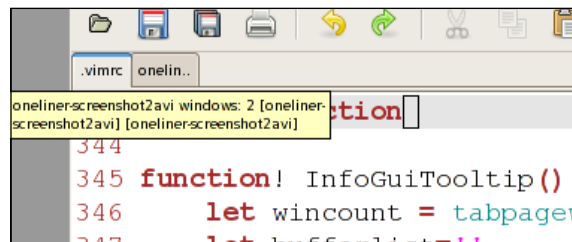
```
:set guitabtooltip=%!bufname($)
```

As with the tabs, the contents of the tool tip can be constructed by a function. Here we have constructed a small function that shows all the information you would normally have in the tabs—but in a more organized way:

```
function! InfoGuiTooltip()  
    "get window count  
    let wincount = tabpagewinnr(tabpagenr(), '$')  
    let bufferlist = ''  
    "get name of active buffers in windows  
    for i in tabpagebuflist()  
        let bufferlist .= '[' . fnamemodify(bufname(i), ':t') . ']'  
    endfor  
    return bufname($). ' windows: ' . wincount . ' ' . bufferlist  
endfunction
```

You use this code described above like this:

```
:set guitabtooltip=%!InfoGuiTooltip()
```



You can probably imagine many other interesting ways to use the small information space the tabs, and tool tips provide, and following the above example, you should have no problems in implementing them.

## Work Area Personalization

In this section, we introduce a list of smaller, good-to-know, modifications for the editor area in Vim. The idea with these recipes is that they all give you some sort of help or optimization when you use Vim for editing text or code.



## [Vim]<sup>7+</sup> [GVim]<sup>7+</sup> Adding a More Visual Cursor

Sometimes, you have a lot of syntax coloring in the file you are editing. This can make the task of tracking the cursor really hard. If you could just mark the line the cursor is currently in, then it would be easier to track it.

Many have tried to fix this with Vim scripts but the results have been near useless (mainly due to slowness, which prevented scrolling longer texts at an acceptable speed). Not until version 7 did Vim have a solution for this, but then it came up with not just one, but two possible solutions for cursor tracking.

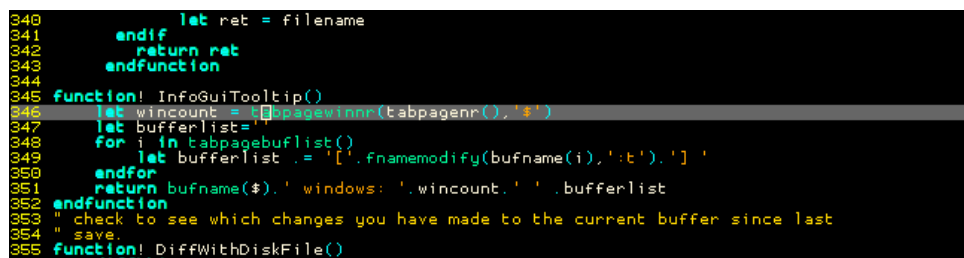
The first one is the **cursorline** command, which basically marks the entire line with, for example, another background color, without breaking the syntax coloring. To turn it on, use the following command:

```
:set cursorline
```

The color it uses is the one defined in the color group *CursorLine*. You can change this to any color or styling you like, for example:

```
:highlight CursorLine guibg=lightblue ctermbg=lightgray
```

See the section *Personal Highlighting* for more info on how to change a color group.



```

340         let ret = filename
341     endif
342     return ret
343 endfunction
344
345 function! InfoGuiTooltip()
346     let wincount = <tabpagenr>(<tabpagenr>,'*')
347     let bufferlist = ''
348     for i in <tabpagebuflist>()
349         let bufferlist .= '['.fnamemodify(bufname(i),':t').'] '
350     endfor
351     return bufname($). ' windows: ' . wincount . ' ' . bufferlist
352 endfunction
353 " check to see which changes you have made to the current buffer since last
354 " save.
355 function! DiffWithDiskFile()
356     diffthis

```

If you are working with a lot of aligned file content (like tab-separated data), the next solution for cursor tracking comes in handy:

```
:set cursorcolumn
```

This command marks the current column the cursor is in by, for example, coloring the entire column in the file.

As with the **cursorline**, you can change the settings for how the cursor column should be marked. The color group to change is named *CursorColumn*.

Adding both the cursor line and column marking makes the cursor look like a crosshair, thus making it impossible to miss.

```
340     let ret = filename
341     endif
342     return ret
343 endfunction
344 function! InfoGuiToolTip()
345     let wincount = tabpagewinnr(tabpagenr(), '#')
346     let bufferlist = ''
347     for i in tabpagebuflist()
348         let bufferlist .= '['.fnamemodify(bufname(i), ':t').'] '
349     endfor
350     return bufname($). ' windows: ' . wincount . ' ' . bufferlist
351 endfunction
352 " check to see which changes you have made to the current buffer since last
353 " save.
354 function! DiffWithDiskFile()
```



#### Warning!

Even though the `cursorline` and `cursorcolumn` functionality is implemented natively in Vim, it can still give quite a slowdown when scrolling through the file.

## [Vim]<sup>6+</sup> [GVim]<sup>6+</sup> Adding Line Numbers

Often when compiling and debugging code, you will get error messages stating that the error is in some line. One could of course start counting lines from the top to find the line, but Vim has a solution to go directly to some line number. Just execute `:xxx` where `xxx` is the line number, and you will be taken to line `xxx`.

Alternatively, you can go into normal mode (press *Esc*) and then simply use `xxxgg` or `xxxG` (again `xxx` is the line number). Sometimes, however, it is nice to have an indication of the line number right there in the editor, and that's where the following command comes in handy:

```
:set number
```

Now you get line numbers to the left of each line in the file. By default, the numbers take up four columns of space—three for numbers and one for spacing. This means that the width of the numbers will be the same until you have more than 999 lines. If you get above this number of lines, an extra column will be added and the content will be moved to the right.

You can of course change the default number of columns used for the line numbers. This can be achieved by changing the following property:

```
:set numberwidth=XXX
```

Replace `xxx` with the number of columns that you want.



Even though it would be nice to make the number of columns higher in order to get more spacing between code and line numbers, this is not achievable with the *numberwidth* property. This is because the line numbers will be right aligned within the columns.

```

21 When we speak of free sof
22 price. Our General Public
23 have the freedom to distrib
24 this service if you wish),
25 if you want it, that you ca
26 in new free programs; and t
27
28 To protect your rights, w
29 anyone to deny you these ri
30 These restrictions translat
~/ontv/COPYING [FORMAT=unix] [TYPE=]
```



You can change the styling of the line numbers, and the columns they are in, by making changes to the color group *LineNr*.

## [Vim]<sup>7+</sup> [GVim]<sup>7+</sup> Spell Checking Your Language

We all know it! Even if we are really good spellers, it still happens from time to time that we misspell a word or hit the wrong keys. In the past, you had to run your texts (that you had written in Vim) through some sort of spell checker like **Aspell** or **Ispell**, which was a tiresome process that could only be performed as a final task—unless you wanted to do it over and over again.

With version 7 of Vim, this troublesome way of spell checking is over. Now, Vim has got a built-in spell checker with support for more than 50 languages from around the world.

The new spell checker marks the wrongly written words as you type them in, so you know there is an error right away.

The command to execute to turn on this helpful spell checker feature is:

```
:set spell
```

This turns on the spell checker with the default language (English). If you don't use English much, and would prefer to use another language in the spell checker, then there is no problem changing this. Just add the code of the language you would like to use to the *spelllang* property, for example:

```
:set spelllang=de
```

Here, the language is set to German (Deutsch) as the spell checker language of choice. The language name can be written in several different formats. American English, for example, can be written as:

- en\_us
- us
- American

Names can even be an industry-related name like '*medical*'. If Vim does not recognize the language name, it will be highlighted when you execute the property-setting command.



If you change the `spelllang` setting to a language not already installed, then Vim will ask you if it should try to retrieve it from the Vim homepage, automatically.

Personally, I tend to work in several different languages in Vim, and I really don't want to tell Vim all the time which language I am using right now.

Vim has a solution for this. By appending more language codes to the *spelllang* property (separated by commas), you can tell Vim to check the spelling in more than one language.

```
:set spelllang=en,da,de,it
```

Vim will then take the languages from the start to the end, and check if the words match any word in one of these languages. If they do, then they are not marked as a spelling error. Of course, this means that you can have a word spelled wrong in the language you are using but spelled correctly in another language, thereby introducing a hidden spelling error.

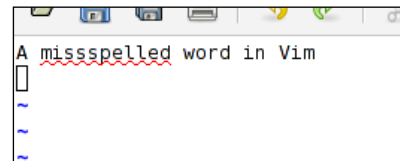
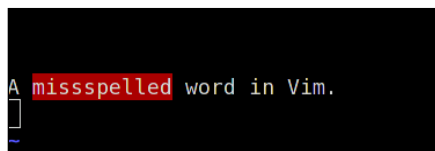


You can find language packages for a lot of languages at the Vim FTP site: <ftp://ftp.vim.org/pub/vim/runtime/spell>

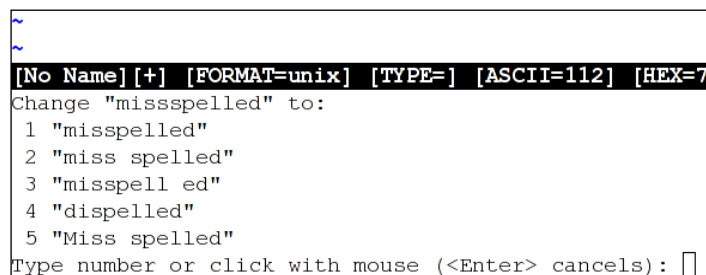
Spelling errors are marked differently in Vim and Gvim.

In regular Vim, the misspelled word is marked with the *SpellBad* color group (normally white on red).

In Gvim, the misspelled word is marked with a red curvy line underneath the word. This can of course be changed by changing the settings of the color group (See the section *Personal Highlighting* for more info).



Whenever you encounter a misspelled word, you can ask Vim to suggest better ways to spell the word. This is simply done by placing the cursor over the word and then going into the normal mode (press *Esc*), and then pressing *z=*.



Vim, if possible, will give you a list of good guesses for the word you were actually trying to write. In front of each suggestion is a number. Press the number you find in front of the right spelling (of the word you wanted) or *Enter* if the word is not there.

Often Vim gives you a long list of alternatives for your misspelled word, but unless you have spelled the word completely wrong, chances are that the correct word is within the top 5 of the alternatives. If this is the case, and you don't want to look through the entire list of alternatives, then you can limit the output with the following command:

```
:set spellsuggest=X
```

Set *x* to the number of alternative ways of spelling you want Vim to suggest.

## [Gvim]<sup>7+</sup> Adding Helpful Tool Tips

In the recipe *Modifying Tabs*, we learned about how to use tool tips to store more information in the tabs in Gvim, without taking up much space. To build on top of that same idea, with this recipe, we move on and use tool tips in other places in the editor.

The editing area is the largest part of Vim; why not try to add some extra information to the contents of this area by using tool tips?

In Vim, tool tips for the editing area are called **balloons** and they are only shown when the cursor is hovering over one of the characters. The commands you will need to know in order to use the balloons are:

```
:set ballooneval
:set balloondelay=400
:set ballonexpr="textstring"
```

The first command is the one you will use to actually turn on this functionality in Vim.

The second command tells Vim how long it should wait before showing the tool tip/balloon (the delay is in milliseconds and as a default is set to 600).

The last command is the one that actually sets the string that Vim will show in the balloon. This can either be a static text string or the return of some function. In order to have access to information about the place where you are hovering over a character in the editor, Vim gives access to a list of variables holding such information:

- `v:beval_bufnr` : Number of the buffer in which the hovered area is.
- `v:beval_winnr` : Number of the window in which the hovered area is shown.
- `v:beval_lnum` : Line number on which the hovered character is situated.
- `v:beval_col` : Number of the column in which the hovered character is.
- `v:beval_text` : Word to which the hovered character is connected.

So with these variables in hand, let's look at some examples.

### Example 1:

The first example is based on one from the Vim help system, and shows how to make a simple function that will show the info from all the available variables.

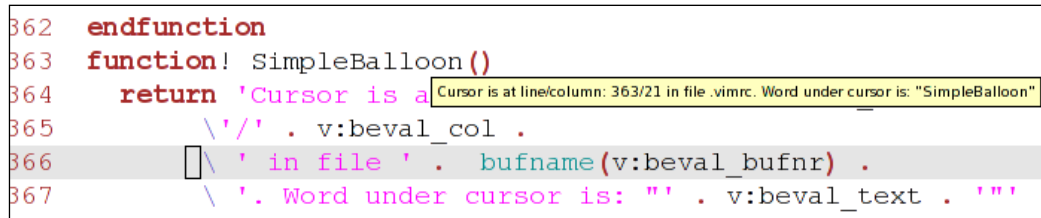
```
function! SimpleBalloon()
    return 'Cursor is at line/column: ' . v:beval_lnum .
```

```

\ '/' . v:beval_col .
\ ' in file ' . bufname(v:beval_bufnr) .
\ '. Word under cursor is: "' . v:beval_text . '"'
endfunction
set balloonexpr=SimpleBalloon()
set ballooneval

```

The result will look like in the following figure:



```

362 endfunction
363 function! SimpleBalloon()
364     return 'Cursor is at line/column: 363/21 in file .vimrc. Word under cursor is: "SimpleBalloon"'
365 \ '/' . v:beval_col .
366 \ ' in file ' . bufname(v:beval_bufnr) .
367 \ '. Word under cursor is: "' . v:beval_text . '"'

```

### Example 2:

Let's look at a more advanced example that explores the use of balloons for specific areas in editing. In this example, we will put together a function that gives us great information balloons for two areas at the same time:

- Misspelled words – the balloon gives ideas for alternative words.
- Folded text – the balloon gives a preview of what's in the fold.

So let's take a look at what the function should look for, to detect if the cursor is hovering over either a misspelled word, or a fold line (a single line representing multiple lines folded behind it).

In order to detect if a word is misspelled, the spell check would need to be turned on:

```
:set spell
```

If it is on, then calling the built-in spell checker function, `spellsuggest()`, would return alternative words if the hovered word was misspelled. So, to see if a word is misspelled is just to check if the `spellsuggest()` returns anything. There is, however, a small catch. `spellsuggest()` also returns alternative, similar words, if the word is not misspelled. To get around this, another function has to be used on the input word before putting it into the `spellsuggest()` function. This extra function is the `spellbadword()`. This basically moves the cursor to the first misspelled word in the sentence that it gets as input, and then returns the word. We just input a single word, and if it is not misspelled, then the function cannot return any words. Putting no word into `spellsuggest()` results in getting nothing back, so we can now check if a word is misspelled or not.

To check if a word is in a line, in a fold, is even simpler. You simply have to call the function `foldclosed()` on the line number of the line over which the cursor is hovering (remember `v:beval_lnum`?) and it will return the number of the first line in the current fold—if not in a fold, then it returns -1. In other words, if `foldclosed(v:beval_lnum)` returns anything but -1 and 0, we are in a fold.

Putting all of this detection together and adding functionality to construct the balloon text ends up as the following function:

```
function! FoldSpellBalloon()
  let foldStart = foldclosed(v:beval_lnum )
  let foldEnd   = foldclosedend(v:beval_lnum)
  let lines = []
  " Detect if we are in a fold
  if foldStart < 0
    " Detect if we are on a misspelled word
    let lines = spellsuggest( spellbadword(v:beval_text)[ 0 ], 5, 0 )
  else
    " we are in a fold
    let numLines = foldEnd - foldStart + 1
    " if we have too many lines in fold, show only the first 14
    " and the last 14 lines
    if ( numLines > 31 )
      let lines = getline( foldStart, foldStart + 14 )
      let lines += [ '-- Snipped ' . ( numLines - 30 ) . ' lines --' ]
      let lines += getline( foldEnd - 14, foldEnd )
    else
      "less than 30 lines, lets show all of them
      let lines = getline( foldStart, foldEnd )
    endif
  endif
  " return result
  return join( lines, has( "balloon_multiline" ) ? "\n" : " " )
endfunction

set balloonexpr=FoldSpellBalloon()
set ballooneval
```



The result is some really helpful balloons in the editing area of Vim that can improve your work-cycle tremendously. The following figure shows how the info balloon could look when using it to preview a folded range of lines from a file.

The screenshot shows a Vim editor window with a file named `vimrc`. Lines 268 to 296 are folded. An info balloon is displayed over the folded range, showing the following text:

```

268 +-- 8 lines: s:RestorePosn: this function resto
276 +-- 13 lines: CleanupSessionFile: if you exit Vi
289 " GotoWinNum:
290 fun! s:GotoWin
291 " call Dfunc(
292 if a:winnu
293 exe a:winnu
294 endif
295 " call Dret("
296 endfun
  
```

If the balloon is instead used on a misspelled word, it will look like this:

The screenshot shows a Vim editor window with a file named `vimrc`. Line 460 contains the text `Evenlines`, which is misspelled. A spelling correction balloon is displayed over the word, showing the following suggestions:

- Even lines
- Evelin's
- Evenings
- Eveline's
- Eventides



In Chapter 4, you can learn more about how to use folding of lines to boost productivity in Vim.

## [Vim]<sup>6+</sup> [GVim]<sup>6+</sup> Using Abbreviations

We all know the feeling of writing the same things over and over again, a dozen times during a day. This feeling is the exact opposite of that the philosophy of Vim tries to teach us.

The philosophy of Vim says that if you write a thing once, it is OK, but if you're writing it twice or more times, then you should find a better way to do it.

One of the methods for getting around writing the same sentences over and over again is by using **abbreviations**.

In Vim, abbreviations are created with one of the following commands depending on which mode they should be available in:

```
:abbreviate  : Abbreviations for all modes
:iabbrev      : Abbreviations for insert mode
:cabbrev      : Abbreviations for the command line only
```

All of the commands take two arguments: the abbreviation, and the full text it should expand to.

So let's start with a simple example of where the abbreviations can come in handy.

### Example 1:

I have moved around a bit during the last few years, so a common task for me is writing messages where I tell about my new address. It didn't take me long before I had an abbreviation ready, so I didn't have to write the entire address.

Here is what it looked like:

```
:iabbrev myAddr 32 Lincoln Road, Birmingham B27 6PA, United Kingdom
```

So now, every time I need to write my address, I just write **myAddr**, and as soon as I press *space* after the word, it expands to the entire address.

Vim is intelligent about detecting whether you are writing an abbreviation or it is just part of another word. This is why **myAddr** only expanded to the full address after I pressed *space* after the word. If the character right after my abbreviation was a normal alphabetic letter, then Vim would know that I wasn't trying to use my abbreviation and it would not expand the word. Examples with the abbreviation 'abc':

- **abc<space>** and **abc<enter>** : Both expand.
- **123abc<space>** : Will not expand since abbreviation is part of a word.
- **abcd<space>** : Will not expand because there are letters after the abbreviation.
- **abc** : Will not expand until another special letter is pressed.

A good place to keep your abbreviations, so that you don't have to execute all the commands by hand is in a file in your **VIMHOME**. Simply place a file there (let's call it `abbreviations.vim`) and write all your abbreviations in it. Then, in your `vimrc` file, you just make sure that the file is read, which is done with the source command:

```
:source $VIM/abbreviations.vim
```

Every time you realize that you will need a new abbreviation, you first execute it, and then you add it to your `abbreviations.vim`.

By now you have probably realized that you can use abbreviations for a lot of other interesting things. But anyway here is a short list of examples to give you some ideas:

- Correct typical keypress errors:  

```
:iabbr teh the
```
- Simple templates for programming:  

```
:iabbr forx for(x=0;x<100;x++) {<cr><cr>}
```
- Easy commands in the command line:  

```
:cabbr csn colorscheme night
```

Getting used to adding your abbreviations to a file every time you find a new one, might seem weird and inconvenient at first. At the end of the day, however, you will realize that it has saved you a lot of typing and will keep doing so, over and over again. The only thing you have to do is add your abbreviations, and reload the abbreviations file once in a while.

## [Vim]<sup>6+</sup> [GVim]<sup>6+</sup> **Modifying Key Bindings**

All of us have probably at some point used an editor other than Vim. Because of this most of us have learned to use some very specific keyboard shortcuts for doing different tasks.

Even though the key-bindings for the keyboard shortcuts in Vim are created with ease and speed of use in mind, it can still be faster sometimes to use the shortcuts you already know.

To facilitate this, Vim gives you the possibility to re-bind almost every single key binding it has.

In this recipe, we will learn how to change the key bindings when using Vim in different modes.

The main commands to know when dealing with key bindings are:

- `:map` for the modes **Normal**, **Insert**, **Visual** and **Command-line**
- `:imap` for **Insert** mode only
- `:cmap` for **Command-line** mode only
- `:nmap` for **Normal** mode only
- `:vmap` for **Visual** mode only

Each of the command takes two arguments—the first is what keys the command should be bound to, and the second is the command to bind. So let's look at an example.

Say you can't really get used to saving an open file by executing `:w` in the normal mode, because you are used to using *Ctrl-s* to save a file and would like to keep it like that.

A mapping for this could be:

```
:map <C-S> :w<cr>
```

Notice the `<C-S>` in the command. This is the Vim way for writing 'key combination *Ctrl-s*'. Instead of `C` for *Ctrl*, you could also use `A` for *Alt* or `M` for *Meta*. The `<cr>` at the end of the command is what actually executes the command. Without it, the command would simply be written to the command line but not executed.

But maybe you only want to be able to save when you are in insert mode, and actually editing the file. To change the command for this, you only need to have the following:

```
:imap <C-S> <esc>:w<cr>a
```

So what happens now, is that you map the *Ctrl-s* to do a combination of key presses. First, `<esc>` (the Escape key), to get out of insert mode and into normal mode. Then, `:w<cr>` to execute the actual saving of the file, and finally the `a`, to get back into insert mode and go to the end of the line.

You could expand the mappings to fit all of the standard copy/paste/cut/save shortcuts from many applications. This could be constructed like:

```
" save file (ctrl-s)
:map <C-S> :w<cr>

" copy selected text (ctrl-c)
:vmap <C-C> y

" Paste clipboard contents (ctrl-v)
:imap <C-p> <esc>P
```

```
" cut selected text (ctrl-x)
:vmmap <C-x> x
```

If you are in Gvim, you can even get dialogs shown for the **Save-as** and **Open** functionalities.

```
"Open new file dialog (ctrl-n)
:map <C-n> :browse confirm e<cr>
"Open save-as dialog (ctrl-shift-s)
:map <C-S-s> :browse confirm saveas<cr>
```

With the ability to change the keyboard mapping in Vim, you really have access to a powerful way of modifying the editor completely according to your needs.



You can read more about mappings in the vim help system under:  
**:help key-mapping**

## Summary

In this chapter, we have looked at how to make Vim a better editor for you by modifying it to your personal needs.

We started out by learning about how basic modifications of font and color scheme can give you editor a personalized look.

Then we dived a bit deeper into using colors for marking search matches, thereby making them easily recognizable.

To get the most out of an editor like Vim, you would often like it to have a large area for editing the files, and less space spilled on GUI. We looked at ways of modifying both the status line and tabs to be smaller and more informative. If you don't want the menu and toolbar at all, you have also been shown a way for toggling its visibility.

Even though the menu and toolbar can be in the way, they can also be very usable additions to your editor. In this chapter, we have learned how to add our own menu to the menu bar and even how to add icons full of functionality to the toolbar.

Many things can be done to the editing area to make it fit your personal needs. In this chapter we have looked at how to make it easier to get an overview of the editing area. Better and more visual cursors have been proposed and line numbers have been added to the area.

If you need help with your spelling, then Vim has methods for helping you there. We have looked at how to make the spell checker in Vim follow your preferred language, so that you will never again misspell a word. If using spell-checking is not enough to correct your errors, then maybe the use of abbreviations can help you. On the other hand, abbreviations also do a great job minimizing the number of characters to write if you use the same text over and over again.

Finally, we have looked at how we can change the key bindings in Vim in such a way that it will react on keyboard shortcuts you are used to from other editors.

With all the recipes in this chapter, you should have a fully personalized Vim editor, and you are now ready to move on and learn more about how you can optimize your navigation around the files in Vim.

## Where to buy this book

You can buy Hacking Vim: A Cookbook to get the Most out of the Latest Vim Editor from the Packt Publishing website: <http://www.packtpub.com/Vim/book>.

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



[www.PacktPub.com](http://www.PacktPub.com)

For More Information: <a href="http://www.PacktPub.com/Vim/book">www.PacktPub.com/Vim/book</a>
--