From Technologies to Solutions

Hacking Vim

A cookbook to get the most out of the latest Vim editor

From personalizing Vim to productivity optimizations:
Recipes to make life easier for experienced Vim users

Kim Schulz

PACKT PUBLISHING

Hacking Vim

A cookbook to get the most out of the latest Vim editor

From personalizing Vim to productivity optimizations: Recipes to make life easier for experienced Vim users

Kim Schulz

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Hacking Vim

A cookbook to get the most out of the latest Vim editor

First published: May 2007

Production Reference: 1140507

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847190-93-2

www.packtpub.com

Cover Image by www.visionwt.com

Credits

Author

Kim Schulz

Reviewers

Brian Jørgensen

James Eaton-Lee

Kenneth Geisshirt

Sven Guckes

Development Editor

Nanda Nag

Nikhil Bangera

Technical Editor

Ajay S

Editorial Manager

Dipali Chittar

Project Manager

Patricia Weir

Project Coordinator

Sagara Naik

Indexer

Bhushan Pangaonkar

Proofreader

Chris Smith

Layouts and Illustrations

Shantanu Zagade

Cover Designer

Shantanu Zagade

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 Fundanemt. This has lead to him now owning the web-hosting company Devteam Denmark that specializes in hosting and development of Fundanemt-based websites.

I would like to thank my girlfriend, Line, for letting me take the time to write this book. Without her positive attitude and help, I would never have got this book ready.

I would also like to add a great thank you to Bram Moolenaar, for developing the Vim editor, and making it what it is today.

About the Reviewers

Brian Jørgensen was born in 1982 in Northern Jutland, Denmark. His early interest in computers and programming on the Commodore 64 and later the PC resulted in him studying computer science at Aalborg University, from where he is about to graduate with a masters degree in software engineering. In the late 90s he found a great interest in the Linux operating system, which he has been using since. When he is not studying or working as a freelance developer, his spare time is spent working on Open Source software projects. He is one of the core developers on the Fundanemt CMS. His main interests are in programming languages, Internet technologies, and keeping a tap on a wide range Open Source projects. Brian has a personal website (`http://qte.dk/`) where he runs a blog and writes about his software projects.

James Eaton-Lee works as a Consultant specializing in Infrastructure Security and has worked with clients ranging from small businesses with a handful of employees to multinational banks. He has a varied background, including experience working with IT in ISPs, manufacturing firms, and call centers. James has been involved in the integration of a range of systems, from analogue and VOIP telephony to NT and AD domains in mission-critical environments with thousands of hosts, as well as Unix and Linux servers in a variety of roles. James is a strong advocate of the use of appropriate technology, and the need to make technology more approachable and flexible for businesses of all sizes, but especially in the SME marketplace in which technology is often forgotten and avoided. James has been a strong believer in the relevancy and merit of Open Source and Free Software for a number of years and —wherever appropriate—uses it for himself and his clients, integrating it fluidly with other technologies.

Kenneth Geisshirt is a chemist by education, and is a strong free-software advocate. He spent his Christmas holidays in 1992 installing SLS Linux, and GNU/Linux has been his favorite operating system ever since. Currently, he does consultancy work in areas like scientific computing and Linux clusters. He lives in Copenhagen, Denmark with his partner and their two children. You can find him at `http://kenneth.geisshirt.dk/`.

Table of Contents

Preface

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.

Good luck and happy reading!

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

What You Need for This Book

Over the course of the last decade, Vim has evolved into a feature-rich editor. This means that the some of the features from the later versions of Vim are not accessible in the older versions of Vim.

Vim is available for a wide variety of platforms and not all recipes might work on all platforms. This is typically due to the use of system-specific functionality that is not available on other platforms.

This book will focus on two of the platforms where Vim is most widespread, namely Linux and Microsoft Windows. As the Linux system resembles the system used in most Unix platforms, the recipes will work onother *NIX platforms.

You can find the latest source code and binary packages for the Vim Editor at `www.vim.org`.
If you use Linux it is, however, most likely that Vim is already packed with your Linux distribution as it is the default editor on most Linux systems.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

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

Any command-line input and output is written as follows:

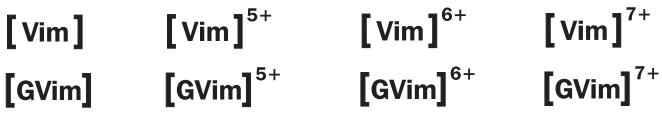
```
:amenu icon=/path/to/icon/myicon.png ToolBar.Bufferlist :buffers<cr>
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

This book primarily focuses on the features available in Vim version 7+, but since some of the recipes cover tasks available in earlier versions, each recipe will be marked with one of the following icons that specify the version for which the recipe is applicable:



Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/support`, and select this book from the list of titles to download any example code or extra resources for this book. The files available for download will then be displayed.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Introduction

The Vim editor (or Vi IMproved) was first released by Bram Moolenaar in November 1991 as a clone of the Unix editor vi for the Amiga platform.

The first release of Vim for the Unix platform was out a year later and right away, it started to become an alternative to the vi editor.

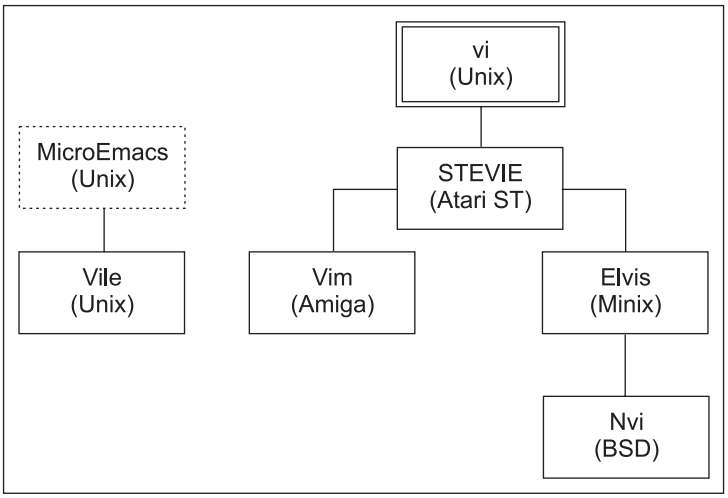
The combination of a more liberal licensing model and the fact that Vim started to become a superset of vi's functionality resulted in it becoming progressively more popular with the Open Source community. Soon more and more Linux distributions started to adopt Vim as an alternative to vi. Even if the users thought they used vi (if they actually executed the **vi** command) they opened Vim (the **vi** command had simply been substituted with a link to the **vim** command, which has often lead to the misunderstanding that vi and Vim are actually the same program).

During the late 90s, Vim took over where vi was lacking behind in the so-called editor-war that existed between the vi editor and the Emacs editor. Bram implemented a lot of the missing features that the Emacs community used as arguments for why Emacs was better than vi/Vim, but he did it without ever neglecting the main focus areas that the vi editor had had, right from the beginning.

Today, Vim is a feature-rich, fully configurable editor loved by many. It supports syntax-highlighting of more than 200 different programming languages, auto-completion for a fast growing number of languages, folding, undo/redo, multiple buffers/windows/tabs, and a lot of other features.

Vi, Vim, and Friends

Vim is just one of many derivatives of the original vi that Bill Joy released back in 1976. Some have a feature list very close to that of vi, while others have chosen to add a wide variety of new features. Vim belongs to the group of vi clones that has chosen to add extra features. In the next section, we will introduce some of the better-known clones of vi and briefly describe the distinct features that each clone has.



vi

Vi is the original root of the Vim family tree. It was created by Bill Joy in 1976 for one of the earlier versions of BSD (Berkeley Software Distribution). The editor was an extension of the most common editor at that time, **ex**. Ex was, in turn, an extension of the Unix editor '**ed**'. The name 'vi' is actually an abbreviation of '**visual in ex**'. As the name indicates, vi was actually just a command that started the ex editor in one of its modes—the visual mode.

Vi was one of the first editors to introduce the concept of modality. What this means is that the editor has different modes for different tasks—one mode for editing text, another for selecting text, and yet another for executing commands.

This modality is one of the main features in vi that makes enthusiasts like the editor, but it is also what makes others dislike it even more.

Not much has changed in **vi** since the first version, but it is still one of the most used editors in the Unix community. This is mainly because **vi** is considered a required application for a Unix to comply with the **Single Unix Specification** (SUS)—and hereby be able to call itself a Unix.

STEVIE

In 1987, Tim Thompson got his first Atari ST. In this platform, there weren't any really good editors so he decided to clone the editor **vi**, which was known from the Unix platform. In June 1987, he released an editor under a license that resembles what has later become known as open-source. He released it on Usenet and named it **STEVIE**—an abbreviation for '**ST Editor for VI Enthusiasts**'.

It was very simple and only provided a very small subset of the functionality that vi provided. It did, however, provide a familiar environment for vi users moving to the ST.

After the release, Tim Thompson discontinued work on the editor. But soon Tony Andrews took over, and within a year he had ported it to Unix and OS/2. More features were added along the way but at some point around 1990, the development stopped.

STEVIE as an editor might not have survived throughout the years, but since both Tim and Tony released the source code on Usenet as public-domain for anyone to use, a lot of the later vi clones have been both inspired and based on this code.

Elvis

STEVIE was one of the more common editors around. It was, however, full of bugs and had some quite unpractical limitations. Steve Kirkendall, who at that time used the operating system Minix, noticed one very big limitation, i.e. the STEVIE editor held the entire file in memory while editing. This was not an optimal solution when using Minix, so Steven decided to rewrite the editor to use a file as buffer instead of editing in RAM. This turned into **Elvis**, version 1.0.

Even though Elvis was an improvement over the vi editor, it still suffered from some of the same limitations that vi had—max length of lines and only a single file buffer.

Steve Kirkendall decided to rewrite Elvis completely to get rid of the limitations, and this turned into Elvis version 2, which is the generation of the editor currently available (version 2.2).

With generation 2 of Elvis, Steve also included support for a range of other features that weren't in the original vi editor. Among these, a few features that are interesting and worth mentioning are:

- Syntax highlighting
- Multiple windows support
- Networking support (HTTP and FTP)
- Simple GUI front ends

Elvis is not actively developed anymore, but is still widely used. It is available for Unix, MS Windows (console or GUI with WinElvis), and OS/2.

The latest version of the Elvis editor can always be found here:
`http://elvis.the-little-red-haired-girl.org/`

Nvi

Nvi, or new vi (as its full name is) is a result of a license dispute between AT&T and the Computer Science Research Group (CSRG) at University of California, Berkeley. Vi was based on an original code from the editor ed, which was under the AT&T System V Unix license, so it was not possible for CSRG to distribute vi with BSD.

CSRG decided to replace the vi editor with an alternative editor under a freer license—their own BSD license.

Keith Bostic was the man that took on the job to make the new vi. The vi clone Elvis was already freely available, but Keith wanted an editor that resembled the original vi editor even more. He took the code for Elvis and transformed it into an almost 100% vi compatible clone—the nvi editor. Only the **Open Mode** and the **lisp edit** option from the original vi functionality set is left out.

By the release of 4.4BSD, the vi editor was completely substituted by nvi, and the software distribution was once again completely covered by a free license.

Today nvi is the default vi editor in most BSD-derived distributions like NetBSD, FreeBSD, and OpenBSD, and has evolved into a more feature-rich editor than the original vi.

Compared to the original vi editor, nvi has been extended to support new features like:

- Multiple edit buffers
- Unlimited Undo
- Extended Regular Expressions
- CScope support
- Primitive scripting support in Perl and Tcl/Tk

Keith Bostic is still the maintainer of the nvi source code, but not much development has been done to the code for some time now.

The latest version of the nvi editor can always be found here:
`http://www.bostic.com/vi/`

Vim

The editor **Vim** is the golden child of the vi family. Ever since Bram Moolenaar released the first version of Vim to the public in November 1991, this editor has evolved into one of the most feature-rich editors around.

The first version of Vim was, like the Elvis editor, based on the source code of the editor Stevie. Bram, however, released Vim only for the Amiga platform, which was one of the most widespread platforms, at that time, among home computer enthusiasts. At that time Vim was an abbeviation for **Vi-IMitation**, which described Vim quite well in that it simply tried to do what vi did.

A year later, in 1992, however, Bram made a port of his Vim editor for the Unix platform. The result of this was that Vim went beyond simply being a clone of vi for a different platform, to becoming a competitor. The development of Vim was quick and fast, and soon Vim had a range of features that the original vi editor did not have. Because of this, the abbreviation Vim was at some point changed into being Vi-IMproved instead of Vi-IMitation.

Within a couple of years, Vim grew to having a range of features that a lot of vi users missed. This made more and more users switch over to using Vim instead of vi as their primary editor.

In 1998, the fifth generation of Vim was released, and with it one of the most used features of today, scripting, was introduced.

Now, it was possible for the user to write their own scripts for Vim, and in that way expand the functionality of the editor. This was a really strong addition to the feature set of Vim, because it would normally have required coding in a lower-level language and recompilation of the editor in order to add even simple features.

A lot of features have been added to Vim throughout the last decade, and many of these are quite unique compared to the other editors and vi clones in particular.

Here we will list just a few of the more distinct features of Vim, since the complete feature list would be too long:

- Editing multiple files in multiple buffers, windows, and tabs
- Advanced Scripting language
- Support for scripting in Perl and Python

- Syntax highlighting for 200+ programming languages
- Unlimited undo/redo with branching
- Context-aware completion of words and functions
- Advanced pattern-matching with Regular Expressions
- Close integration with a wide range of compilers, interpreters, and debuggers
- More than 1500 Vim scripts freely available online

Vim is available for an enormous variety of platforms like all types of Unix, Linux, MS Dos, MS Windows, AmigaOS, Atari MiNT, OS/2, OS/390, MacOS, OpenVMS, RISC OS, and QNX.

Vile

Vile is maybe the vi clone that looks least like the original vi editor—some would even say that it's not a clone at all. Vile is actually an attempt to bring the best of two worlds together in one editor: the modality of vi and the feature set of Emacs.

This also explains the name Vile, which is short for "VI Like Emacs."

The Vile editor project was started by Paul Fox during the summer of 1990. The code was based on the core code from the public-domain editor **MicroEmacs**. Paul then modified it to have modality and other vi-like features.

The MicroEmacs code did not have all the features of the Emacs editor, but it had support for long lines and editing multiple files in multiple windows at the same time. These were features that vi did not have and which many programmers needed in their editor.

A lot of work was done to get the MicroEmacs code to be more vi-like, and several other developers joined the project. Thomas E. Dickey joined the project in 1992 and added a wide variety of features to Vile and fixed a lot of bugs in the code.

In 1994, Kevin Buettner joined the project and started working on the GUI version of vile— **xvile**. He added support for some of the most common widget sets at that time, like Athena, OpenLook, Motif, and the Xt Toolkit.

Today Thomas is the primary maintainer of Vile and the development is steered by him. His time for working on the editor is, however, very limited. So, it is mostly only bugfixes that he adds to the editor.

Vi and Vile are not very similar in the way they work, and only a minor subset of the vi features are present in Vile. The main feratures of Vile are:

- Editing modes—one mode for each file type
- Vile procedure language for macros
- (Experimental) Perl Support
- Named functions that can be bound to keys as the user wishes

Vile is available for Unix, Linux, BeOS, OS/2, VMS, and MS Windows and exists in both a console version and a GUI version.

The latest version of the vile editor can always be found here:
`http://www.vile.cx/`

Compatibility

Though all the vi clones have at some point tried to behave like the vi editor, most of them have evolved in very different directions. This means that even though a lot of them support features such as syntax highlighting, they do not necessarily implement them in the same way. Therefore A syntax file from Vim cannot be used in Elvis.

Even the features that originate from vi are not necessarily implemented the same way. Some of the clones have implemented features less accurately than others. Maybe the idea behind the feature is the same, but the actual result of using it is completely different.

In the following table, I have tried to give a percentage of how accurately the mentioned clones resemble the vi editor (0% being least compatible and 100% being completely compatible). The comparison has been done by looking at how much effort the clone developers have made in order to implement the features of vi as precisely as possible.

Clone	vi compatibility	Comment
STEVIE	10%	Only a very small feature set in common.
Vile	10%	Only general concepts like modes in common.
Elvis	80%	Large feature set in common, some features behave quite differently though.
Nvi	95%	Nearly perfect compatibility, but a range of the features behave differently.
Vim	99%	In the 'compatible mode' nearly all features are compatible.

In the table, only the features that the clones share with vi are considered. This means that even though for example, Vim has a lot of features that vi does not have, it still resembles vi very precisely on the features that they share. Besides this, Vim implements nearly all of the features that vi has. Only some of the features that Bram Moolenaar considered as bugs in vi are implemented differently in Vim. Note that in order to make Vim 99% compatible with vi, you will have to set it into compatible mode with the command:

```
:set compatible
```

In Vim you can read more about vi and Vim differences with the command: **`:help vi-differences`**.

Another interesting observation is that even though STEVIE implemented a subset of the vi functionality very accurately, it did not implement enough of the vi features to be considered a close relative.

Vim is Charityware

Bram Moolenaar, the developer of the Vim editor, has chosen to release Vim under a so-called charityware license. What this means is that you can copy Vim as much as you like, but in exchange you are encourage to make donations to a charity.

You can read more about the project if you open Vim and execute the command:

```
:help uganda
```

You can also get more information about how you can sponsor the Vim project if you go to the website `http://www.vim.org/sponsor/`.

As a Vim sponsor, you will get to vote for new features that should be implemented in Vim. So besides supporting a good cause, you will also get some say on how Vim will evolve in the future.

Summary

In this chapter, we introduced Vim and looked at what this book is about. Vim is just one of many clones of the old Unix editor vi, so to get a broader view of the vi-family tree, this chapter introduced some of the more well-known clones. Their history and relation to vi were briefly described and we learned that even though the vi clones at some point have tried to be like vi, they are not really compatible with each other.

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

Some of these tasks contain more that 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]$^{6+}$ 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 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]$^{5+}$ 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:

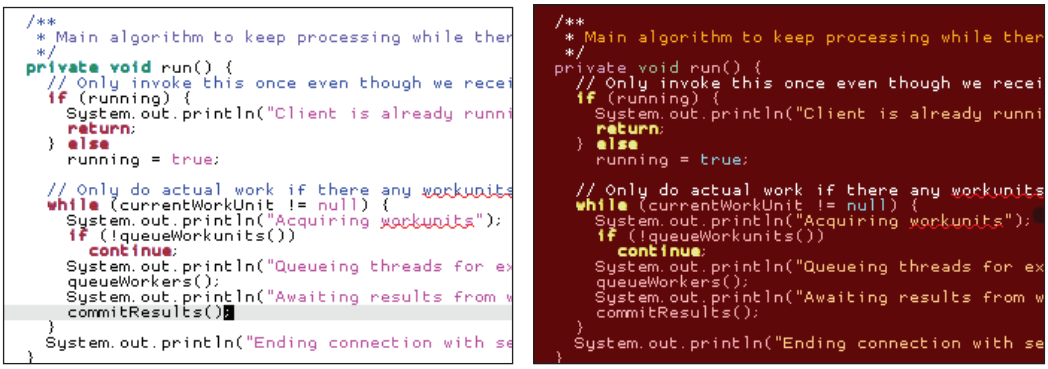
```
:colorscheme
```

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.



[Vim]$^{6+}$ [GVim]$^{6+}$ 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.

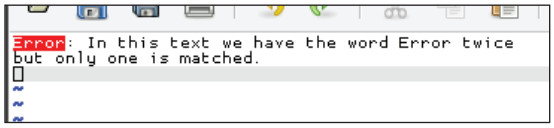
Compared to a color scheme, which affects the entire color setup, a color group is a rather small combination of background (or foreground) colors that you can use for things like matches. When Vim is started, a wide range of color groups are set to default colors, depending on the color scheme you have 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 ErrorMsg /\%>73v.\+/
```

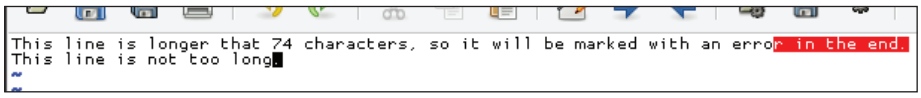
Here, every character after the 73rd character will be marked as an error. This match is a regular expression that when broken down consists of:

`\%>` : 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 errorMsg /[^\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

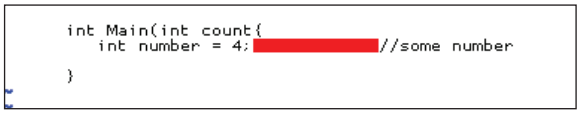
`\t` : The tab-character

`]` : 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).



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]$^{6+}$ [GVim]$^{6+}$ 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]$^{6+}$ 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]$^{7+}$ Adding Your Own Menu and Toolbar Buttons

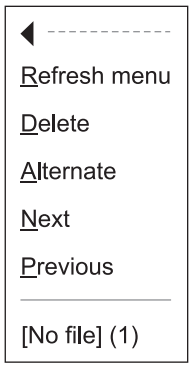
If you are in Gvim, you can make a handy menu with all the functionality you use the most. You might not always need to use it from the menu, but whenever you forget how to use it, you can always just find it there. If you need to get to the functionality really fast, you can even add it directly in 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 **Normal** mode.
:imenu – for **Insert** mode. **^O** is prepended.
:vmenu – for **Visual** mode. **^C** is prepended and **^\^G** is appended.
:cmenu – for **Command-line** mode. **^C** is prepended and **^\^G** is appended.
:omenu – for **OP-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]$^{6+}$ 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.

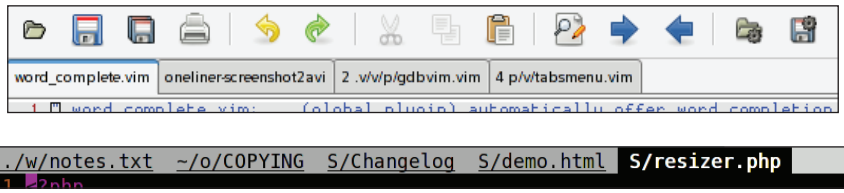
If you want your menu or toolbar icon placed elsewhere than to the right of the others, then you can use priorities. Read more about how in: **`:help menu-priority`** and **`:help sub-menu-priority`**

[Vim]$^{7+}$ 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).

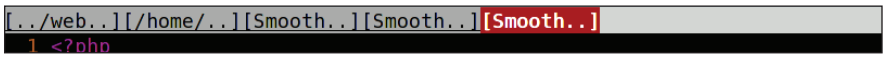
```
function ShortTabLine()
  let ret = ''
  for i in range(tabpagenr('$'))
    " select the color group for highlighting active tab
      if i + 1 == tabpagenr()
      let ret .= '%#errorMsg#'
    else
```

```
            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
        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:

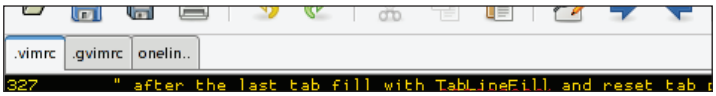
```
function ShortTabLabel()
   let bufnrlist = tabpagebuflist(v:lnum)
```

```
" 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:

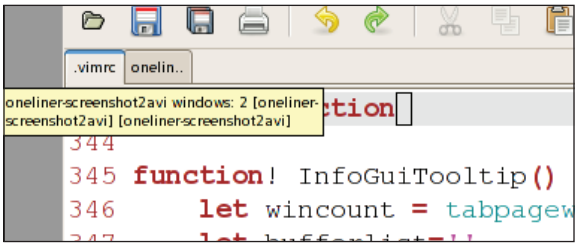
```
:set guitabtooltip=%!bufname($)
```

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]$^{7+}$ [GVim]$^{7+}$ 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.

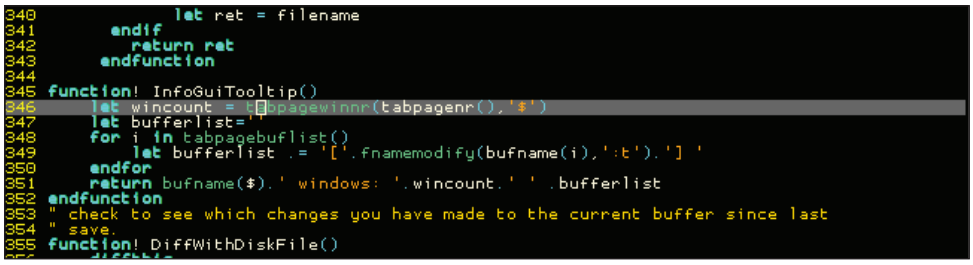
The first one is the **cursorline** command, which basically marks the entire line with, for example, another background color, without breaking the syntax coloring. To turn it on, use the following command:

```
: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.



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
345 function! InfoGuiTooltip()
346     let wincount = tabpagewinnr(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()
```

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]$^{6+}$ [GVim]$^{6+}$ 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]$^{7+}$ [GVim]$^{7+}$ 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
```

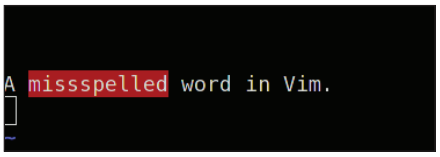
Vim will then take the languages from the start to the end, and check if the words match any word in one of these languages. If they do, then they are not marked as a spelling error. Of course, this means that you can have a word spelled wrong in the language you are using but spelled correctly in another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

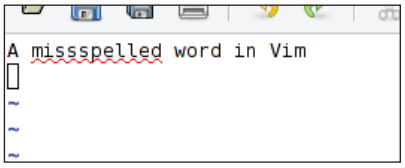
You can find language packages for a lot of languages at the Vim FTP site: `ftp://ftp.vim.org/pub/vim/runtime/spell`

Spelling errors are marked differently in Vim and Gvim.

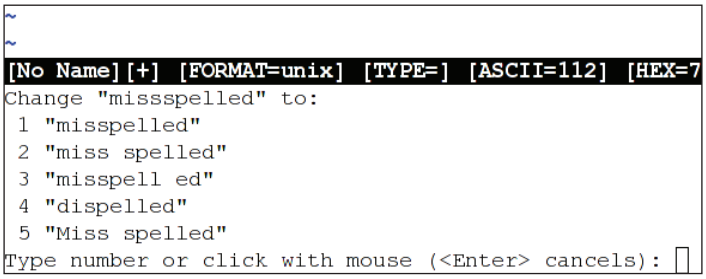
In regular Vim, the misspelled word is marked with the *SpellBad* color group (normally white on red).

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]$^{7+}$ 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 balooneval
: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.

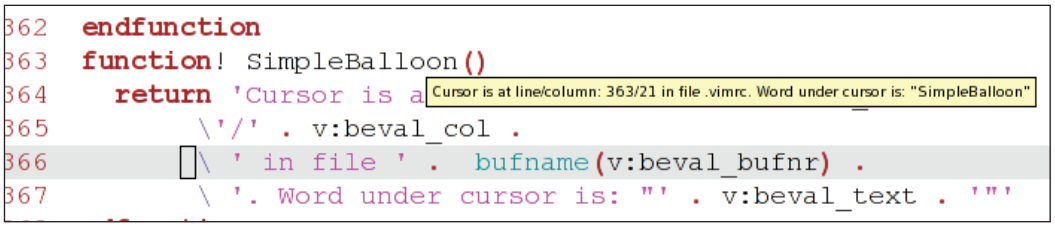
Example 1:

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:



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
```

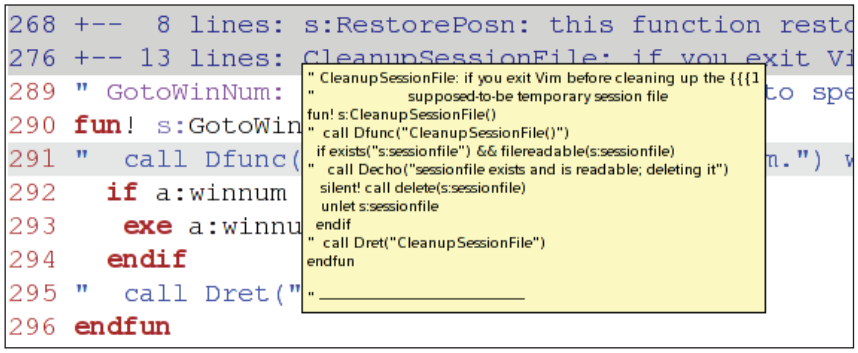
If it is on, then calling the built-in spell checker function, `spellsuggest()`, would return alternative words if the hovered word was misspelled. So, to see if a word is misspelled is just to check if the `spellsuggest()` returns anything. There is, however, a small catch. `spellsuggest()` also returns 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.

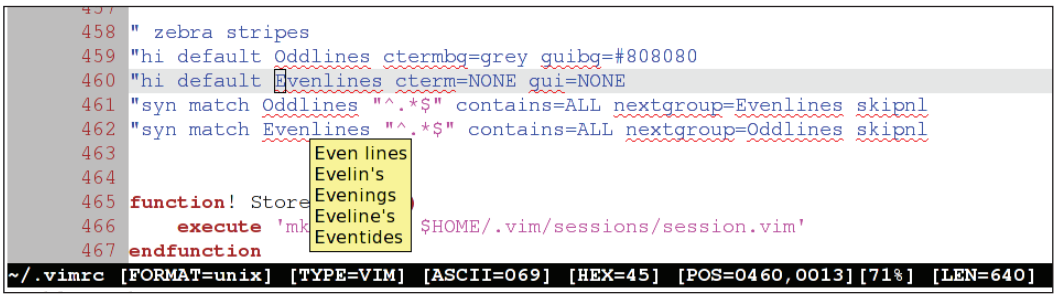
Putting all of this detection together and adding functionality to construct the balloon text ends up as the following function:

```
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.



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



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

[Vim]$^{6+}$ [GVim]$^{6+}$ 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]$^{6+}$ [GVim]$^{6+}$ 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)
   :vmap <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.

3
Better Navigation

Working with large files, or many files at the same time, can be a troublesome task. Sometimes, you realize that you waste more time looking for content to edit than doing the actual editing.

The philosophy of Vim is all about not wasting our valuable time, so Vim has means for optimizing the way we navigate files.

In this chapter, we will look at some of the ways in which Vim helps us easily navigate through our files, whether we're dealing with one file or fifty. Some recipes use marks to mark a spot for later return, while other uses search techniques to find the place you are looking for.

The recipes in this chapter cover the following areas:

1. Faster navigation in a file
2. Faster navigation in the Vim help system
3. Faster navigation in multiple buffers
4. Faster lookup of files using the Vim file explorer
5. In-file searching
6. Searching in multiple files or buffers with **vimgrep**
7. Using marks as a tool for navigation
8. Using signs as a tool to get better overview

After reading this chapter, you should be able to boost your navigation speed, and have no problems finding the files you are looking for.

Faster Navigation in a File

Sometimes even the simplest of tasks like navigating through a single file can be optimized. Vim offers several methods of navigation within a file, which can adapt to the contents of the file and how it is organized. Some of these methods are obvious, while others are more complex.

[Vim]$^{6+}$ [GVim]$^{6+}$ Context-Aware Navigation

Mostly, the files we are editing are well structured. If our files are text, then this structure can be in the form of paragraphs, sentences, and words, or at other times code with functions, blocks, and code lines.

Vim supports jumping around the file, according to the structure in the file, and has key bindings that make it easy to go to the exact place in the file where you want to go.

Let's look at some examples:

Example 1—Moving within a text file:

You are working on a normal text file and in the middle of a sentence you realize that you have forgotten to make the first letter in the paragraphs uppercase. You could of course, use the arrow-keys or the h/j/k/l navigation keys to move to the beginning of the paragraph to correct this. It would, however, be faster to just press the following in normal mode:

```
{
```

You are now placed at the beginning of the paragraph—or in the empty line just above it (if one exists). Now, you go into normal mode by pressing the *Esc* key and then use the movement command `{` to go to the beginning of the paragraph. The same can be applied if you want to go back to the end of the paragraph—you simply have to use the counterpart of `{`, namely:

```
}
```

Maybe you were not actually working at the end of the paragraph, but rather on correcting some text in the middle of the paragraph. Vim remembers where you were making changes to the file previously (and actually up to 999 of the last places you have changed something) and you can just ask it to take you back to the correct place. Just use the following command in normal mode:

```
g,
```

Pressing this command several times in a row will loop you through locations of previous changes in the file. As with the `{` command, this command also has a counterpart that moves forward through the list of recent locations where changes have taken place. The command for this is:

```
g;
```

Vim will alert you if you get to one of the ends of the list of changes.

Maybe it wasn't at the beginning of the paragraph that you had forgotten to capitalize a letter, but rather at the beginning of the current sentence. Again, Vim helps you move faster and offers you a pair of commands to move to the beginning and end of the current sentence. The commands are as follows:

`(` : Move to the beginning of the sentence.

`)` : Move to the end of the sentence.

Vim doesn't want us to waste any time when working in it. Even though you could easily go through the letters of a word by simply using the arrow keys, Vim still thinks this is waste of key-presses and instead offers a set of commands for word movement:

`w` : Move to the beginning of the next word.

`b` : Move to the beginning of the previous word.

`e` : Move to the end of the word.

These commands can be combined such that if you want to go to the end of the next word you simply press:

```
we
```

When it comes to what a word actually consists of, Vim has two definitions. In Vim we have:

- A *word* consisting of alphabetic letters, numbers, dashes, and underscores
- A *WORD* consisting of any character except white spaces (tab and space)

The above-mentioned movement commands work on *word*, but of course Vim has the same commands available for *WORD*. Simply use the same commands, but use them in uppercase instead (e.g., `W` to go to the beginning of the next *WORD*).

If you want to execute one of the commands mentioned in this section more than once in a row, simply add the number of times you want it executed in front of the command. For example, `5g`, to go to the place you changed something 5 changes ago.

Example 2 – Moving in a Code file:

Compared to text files, code does not have any paragraphs or sentences to navigate through. It does, however, frequently contain a lot of structures and blocks, each of which has a very specific contextual meaning within the code. An example could be the simple code block:

```
If( a == b) {
   print "a and b are the same";
}
```

Here the line with `print` is within the context of the `if` block surrounding it.

Because Vim is the favorite editor of many programmers, it offers a lot of movement commands to use when you are working with code. Common for all of them is that the parts of the code you want to jump between need to have a contextual connection to each other.

A simple example could be a construction like the `#if-#else-#endif` construction from the C programming language. Here we have a beginning (`#if`), an end (`#endif`) and a midpoint (`#else`).

So if you are in the line with `#if` and press the following command:

```
%
```

you will go to the `#else` part. Pressing it again, will take you to the `#endif`, and yet another execution of the command will get you back to the `#if`.

Vim does not know all programming language constructs, but by default it knows most of the contextual constructs of the C programming language. Besides this, it knows the normal block construction methods from most programming languages—the use of parentheses and brackets (for example, '{' is block start and '}' is block end).

If you want Vim to know the constructs of many other programming languages, then install the **MatchIt** plugin. This plugin is available with Vim as of version 7.0, but can also be found on:
`http://www.vim.org/scripts/`

Simply by knowing the programmer's common use of parentheses/brackets, Vim can provide us with several useful navigation commands. This means that as long as the code uses some start parenthesis/bracket to begin a block, and the counterpart to end it, Vim will understand it.

Let's say you are in a function that consists of many lines and you want to go to the beginning of the function. Mostly the brackets surrounding the contents of a function are the outermost pair of brackets around where you currently are in the file (given that you are editing the current function). So for Vim to find the beginning of the function, it simply has to find the outermost bracket pair and then go to the opening bracket.

```
function myExample() {
   ...many lines of code...
     /* cursor is placed at the beginning of this line */
   ...many lines of code...
}
```

In the above example, the `%` command would take us to the closing bracket and pressing it again would take us to the opening bracket. But what if the cursor was actually placed inside another pair of brackets? In that case, the `%` command would only move the cursor to them, and not the beginning of the function.

Again Vim has some handy commands for you:

`[[` and `]]` : Move backwards/forward to the next section beginning (e.g., start of a function)

`[]` and `]]` : Move backwards/forward to the next section end (e.g., end of a function)

Executing these commands multiple times in a row takes you to the beginning/end of the next/previous section, and therefore gives you a convenient way of cycling through the functions in a file.

Note that in most object-oriented languages, the Class beginning/end is often the outermost section.

Often, you just want to go to the beginning of the current block (e.g., the beginning of a while-loop) because it is here that you have defined all the local variables for the scope of this block. For this also, Vim has a set of movement commands:

`[{` : Move to the beginning of the block

`]}` : Move to the end of the block

If the block in the code is a comment, then it does not have any brackets around it; hence Vim cannot use the brackets to navigate to its beginning/end.

Therefore Vim has some special movement commands for comment blocks:

`[/` : Move to the beginning of the comment block

`]/` : Move to the end of the comment block

By default, not all comment formats are supported by Vim. It supports the comment formats used in the C programming language (`/* */`), C++ (`//`) and in many scripting languages (`#`). It is, however, possible to add support for extra comment formats when you add support for the syntax of new programming languages.

Sometimes, when you work on a piece of code, you tend to forget how a variable is actually defined. Vim has a command that can help you look up the definition of the variable (or the first occurrence of it, in the case of interpreted languages like Python) if it is defined in the current file. The command is as follows, and should be pressed while the cursor is placed on the variable name you want to look up:

`gd`

This command is easy to remember if you just think of the phrase 'Goto Declaration' and take the first letters from the two words.

What this command actually does is start by going to the beginning of the current section (remember the `[[` command) because this is where the local definitions are normally placed. Then, it makes a search forward in the file for the first occurrence of the variable name. If it does not find it before reaching the place where you started the lookup, then it moves to the first line of the file and again searches forward in the file looking for any global definitions of the variable. If it still does not find the definition, then Vim does a `*` search for the variable in the file (read more about the `*` search in the section *Search and You Will Find*).

If you know that the variable is globally defined, or if you want the global definition of the variable, then Vim has a command that starts by looking from line 1 of the file, instead of looking in the current section first. The command is:

`gD`

Vim is naturally smart enough to ignore any references to the variable in comment blocks because these are definitely not the declaration of the variable.

If Vim finds the variable definition (or the first available usage of the variable in the file), then it moves the cursor to this place.

Put a `1` in front of the `gd` command (like `1gd`) if you want Vim to ignore all matches that are inside a `{ }` block that ends before the current cursor position (for example, in another function block earlier in the file).

Navigating Long Lines

Some like their long lines visually wrapped in Vim while others want them to extend beyond the border of the editor (i.e., not shown). Personally, I like to have my text lines wrapped because it makes the overview of the text a lot better. This does, however, introduce an irritating problem. If you have a long line and it is wrapped, then the wrapped part shows up as a new line in the visualization of the file contents. This is not a problem in itself, but navigating to the wrapped part of the line is. If you use the arrow keys, or j/k to navigate between the lines, then it simply ignores the wrapped part of the line and goes directly to the next actual line in the file. If you dislike this behavior, then here's a short little recipe to fix this problem.

I have chosen that if I hold down the *Alt* key, while I use the *up/down* arrows to navigate the lines in the file, then Vim should follow the lines as shown visually in Vim, and not as the actual lines in the file. The key mappings to make this work are as follows and should simply be added to your `vimrc` file:

```
map <A-DOWN> gj
map <A-UP> gk
imap <A-UP> <ESC>gki
imap <A-DOWN> <ESC>gji
```

The mapping works in the normal mode and insert mode. If you want this to happen without having to remember to hold down the *Alt* key, then simply remove the `A-` part of the key combination to which the commands are mapped (e.g., `map <DOWN> gj`).

[Vim]$^{6+}$ [GVim]$^{6+}$ Faster Navigation in Vim Help

Vim comes with a very useful and comprehensive help system that you have probably by now already played around with. What you might not know, however, is that the help system comes with hyperlink support that resembles the hyperlinks we know from the Internet. There are two types of links—subject links marked as 'some subject' and option links marked as 'option'.

A **subject** link refers to the beginning of a section in the help system, whereas the **option** links takes you directly to the description of a certain option. When you place the cursor on a link, you can press *Ctrl-]* to follow the link no matter what type of link it is. This is very nice, but if you are using a non-English keyboard layout, the key for *]* is often not

available with a single key press. In this case, it could be nice to remap the key to some other key. In an Internet browser, you could navigate to a link and press *Enter*, and to reflect this you could have a mapping like the following:

```
nmap <buffer> <CR> <C-]>
```

If you are in a browser and want to return to the previous page you visited, then you can press the *Backspace* key. This feature would be nice to have in the Vim help system; hence a mapping like the following could be useful:

```
nmap <buffer> <BS> <C-T>
```

Now we can move forward and backward in the hyperlinks in the help system, with easy to remember key bindings. Now let's also add some easy navigation keys for finding the next/previous place where a subject or an option link is situated in the currently open help file. This way we can easily scroll through the help file until we find what we are looking for.

```
nmap <buffer> o /''[a-z]\{2,\}''<CR>
nmap <buffer> O ?''[a-z]\{2,\}''<CR>
nmap <buffer> s /\|\S\+\|<CR>
nmap <buffer> S ?\|\S\+\|<CR>
```

Now you can press *o* to go to next place where an option link is, or *s* if you want to go to the next subject link. The same is available if you want to move backwards, you just have to press the capital letters instead—hence *O* for previous option link and S for previous subject link.

To prevent the mappings from interfering with other key mappings, you can add them to a file called **help.vim** and place it in **$VIMHOME/ftplugin/**.

So now, we are only missing a final bit of our improved help system navigation. We need a way to open the help system a bit faster. Normally, when you press the *F1* key, the help system opens on the default page. It would, however, be nice if the key instead did a lookup of the word currently under the cursor. So let's look at a key mapping for this:

```
:map <F1> <ESC>:exec "help".expand("<cWORD>")<CR>
```

This one is a bit hard. As the `:help` command is normally used for looking up Vim commands, the commands on the line after the `:help` are not interpreted. Because of this we have to wrap the command in the `:exec` command.

To get the word under the cursor, we use `<cWORD>`. The *WORD* part is in uppercase; this means that all characters except white spaces (space and tab) can be part of the word. This is needed because Vim commands can contain special characters other than alphanumeric (think of if you were to look up `<cWORD>`).

This key mapping can be used from outside the help system, and could therefore be added to your `vimrc` file, and not placed in **help.vim**.

[Vim]$^{6+}$ [GVim]$^{6+}$ Faster Navigation in Multiple Buffers

Often, you are not just working on one file, but have multiple files open. For every file you have open, you have a Vim buffer. A buffer can be shown or hidden, which means that to find the file you want to work on, you will need to find the buffer containing it.

You could of course, bring up the list of buffers and find the right buffer in the list. To show the list of buffers you can use the command:

```
:buffers
```

This list is not interactive, so in order to select the buffer you want to go to, you need to look up the number at the beginning of the line where the file is listed. This is the number of the buffer where the file is placed. With this number, you can now go directly to the buffer by executing the following command:

```
:buffer N
```

where N is the number of the buffer.

This way of navigating the buffers is not always the most efficient. You could also cycle through the buffers by using the following commands:

```
:bnext
:bprevious
```

Even though these commands can be accessed via their shorter names `:bn` and `:bp`, they are still commands you have to write in normal mode. This means that it takes at least five key presses to execute the command, which is not convenient.

So in order to make this buffer cycling a lot faster, you could add the following mapping to your `vimrc` file:

```
map <C-right> <ESC>:bn<CR>
map <C-left> <ESC>:bp<CR>
```

What these map lines do is make it possible to use *Ctrl+left arrow* key to go to previous buffer and *Ctrl+right arrow* key to go to next buffer. So by holding down *Ctrl* while pressing the left/right keys repetitively, you can easily and quickly cycle through the files you have open.

If you want to toggle back and forth between current and previous buffer, then you can use *Ctrl-6* (*Ctrl-o Ctrl-6* if in insert mode) or `:e #`.

[Vim]$^{7+}$ [GVim]$^{7+}$ Open Referenced Files Faster

In many programming languages, you can include other files in the current file, and thereby split the contents across multiple files. Often the inclusion of the file resembles something like:

```
#include "somefile.h"
```

Here we have `"somefile.h"` as the name of the file we included.

It would be nice to have an easy way to open the included file. Vim has a command that helps you in doing exactly that. Move the cursor to the place in the file where the filename of the file you want to open is, and execute the following command in normal mode:

`gf`

You can remember this command by thinking of 'goto file'. Vim looks for the file in several different places:

1. Vim looks in the places it has defined in the *Path* option, and relative to the currently open file.
2. If not found, Vim uses the *suffixadd* function to see if it can find the file by adding one of the suffixes (e.g. adding `.c` to the filename).
3. If still not found, Vim uses the *includeexpr* expression to convert the filename to something that is hopefully understandable as a filename (for example, **java.com.http** is translated to **java/com/http.java**)

If Vim finds the file, then it opens the file in the current buffer, and if not, it returns an error message. If the buffer you are currently in is not saved, or if anything else is going on such that Vim cannot abandon the currently open file, then Vim cannot open the file. This can be quite annoying, but it is a problem we can prevent from happening. By simply adding the following command to your `vimrc` file, you will always open the new file in another buffer and Vim does not have to abandon the currently open file:

```
:map gf :edit <cfile><CR>
```

This command simply overwrites the **`gf`** command to instead open the file under the cursor with the `:edit` command—and if it does not exist, then open a new empty buffer.

If you want Vim to support filenames with spaces in when using **`gf`**, then add the following to your `vimrc`: `set isfname+=32`
32 is the decimal number representing space in the ASCII table.

Search and You Will Find

We all know the feeling of having seen the things we have misplaced somewhere, but not remembering exactly where. What we normally do in a situation like this is search for the thing we are missing.

In Vim, we can do the exact same thing. Let's split the search into three cases:

1. Searches in current file
2. Searches in multiple files
3. Searches in help files

In the following sections, we will look at recipes that help you with the three types of search.

[Vim]$^{6+}$ [GVim]$^{6+}$ Search the Current File

Even though your file might not be that long, it can still be a pain to find something you are looking for. Vim has several ways to help you find what you are looking for.

So let's look at some examples.

Example 1—Find next occurrence of a word

You know that around where you use the word "someWord", you have the text you are looking for. To find this, you simply need to do a search for it by executing the following command in normal mode:

```
?someWord
```

The command searches backwards in the file for the first occurrence of the word after the question mark. If you are at the end of the file, this is the perfect way to search for a word, but if you were at the beginning of the file it would make more sense to search forward in the file. This is done by exchanging the question mark for a slash:

```
/someWord
```

The word might be in the file several times, and maybe the first place you found wasn't the place you were looking for. No worries, you simply need to press `n` to go to the next occurrence of the word in the direction of the search. If you would rather change direction, then simply press `N` instead, and it will instead find the preceding occurrence of the word.

If you want to do the same search again, simply use `??` or `//` instead of writing the entire word again.

If you add `set incsearch`, your search will be live and the cursor will start jumping through the file as you type. At any point while writing, the cursor will be placed at the next/previous occurrence of the word as it is written right now.

You have to press *Enter* in order to actually execute the search in the end, else the cursor will go back to where it came from. To cancel a search and go back to where you came from, you can simply press the *Esc* key.

Example 2—Search for word under cursor

If you are already near one occurrence of the word you are looking for, but it is just not the right one, or maybe you want to look through all places where a certain word is used, and the word is already written, why use extra key presses on writing the word again? Vim has just the right commands for you. Place the cursor on or just in front of the word you want to search for and press either one of the following two keys in normal mode:

```
#
*
```

The first one searches for the previous occurrence of the word under the cursor, and the second one searches for the next occurrence of the word. Pressing the key multiple times jumps to the next/previous occurrence of the word, over and over again. This makes it really fast to jump through all occurrences of the word.

Maybe your word isn't actually a complete word, but just a part of a word. Vim also has a command for this. Simply press the following key combination in normal mode:

```
g#
```

```
g*
```

Now Vim does not just jump to the next occurrence of the word, but also to any occurrences where the word is part of another word. For example, placing the cursor on the word "*foo*" and pressing `g#` will make Vim jump to the next "*foo*", in both "*foobar*" and "*food.*"

[GVim]$^{7+}$ Search in Multiple Files

Maybe what you are looking for is not in the current file. Maybe you are not even sure which file you should be looking in to find what you are looking for. On a Unix-flavored operating system like GNU/Linux, you typically have the command-line tool grep, that looks for certain words or patters in all the files specified. In Microsoft Windows there is a similar tool available as the commands FIND and FINDSTR. These are, however, not commonly used by Windows users. In order to provide all Vim users, no matter which platform, with a way to search through files, Vim has its very own grep command. The command to use is:

```
:vimgrep /pattern/[j][g] file file2... fileN
```

This command takes two arguments. The first is the pattern you want to search for. You can use Vim's regular expressions in the pattern or you can just write a word. The pattern needs to be enclosed in `/` and after the last `/` you can add either of the two flags `j` and `g`. The flags helps you select how much to get in your result, and how it should be presented to you.

Instead of `/` around your pattern, you can use any non-ID character. A non-ID character is any character not defined in the *isindent* option.

If the `g` flag is added, then the result will include a line for each match of the pattern. This means that if your pattern is matched three times in the same line, then you will get the line three times in your result. If the `j` flag is added to the end of your pattern, then you will not be presented with the result but it will just be updated into your **quickfix** list for later retrieval (see `:help quickfix` for more information about quickfix lists). Without the `j` flag, you will be moved directly to the first match and the rest of the result will be added to your quickfix list.

To show your quickfix list with the vimgrep result in, simply use the command:
`:clist` or navigate to next/previous match with **`:cnext/:cprevious`**

The second argument to the Vim grep command is the list of files you want to search through. The file list can consist of a single filename, a list of filenames or a pattern using the star wildcard (for example, *.c *.h). You can also use the ** wildcard, like **/*.c if you want to search in all the C files in the current folder, and recursively through all subdirectories.

```
:clist..
 1 Desktop/diverse/fpix-0.90.1/driver/finepix-main.c:309 col 95: dev_err
             rame. Please, report to driver maintainer.\n");
 2 Desktop/diverse/fpix-0.90.1/userspace/fpix-stress-v4l2.c:3 col 13: *
 3 Desktop/diverse/fpix-0.90.1/userspace/fpix-stress-v4l2.c:19 col 5: in
 4 Desktop/diverse/fpix-0.90.1/userspace/fpix.c:3 col 13: * Public domai
 5 Desktop/diverse/fpix-0.90.1/userspace/fpix.c:44 col 5: int main(void)
 6 Desktop/diverse/fpix-0.90.1/userspace/fpixtest.c:3 col 13: * Public d
 7 Desktop/diverse/fpix-0.90.1/userspace/fpixtest.c:16 col 5: int main(v
 8 Desktop/diverse/fuji-finepixa310-test/stream.c:255 col 5: int main(in
 9 bin/bin2iso.c:29 col 5: int main( int argc, char **argv )
10 bluemote/bluemote.c:56 col 5: int main(int argc, char *argv[])
11 bluemote/bluemote.c:147 col 11: if(init_mainmenu()==-1) continue;
12 bluemote/bluemote.c:261 col 10: int init_mainmenu()
13 bluemote/bluemote.c:291 col 22: while((ret = init_mainmenu())==0);
14 bluemote/mouse.c:75 col 1: main (int argc, char **argv)
15 btsco/a2play.c:536 col 5: int main(int argc, char *argv[])
:vimgrep /main/ **/*.c
```

[Vim]$^{7+}$ [GVim]$^{7+}$ Search the Help System

Sometimes when you need help for something in Vim, you might not know exactly what you should look for. You could of course start going through the entire help system, but it consists of several different files and thousands of possible keywords.

So, Vim has the right command to help you out here. As in the previous recipe, the keyword is grep, and for the Vim help system it is centered around the following command:

```
:helpgrep pattern [@LANG]
```

The command takes one argument, the pattern you search for, plus one optional argument to limit the language. Let's look at an example to make it clearer. You need some information on auto completion but do not know where to look for it. You are able to read English hence only want help in this language. A search for this could look like this:

```
:helpgrep completion@en
```

What the command does is search for the word **completion** through all the English (en) documentation. The command takes you to the first match it finds and the rest of the matches are added to the quickfix list for later retrieval.

If you want to use the location list instead of the quickfix list for your result, then you can use the command **:lhelpgrep** instead.

The helpgrep command does not actually look through all the documentation when searching, but uses a tag list containing tags for all the available documentation to look up a pattern. This tag list is, however, not created automatically, so it is important to note that if you install a Vim plugin that has its own documentation, then you need to use the following command:

```
:helptags /path/to/documentation
```

The path to the documentation only needs to be where you have installed the new documentation. But in order for the Vim to actually be able to find the documentation, it has to be in a **docs/** directory in one of the places defined in the *runtimepath* in Vim (see `:help 'runtimepath'`)

X Marks the Spot

Sometimes when editing a line in a file, you have to go to somewhere else in the file to look up something. Afterwards, it can be difficult to find the line you were editing, and you waste valuable time on finding it.

Wouldn't it be nice if you could mark the spot before leaving it, such that it is easy to find later?

Vim has some tools for you that can do just that. We can split it into two categories:

- Visible markers
- Hidden markers

In the following two sections, we will look at the possible ways of adding marks in Vim, and then it's up to you to figure out which one fits your needs the best.

[Vim]$^{7+}$ [GVim]$^{7+}$ Visible Markers—Using Signs

In Vim we have a nice feature for marking a line with a visible mark—signs. A sign is a mark that will show up in the leftmost column in the editor.

If you want to change the color of the column in which the sign is shown, then you can use the following command:
`:highlight SignColumn guibg=darkgrey`

Depending of whether you are using Vim in a console or as Gvim, the sign can be either a combination of characters (for example, >>) or an icon. To use the signs you will need a bit of setting up. You only have to do this once if you have it in your `vimrc` file.

The first thing you have to do is define the signs you want to have. The command you need to use is as follows:

```
:sign define name arguments
```

The arguments can be one of the following:

- `linehl` : Color group you want to mark the line with.
- `text` : The text used as a sign in console Vim (e.g. >> !! or ++). A maximum of two characters can be used per sign.
- `texthl` : Color group you want the sign text marked with.
- `icon` : Full path to the icon you want for the sign in Gvim. The icon should be small enough to fit the size of only two characters. The format should be a bitmap format, but .xpm format is preferred.

An example could be:

```
:sign define information text=!> linehl=Warning texthl=Error icon=/
path/to/information.xpm
```

Now we have defined a sign and added it to our `vimrc` file, and are ready to place the sign somewhere. The command is :

```
:exe ":sign place 123 line=" . line(.) ."name=information file=" .
expand("%:p")
```

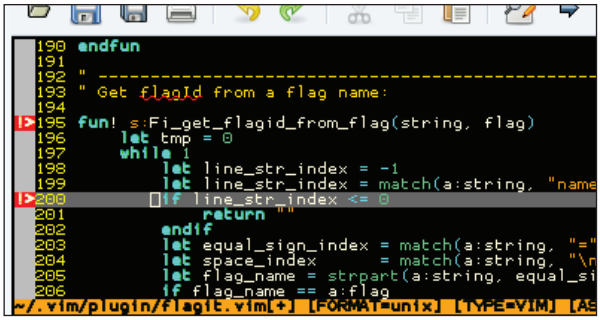
Replace the number 123 with any number you will use as ID for this sign.

As you can see, this is a bit harder, but it can easily be mapped to a key.

What it does is add the sign named **`information`** under the ID 123 to the current line (**`line(.)`**) in the currently open file (**`expand("%:p")`**). Mapping this to a line is:

```
:map <F7> :exe ":sign place 123 line=" . line(".") ."name=information
file=" . expand("%:p")<CR>
```

This maps the information sign to the F7 key such that it will be placed in the current line whenever you press the F7 key.



Sometimes we also want to remove the sign again. In Vim this is called to 'unplace' a sign:

```
:sign unplace ID
```

The *ID* is the ID you gave your sign when you placed it (123 in the above example). This removes the sign from all the places where you have added the sign with that ID. You might want to remove it only from the current file, and can therefore add another argument for the file like this:

```
:sign unplace ID file=name
```

Or from the buffer:

```
:sign unplace ID buffer=bufferno.
```

where **`bufferno`** is the number of the current buffer (see **`:buffers`**).

If you want to remove the sign in the current line, then you can simply use:

```
:sign unplace
```

Let's map this to *Ctrl-F7*, just to make it symmetric with the sign placement mapping we have defined earlier:

```
:map <C-F7> :sign unplace<CR>
```

If you have added several signs with the same ID to a file, then the above mapping will only remove the uppermost sign with the specific ID and not the one in current line.

As this is a chapter about navigation, we also need to have a bit about navigating to a sign here. This is called 'sign-jumping' in Vim and uses the following command:

```
:sign jump ID file=file
```

Here *ID* is the ID of the sign you want to jump to and *file* is the file you want to find the sign in. Instead of `file=file`, you can instead use `buffer=bufferno`.

Again, if the sign has been added with the same ID several times in the file/buffer, then it will jump to the first sign in the file.

Poul Rouget has created a Vim script that makes the usage of signs a lot easier. You can find it here:
`http://www.vim.org/scripts/script.php?script_id=1580`

[Vim]$^{6+}$ [GVim]$^{6+}$ Hidden Markers—Using Marks

Marks is the fast and easy way to add a mark to the current line such that you can later jump to it easily. Basically, it consists of a normal mode command that sets the mark, and a normal mode command to jump to the mark. You won't be able to see if a line is marked or not unless you open the list of marks.

So let's look at how to mark the current line. We simply press the key *m* in normal mode, followed by one of the characters *0-9*, *a-z*, or *A-Z*. If you for instance press `ma`, then it means that the current line is marked with the mark named *a*. If you later want to jump to this line, then you simply press `'a` (single quote + mark name) and you will be taken to the beginning of the line you marked (if indented, then to just before the first non-whitespace character).

In some cases it might not be efficient to be placed at the beginning of the line, but it would be much better to be placed where you were when you added the mark. To jump to this place instead, you simply replace the single quote with a `` ` `` (backtick) like `` `a ``.

The different mark names have different meanings and work areas:

- *0-9* : Marks set from `.viminfo` and normally only used by Vim itself (e.g. mark *0* is the place where cursor was when the file was last exited). A user can, however use this to make an "open recently used" functionality.
- *a-z* : Marks only available in the current file. These marks are deleted when the file is closed. You can only jump to a lowercase mark if you are inside the buffer containing the file.
- *A-Z* : Marks available across files. These marks can be jumped to, even if you are not in the file where the mark is situated. If a viminfo file is available then these marks are saved until next time you edit a file.

You can always get a complete list of your marks by using the following command:

```
:marks
```

This shows which files the different marks are set in and on what lines. To delete one or more marks, you can use the command:

```
:delmarks markid markid...markid
```

Examples of how it can be used are:

```
:delmarks a b c
:delmarks a-c
:delmarks a f-i 1-4
```

If you want to delete all marks in the current buffer, then simply use the command:

```
:delmarks!
```

Other types of marks are set by Vim all the time when using it. These can be marks for where the cursor was last time the insert mode was exited, beginning/end of text selected in visual mode, the last place you changed something, etc.

Look in `:help mark-motions` for more information on how to use marks and which other types of marks you have available.

Summary

In this chapter, we have looked at alternative ways for boosting the speed at which we navigate through files and buffers in Vim.

First, we looked at how to navigate through a single file faster by using the contextual structure of the file for navigation. We also looked at a nice recipe for how we can make it easier to navigate files with long, wrapped lines.

Next, we looked at how to to navigate the Vim help system faster and learned how simple key bindings can make the help navigation more intuitive and recognizable.

Now, we knew how to navigate inside a file, but we also needed to know how to navigate between files and buffers. The next section took us through how to navigate the buffers faster and how to open a file that is referenced by another file with only two key presses.

We can navigate in many ways, and in the preceeding sections we looked at how to use the search mechanisms in Vim to navigate not only the open files, but also files on the disk. We also learned how to use searches in the help system to find help on topics we could not find normally.

Finally we have looked at how to use signs and marks to jump around in files, and how Vim helps us simply by adding some marks automatically when we use it.

4
Production Boosters

In this chapter, we will look at how even small changes can make work go faster and more smoothly in Vim. Some recipes introduce you to features in Vim, while others will show you how scripts can help.

It doesn't matter whether you use Vim for making small changes to configuration files, or if you use it as you primary editor in a large development project, you will find recipes in this chapter that can help you improve your performance when using Vim.

This chapter contains recipes that cover:

1. Templates using simple template files
2. Templates using abbreviations
3. Auto-completion using known words and tag lists
4. Auto-completion using omni-completion
5. Vim macros and macro recording
6. Using sessions
7. Project management using sessions
8. Registers and undo branches
9. Folding for better overview and outlining
10. vimdiff for change-tracking
11. Opening files everywhere using Netrw

After reading this chapter, you should be able to boost your productivity in Vim by several percent.

Using Templates

No matter what type of files you are working with, there are always some basic things to set up when the starting off on a new file. Creating this setup is a tedious task, and even worse is the fact that you have to do it again when you start on a new file. So why spend a lot of time on these things when you could just as well create templates for these types of structural patterns?

In the next couple of sections, we will look at recipes for two types of templates:

1. File type-specific templates for new files
2. Content-pattern templates

So let's get started on creating some templates.

[Vim]$^{6+}$ [GVim]$^{6+}$ Using Template Files

Every time you start working on a new file, it is most likely that the first thing you'll do is add some sort of header (or other information) to the file. What you have to add is of course, dependent on which file type you are working on. Some examples could be :

- Adding basic structure (`<html>`, `<head>`, and `<body>`) to new HTML files
- Adding a header to all C files and also a main function to `main.c` files
- Adding the main class structure to a Java file

You can probably find many other things you would like to add to the file types you work with.

So, how do we create a template file? Let's use an HTML file template as an example. The structure in such a file is quite static, and hence great to have a template for. Our simple template could look like:

```
<html>
    <head>
       <title></title>
       <meta name="generator" content="Vim" />
       <meta name="author" content="Kim Schulz"/>
    </head>
   <body>
       <p>Content goes here...</p>
   </body>
</html>
```

We create a directory in our **VIMHOME** called `templates/` and place a file with the above HTML code in the directory—save the file as `html.tpl`.

Now the first template is in place, but we need to get it loaded into all new HTML files that we create. To do so, we add the following auto-command to our `vimrc` file:

```
:autocmd BufNewFile *.html 0r $VIMHOME/templates/html.tpl
```

What this command does is ensure that when you create a new file with the file extension `*.html`, the content of your template file is read into your new file. This way your file gets prepared with the template's content before you can start editing it.

All this is very nice, but after adding a bunch of templates, you might get tired of adding lines to your `vimrc` file. So let's make our first line a bit more intelligent:

```
:autocmd BufNewFile * silent! 0r $VIMHOME/templates/%:e.tpl
```

What this single line does is that, whenever you open a file, it looks for a template that matches the extension of the file—for example, when creating the file `index.html`, it looks in `$VIMHOME/templates/` for a file named `html.tpl`.

If there is no template for the file type, then it simply creates an empty file as usual.

Let's take these templates even further by adding support for placeholders (for places where you want to add text to the file, fast). A placeholder could look very different depending on what you like, but I propose something like `<+KEYWORD+>`. So if we take a line from the HTML template mentioned above, and add a placeholder to it, it could look like:

```
<html>

    <head>
       <title><+TITLE+></title>
       <meta name="generator" content="<+GENERATOR+>" />
       <meta name="author" content="<+AUTHOR+>"/>
    </head>
   <body>
       <p><+CONTENT+></p>
   </body>
</html>
```

Now we have the placeholders in place and only need a way to jump between them. So let's add a command to our `vimrc` that will make it easy to make this jump. We want to use *Ctrl-j* as the jump key binding, because it can easily be used in the insert mode and the 'j' (for jump) makes it easy to remember. The command could look like this:

```
nnoremap <c-j> /<+.\{-1,}+><cr>c/+>/e<cr>
inoremap <c-j> <ESC>/<+.\{-1,}+><cr>c/+>/e<cr>
```

Now you can easily jump to the next placeholder in the file, change the text, and jump on to the next placeholder—simply by pressing *Ctrl-j* text, *Ctrl-j* text, etc.

By having the keyword in the placeholder, you can easily see what you are supposed to add there.

You can mark your placeholders by adding a match command to your `vimrc`: `match Todo /<+.\++>/` (replace Todo with whatever color group you like).

[Vim]$^{6+}$ [GVim]$^{6+}$ Abbreviations as Templates

In the previous section, we've learned how to make templates for entire file types, so now, let's look at how to make templates for patterns inside the file-content itself.

In Chapter 2, we briefly looked at how to use abbreviations for limiting the amount of key presses, whenever possible. Now, let's take the idea of using abbreviations and copy it to our template system. Let's look at the command and what it's all about, just to refresh our memory:

```
:iabbrev match replace-string
```

We only want the command for insert mode, because it is there we want to use the pattern templates. An example could be the following pattern template for a C file:

```
:iabbrev <buffer> for( for (x=0;x<var;x++){<cr><cr>}
```

which gives us a nice little for loop whenever we input `for(` in the contents of the file. The `(` is added to prevent manually written for loops from being converted. The inserted content will look like:

```
for (x=0;x<var;x++){

}
```

Having **<buffer>** in front of an abbreviation limits its availability to current buffer.

As you can see, this is quite static code, so in order to make it a bit more flexible, let's use the placeholder concept we introduced in the previous section.

Because the placeholders, in this case, are more like jump points, we simplify them to be just:

```
<+++>
```

Besides this, we need one single placeholder for where the cursor should be placed after inserting the pattern template. In case of the above example, it would of course be nice to have the cursor placed right after the start parenthesis.

So to make this work, we introduce the placeholder, `!cursor!`, and the command will look like:

```
iabbrev for( for(!cursor!;<+++>;<+++>){<cr><+++><cr>}<Esc>
:call search('!cursor!','b')<cr>cf!:
```

(All of above on one line.)

Now whenever the abbreviation `for(` is written, the for loop is inserted, the cursor is moved to the placeholder `!cursor!` (which is removed), and you will be ready to fill in the parameters for the for loop—and jump to the next parameters with *Ctrl-j*.

It probably won't take you long to realize that many programming languages have the same main structures (like the for loop), but they differ just enough to not be able to use the same pattern template. So let's go back and look at what we already have in our template system, and see if we can make the templates file-type aware.

In the previous section, we opened a template file depending on the extension of the file. It looked like:

```
:autocmd BufNewFile * silent! 0r $VIMHOME/templates/%:e.tpl
```

Let's modify this such that it also loads the appropriate abbreviations for our pattern templates.

To make the code clearer, we also move the functionality out into a function of its own. It could look like the following:

```
function! LoadTemplate(extension)
    silent! :execute '0r $VIMHOME/templates/'. a:extension. '.tpl'
    silent! execute 'source $VIMHOME/templates/'.a:
extension.'.patterns.tpl'
endfunction
```

And to actually call the function we change the above `autocmd` to look like:

```
:autocmd BufNewFile * silent! call LoadTemplate('%:e')
```

The `LoadTemplate` function looks in the templates folder in your `$VIMHOME` for two files: `EXTENSION.tpl` and `EXTENSION.patterns.tpl` where EXTENSION is replaced with the extension of the file you are currently opening.

The first file will hold your template for the file type, and the second file will contain the abbreviation commands for all the patterns you have created for this particular file type. If it does not find the file, then the `silent!` will make sure that it does not give you an error message, but simply returns nothing to you.

So now it is up to you to fill in the templates and thereby complete your personal templating system.

Many template system scripts exist for Vim. Most of them are based on the same concept as described in this chapter. They have, however, added even more functionality. I will recommend that you look at mu-template by Gergely Kontra, if you would like more templating options than described here: `http://www.vim.org/scripts/script.php?script_id=222`

[Vim]$^{6+}$ [GVim]$^{6+}$ Using Tag Lists

Tag lists are kind of the programmer's dictionary. A tag list is actually a file containing all sorts of keywords that that can identify parts of a program. It can be function names, variable names, class methods, etc., depending on the programming language that you are using. Tag list files are actually not an output from Vim, but rather an output from one of several tag-list generators. Among these the best known are:

- Exuberant Ctags For C, C++, Java, Perl, Python, Vim, Ruby (and 25 others)
- Vtags For Verilog files
- Jtags For Java files
- Hdrtags For C/C++, Asm, Lex/Yacc, LaTeX, Vim, and Maple
- Ptags For Perl files

Because Ctags is absolutely the most used, and the one that supports the most languages, it will be the one we use in the following examples.

We will work with a project written in C and consisting of three files:

`main.c` : The main file containing the main function of the program

`myfunctions.c` : The source for the functions needed in the program

`myfunctions.h` : The header file for the functions in `myfunctions.c`

The code in the files is almost done, so let's make a tag file for the files.

We invoke the `ctags` command-line program as follows in the directory where your source files are placed:

```
ctags *.c *.h
```

You will now notice that a new file called **tags** has been created in the directory where you invoked the `ctags` command. This file is your tags file and contains information about all the functions and variables in your code.

The `ctags` program takes a lot of arguments for choosing programming language etc. See `ctags –help` for more info.

We need to tell Vim that it should use the tags file, which is done by setting the tags setting.

```
:set tags=/path/to/tags
```

Now Vim knows about the tags and you are ready to use it in your work.

In the main file, we use the functions from the `myfunctions.c` file. Let's say there is a function called `calcValue`, but you are not sure which arguments it takes. Here, it would be nice to see how it is defined, and it is here the tags file comes in handy. You simply start typing the function name until the `(` like:

```
myvalue = calcValue(
```

Now place the cursor on the function name, go into normal mode and then press *Ctrl-]*. One of two things happens:

1. There is only one match, and you are moved directly to where the function is defined.
2. There are multiple possible matches and a list of matches is shown.

In the second case, you can select which of the matches in the list you want to jump to. This is nice if you are working with languages where you can overload functions and hence have multiple editions of a function.

After you have looked at the function, you need to go back and actually complete the code that you've started. To do so, you simply press *Ctrl-t* and you will be brought back to where you originally came from.

In Gvim, you can also use the mouse to go to the definition of a keyword. Just hold down the *Ctrl* key while pressing *mouse-button1*.

You can view jumping between tags as using a tag stack. When you go to a keyword, you push the tag on the stack, and when you go back to the previous place, you pop a tag off the stack.

You can actually see the stack by using the following command:

```
:tags
```

```
:tags
  # TO tag         FROM line  in file/text
  1  1 clear_edge_list    71  clear_edge_list();
  2  1 edge_list          62  edge_list = NULL;
  3  1 show_operation_failed_dialog   113  show_operation_failed_dialog();
>
Press ENTER or type command to continue
```

The tag in the list that is marked with **>** at the beginning of the line, is the tag you are currently at. When using *Ctrl-]* and *Ctrl-t*, you move up and down in the stack, but you can also work the stack with commands:

`:tag` move to next tag in stack

`:pop` move to previous tag in stack

When you have used a lot of tags and jumped around between them, then it might be nice to get a list of the ones you have available. To get such a list, simply use one of the following commands:

```
:tselect
:ptselect
```

The first one gives you a list of matching tags and you can then select the one you want by pressing the number you find at the beginning of the line.

The second command does the same, except that it shows the list in a preview window. If you selected the wrong tag in the list, or just want to see one of the others, then you can move between them with the following commands:

`:tnext` : Move to next tag in list

`:tprev` : Move to previous tag in list

You might not see the strength of the tag lists in this example, but imagine that you project was not just three files, but 1000 files across hundreds of directories. Then you suddenly just can't go around remembering where each function is, and you would need a thorough indexing mechanism like tags files.

Easier Taglist Navigation

On most non-english keyboard layouts, the *]* key is not directly available and you would need to press, for example, *Ctrl-AltGr-9* to execute the *Ctrl-]* tag jump. In those cases, it would be nice to map the commands to more accessible keys. I use the following mappings:

```
:nmap <buffer> <F7> <C-]>
:nmap <buffer> <S-F7> <C-T>
:nmap <buffer> <A-F7> :ptselect<cr>
:nmap <buffer> <F8> :tnext<cr>
:nmap <buffer> <C-F8> :tprev<cr>
```

Now, you can jump forth and back between tags with *F7* and *Ctrl-F7*, get a list of the tags with *Alt-F7*, and go through the used tags with *F8* and *Ctrl-F8*.

Other Usages of Taglists

Tag lists are not only used by programmers for looking up functions and variable definitions. They are also used for a lot of other interesting things. Just to give you an idea, here's a short list of examples where they is used in Vim scripts:

- **lookupfile.vim**
  Hari Krishna Dara has created a script that can use tag lists to find files in a tagged project simply by writing the filename. Find the latest version of the script here: `http://www.vim.org/scripts/script.php?script_id=1581`.
- **taglist.vim**
  Yegappan Lakshmanan has created the taglist plugin for Vim, which is very popular among programmers. It is a complete source-code browser that gives a great overview of the functions, keywords, variables, definitions, etc. in a split-window. In Gvim it can even give you a complete menu with the tags of the project. You can find a lot more information about this plugin here: `http://www.vim.org/scripts/script.php?script_id=1581`.
- **ctags.vim**
  Gary Johnson and Alexey Marinichev have created a simple, yet powerful plugin called **ctags.vim**. It simply shows the name of the function the cursor is currently placed in, in the status bar or window title. The script automatically generates tags files for the currently opened file using the program **Exuberant Ctags**. Find more info about the script here: `http://www.vim.org/scripts/script.php?script_id=610`.
- **autoproto.vim**
  Jochen Baier has created a very useful script for C programmers. This script shows the prototype of the currently typed in function in a preview window whenever the programmer presses the first `(` after the function name. Find more info about the script here: `http://www.vim.org/scripts/script.php?script_id=1553`.

You can read a lot more about tags and how to use them in the Vim help system under `:help tags`.

Using Auto-Completion

As a Vim user that obeys the philosophy of Vim, you will do anything to minimize the number of key presses—because extra keys pressed equals extra time wasted.

So why type each word to the end, when Vim is able to guess what you are typing and automatically complete the word for you?

In Vim, there are multiple ways to auto-complete the words you are typing. Some methods simply complete words you have written once in one of the opened buffers, while others involve analyzing the code you are working on—not just the current file, but the entire source tree.

In the following sections, we will look at three different ways to use auto-completion in Vim:

- Auto-completion with known words
- Auto-completion using a dictionary file
- Context-aware auto-completion with Omni-completion

There will also be some small tricks on how to make it more comfortable to use auto-completion by using well recognizable key bindings.

[Vim]$^{6+}$ [GVim]$^{6+}$ Auto-Completion with Known Words

In this recipe, we will look at maybe the simplest type of auto-completion and at the same time the most overlooked—auto-completion with known words.

Almost no matter what you are writing, you will eventually write the same words over and over again. In Vim, you can simply type in the first couple of letters of the word and then press *Ctrl-n*.

Example:

You want to write the sentence:

"*I have beautiful flowers in my flower garden.*"

Since you have no text besides this in the file, you will have to write the entire first part of the text until it looks like:

"*I have beautiful flowers in my f*"

Now, you would normally continue the word "*flower*" but since it is already there, you can simply press *Ctrl-n* and the word will expand to "*flower*".

As your text evolves, you will see that you can start using auto-completion on more and more words.

What *Ctrl-n* actually does is look for a matching word by going forward through the file. If you know that you have just used the word, then it will be faster to use *Ctrl-p* instead because Vim will then search backwards in the file for a matching word. In general, you won't feel the difference unless you are working with really large files, or there are many possible matches.

[Vim]7+ [GVim]7+ Auto-Completion using Dictionary Lookup

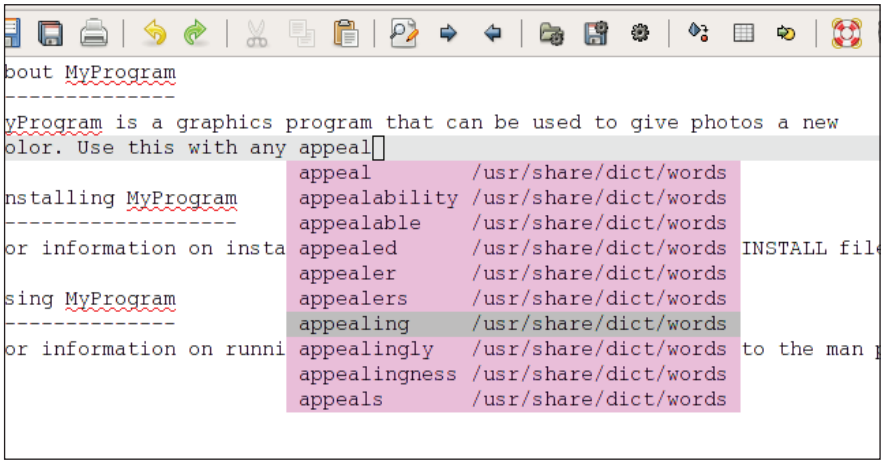
A neat trick is to find a large dictionary file with all kinds of words in your favorite language, then load this file into Vim as an dictionary (such files can easily be found on the Internet). To load the file into Vim as a dictionary, simply add it to the dictionary setting with:

```
:set dictionary+=/path/to/dictionary/file/with/words
```

Now Vim suddenly knows a lot of words beforehand, and you can simple auto-complete using these words. Something is different, however. Now the words we use to look in are not words from one of the open buffers, but keywords from one of the dictionary files available in the dictionary setting. This is why you will need to use another key binding in order to do the completion this time.

```
ctrl-x ctrl-k
```

By pressing *Ctrl-x* you get into a completion mode, and by pressing *Ctrl-k* you do a lookup for a keyword (remember *k* for keyword) in the dictionaries.



Other completion types are available. Some of them are in the following list:

Ctrl-x plus:

- *Ctrl-l* Complete whole lines of text
- *Ctrl-n* Complete words from current buffer
- *Ctrl-k* Complete words from dictionaries

- *Ctrl-t* Complete words from thesaurus (see `:help 'thesaurus'`)
- *Ctrl-i* Words from current and included files
- *s* Spelling suggestions (Vim 7.0 and newer only)

Others will be further described in the next sections.

[Vim]$^{7+}$ [GVim]$^{7+}$ Omni-Completion

We all have our perfect solution for what should be auto-completable and what shouldn't. In Vim there had been no way to give the user complete control over what to do about completion until version 7.0 came out.

Vim 7.0 introduced a new completion technique called 'omni-completion'. It gave the user the possibility to define exactly how the functionality of the completion should work—in fact, then, the user would have to write the completion function himself or herself (unless someone else has already done it).

As with the completions mentioned in the previous section, the completion is invoked by typing in some letters and then going into completion mode by pressing *Ctrl-x* followed by *Ctrl-o* to make an omni-completion.

To add your own completion function, you simply do the following:

```
:set omnifunc=MyCompleteFunction
```

Now you would just have to create a function called `MyCompleteFunction` that gives you the completions. This setting is only available to the currently active buffer, and you will have to set it for all buffers where you want to use it.

Setting `omnifunc` is normally done in a file type plugin such that it is bound to a single file type.

So let's look at an example on how such a function could look like. If for instance, you have a file with all your contacts with one name + email address on each line like:

```
Kim Schulz|kim@schulz.dk
John Doe|john.doe@somedomain.com
Jane Dame|jd@somedomain2.com
Johannes Burg|jobu@somedomain3.net
Kimberly B. Schwartz|kbs@somedomain.com
...
```

Now, you would like to insert an email address by writing a name and doing auto-completion on it. A function for this could look like:

```
function! CompleteEmails(findstart, base)
     if a:findstart
       " locate the start of the word
       let line = getline('.')
       let start = col('.') - 1
       while start > 0 && line[start - 1] =~ '\a'
         let start -= 1
       endwhile
       return start
     else
       " find contact names matching with "a:base"
       let res = []
       " we read contactlist file and sort the result
       for m in sort(readfile('/home/kim/.vim/contacts.txt'))
         if m =~ '^' . a:base
             let contactinfo = split(m, '|')
             " show names in list, but insert email address
             call add(res, {'word': contactinfo[1],
                     \ 'abbr': contactinfo[0].' <'.contactinfo[1].'>',
                 \ 'icase': 1} )
         endif
       endfor
       return res
     endif
   endfunction
```

The function takes two arguments, which are needed for an omni-completion function. The first time Vim calls the function, it sets the first argument, *findstart,* to 1 (and *base* is empty) which means that this is the first invocation, and that it should find the beginning of the word you have written so far.

Vim then invokes the function again, and this time with *findstart* set to 0 and base set to the word you have started auto-complete on. This time the function opens up the contact list file and reads it line by line into a list. It sorts the list such that the list in the completion popup is ordered, and then it iterates over the list.

The lines are split at the `|`, and then the ones that begin with the same letters as the word you completed on are added to a result that the function will return. The function can modify how the popup content looks and what it should match on. This is done by not just adding the email addresses, but instead building a dictionary (see `:help Dictionary`) where some specific keys are set. In this case, we use three keywords:

- word The actual word that should be inserted.
- abbr This word is used instead of "word" in the popup list.
- icase If this is a non-zero value, then the matching is case-insensitive.

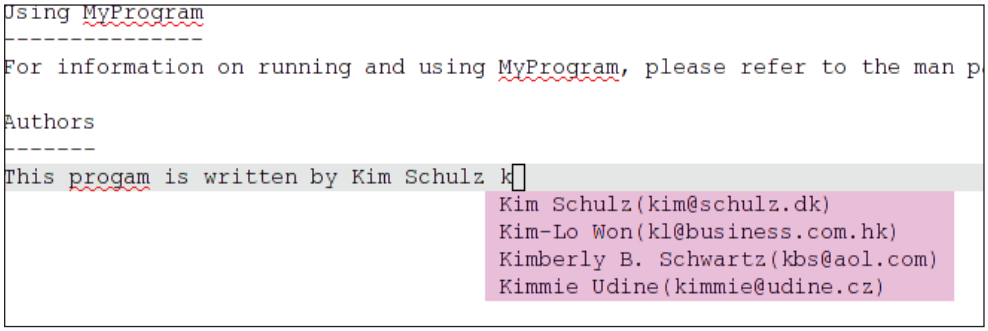
Other keywords and their functionality can be found in the help system under

```
:help 'omnifunc'
```

So now Vim has a list of words for its popup, or in this case small lines like:

```
"Kim Schulz <kim@schulz.dk>"
```

Whenever you write some letters like "ki" and then pres *Ctrl-x Ctrl-o*, then Vim will show the popup with all the names that starts with "ki".



To move between the items in the list, you can keep pressing *Ctrl-o* to cycle through the list. Alternatively, you can press *Ctrl-n* to move forward in the list and *Ctrl-p* to go backwards in the list.

[Vim]$^{7+}$ [GVim]$^{7+}$ All-in-One Completion

You might wonder how you will ever be able to remember all these keyboard shortcuts and why you could not just use the same for all completion types depending on which one you have available. With Vim, you can of course do this if you want. So let's look at how we can do this in a way that is easy to remember for you.

Nearly any other editor that supports completion has this functionality mapped to the tab key.

In the help system, you will find a function called `CleverTab()` if you look in

```
:help ins-completion
```

This function lets you use *Tab* to complete words instead of *Ctrl-n*. It could distinguish between whether it should insert a tab character or do completion. If you pressed tab at the beginning of the line (indention) or after another whitespace character then it inserts a tab character—in the rest of the cases it would try to do known-word completion.

We take this `CleverTab` function and extend it even further such that it selects the completion method to use from this prioritized list:

1. Omni-completion
2. Dictionary-completion
3. Known-word-completion

A function that can do this, could look like the following:

```
function! SuperCleverTab()
   "check if at beginning of line or after a space
    if strpart( getline('.'), 0, col('.')-1 ) =~ '^\s*$'
      return "\<Tab>"
    else
       " do we have omni completion available
       if &omnifunc != ''
          "use omni-completion 1. priority
          return "\<C-X>\<C-O>"
       elseif &dictionary != ''
          " no omni completion, try dictionary completio
          return "\<C-K>"
       else
          "use omni completion or dictionary completion
          "use known-word completion
          return "\<C-N>"
      endif
    endif
endfunction

" bind function to the tab key
inoremap <Tab> <C-R>=SuperCleverTab()<cr>
```

Add the function and the binding to your `vimrc` file, and then you are ready to do completion with your tab key.

You simply have to press the *Tab* key to do you completion, and the function checks to see if it should insert a tab character. If not, then it checks to see if you have a omni-completion function (in *omnifunc*) available. If this is also not the case, then it looks if there is a dictionary available. If you have no dictionaries available, then it falls back on using simple known-word completion.

[Vim]$^{6+}$ [GVim]$^{6+}$ Using Macro Recording

Probably the most overseen production-booster when working with monotonic structured text is the ability to record input macros to do them over and over again.

The interface for doing this is extremely simple, but nearly everything can be recorded, so it reveals a very powerful tool.

Let's start by looking at the commands to use:

qa : Record from now on into register **a**. Any register can be used but q is often used for simplicity.

q : If pressed while recording, then the recording is ended.

@a : Execute the recording in register **a** (replace with any register).

@@ : Repeat last executed command.

You can add any number before the **@** to repeat the execution of the recording that number of times. For example **15@a** will execute the recording in register **a** 15 times.

So let's look at a normal recording session in Vim:

```
qq
command1
command2
....
commandN
q
10@q
```

You might wonder what this can be used for; because when is it exactly that you need to execute a list of commands over and over again? This is best shown with an example.

Imagine you have a large list with information. It could e.g. be a log file from a Unix system. That could look something like:

```
Oct  8 21:23:34 laptopia kernel: ACPI: bus type pci registered
Oct  8 21:23:34 laptopia kernel: PCI: PCI BIOS revision 2.10 entry at 0xe9694
Oct  8 21:23:34 laptopia kernel: Setting up standard PCI resources
Oct  8 21:23:34 laptopia kernel: ACPI: Subsystem revision 20060127
Oct  8 21:23:34 laptopia kernel: ACPI: Interpreter enabled
Oct  8 21:23:34 laptopia kernel: ACPI: Using IOAPIC for interrupt routing
Oct  8 21:23:34 laptopia kernel: ACPI: PCI Root Bridge [PCI0] (0000:00)
Oct  8 21:23:34 laptopia kernel: PCI quirk: region 1000-107f claimed by ICH6
Oct  8 21:23:34 laptopia kernel: PCI quirk: region 1300-133f claimed by ICH6
```

```
Oct  8 21:23:34 laptopia kernel: PCI: Ignoring BAR0-3 of IDE controller 001:1
Oct  8 21:23:34 laptopia kernel: PCI: Transparent bridge - 0000:00:1e.0
...
```

Now, you want to convert this file into an HTML file where the info is presented in a table with each column of text represented by a table column. You basically want each line to look like:

```
<tr><td>Oct 8 21:23:34</td><td>laptopia</td><td>kernel:</td><td>ACPI ...</td><tr>
```

You could start editing the file line by line, until you came to the end of the file. But how about just editing one line while recording it, and then playing back the commands on the other lines? The command execution could look like the following starting with the cursor placed at the beginning of the first line:

`qa`	: Start recording into register a.
`i<tr><td>[ESC]`	: Go into insert mode, insert first HTML, go back to normal mode.
`/ [CR]`	: Search forward for whitespace.
`3n`	: Advance 3 whitespace searches forward.
`xi</td><td>[ESC]`	: Delete whitespace, go into insertmode, add HTML, to normal mode.
`n`	: Goto next whitespace.
`xi</td><td>[ESC]`	: Delete whitespace, go into insertmode, add HTML, to normal mode.
`n`	: Goto next whitespace.
`xi</td><td>[ESC]`	: Delete whitespace, go into insertmode, add HTML, to normal mode.
`A</td></tr>[ESC]`	: Append the final HTML to the end of the line and go in normal mode.
`j^`	: Advance a line and move to the beginning of it.
`q`	: End macro recording.

[ESC] press escape key, [CR] = press return key

So now we have one line ready, the cursor placed just correct on the next line, and a macro recording to play back on the rest of the lines.

You can playback the recording with `@a` or you can simply play back the command on each line in the file with `9999@a`. All you need now is to add a header and a footer to the file, but that is not really interesting here.

This is just one place where macro recordings can be used, and if you think back, you will probably remember situations where you could have optimized your work by using a macro recording.

Using Sessions

Have you ever wondered how much information Vim actually holds for you about a wide range of settings and things like:

- Open files, buffers, windows, and tabs
- Command history
- Points of change in the text
- Selections and undo branches
- Size of windows, splits, and GUI window
- Place of cursor

...and many other things. The stored information can be split into three different categories:

The first type of setting is called a **View** and applies to a single window in Vim. A view can be saved and restored such that a window will have the same look and setup every time you use the view.

The second type of settings are called **Sessions** and are collections of views and info about how they are inter-operating. Like views, sessions can also be saved for later retrieval.

The final type of setting includes all the rest; all the global settings that does not directly apply to any window in Vim. These settings are stored with the session, such that they can also be saved/restored.

In the following sections, we will look at how sessions can be used for different tasks during your daily work in Vim.

[Vim]$^{6+}$ [GVim]$^{6+}$ Simple Session Usage

When using sessions, the most basic thing to do is to save the currently running session (default session when no special session is loaded) to a session file, such that you can load it again later when you need to. The main command to use is:

```
:mksession FILE
```

or if you only want the save the current view, then:

```
:mkview FILE
```

FILE is the name of the file you want to save your session or view to. If no filename is given, Vim uses a file called `Session.vim`, which it puts in the current work directory.

If you have previously saved a session with the same filename, you can add a `!` after `mksession` to make it overwrite the file.

When working with views, you can have many different views at the same time. If every view were saved in the current working directory, then it would be filled up with view files. To prevent this you can tell Vim where it should place the view files with the following command:

```
:set viewdir=$HOME/.vim/views
```

In the above case, you will set it to store the view files in a directory called views, which is placed in your `$HOME/.vim/` directory.

So as an example, we could say that you have three windows open and just before closing Vim you do:

```
:mksession
```

Then the next time you want to open Vim with the same session, you simply start Vim with the command-line argument `-S`, e.g.:

```
vim -S Session.vim
```

Now Vim will be started with the same settings as when you saved the session. Alternatively, you can open Vim as you normally would, and then use the following command to load the session file:

```
:source Session.vim
```

In the case of views, instead you can use:

```
:loadview View.vim
```

Loading a session can change the entire layout of the editor; loading a view will only change the layout of the active window.

If you want Vim to remember settings like cursor placement and folds, when moving between multiple folders, then you can add the following to your `vimrc` file:

```
set viewdir=$VIMHOME/views/
autocmd BufWinLeave * mkview
autocmd BufWinEnter * silent loadview
```

A view of the buffer is saved whenever you show another buffer in the same window, and restored when you show the buffer in the window again.

The trick is to add commands for saving a session when quitting Vim, and restoring the session when opening Vim. This way you can open and close Vim without losing the settings, list of open files, etc. You can do this by simply adding the following commands to your `vimrc` file:

```
autocmd VimEnter * call LoadSession()
autocmd VimLeave * call SaveSession()
function! SaveSession()
   execute 'mksession! $HOME/.vim/sessions/session.vim'
endfunction

function! LoadSession()
   if argc() == 0
      execute 'source $HOME/.vim/sessions/session.vim'
   endif
endfunction
```

If you now close Vim, then it saves a session file in `$HOME/.vim/sessions/session.vim`.

Depending on how you open Vim, it either opens the file specified on the command-line or reopens the latest session, for example:

`vim file.txt` : This opens Vim without loading the last session

`vim` : This opens Vim with the last session loaded. Previously opened files are reopened.

If you want to store additional settings besides what Vim stores in the session file, you can add an extra session file. This is done by creating a file named like your session file, except that the `.vim` extension is replaced by `x.vim`. For example, `Session.vim` has the extra session file `Sessionx.vim`. The extra session file should be placed in the same folder as the session file itself. You can then add all the Vim commands you want to in this file, and these will be executed once the session file has been read.

Satisfy your own Session Needs

It is not always that you want everything saved in your session. Sometimes, it might just be the files you had open that you want to save info about. Other times you want to store every single piece of information you can about a session. Fortunately, Vim gives you a way to set up what you want it to save in a session file.

The setting you should work with is called *sessionoptions* and can be set with:

```
:set sessionoptions=OPTIONS
```

`OPTIONS` is a comma-separated list with one or more of the following options:

- **blank** Save empty windows.
- **buffers** Save info about all buffers including hidden and unloaded buffers.
- **curdir** Save information about current work directory.
- **folds** Save information about folds in the buffer contents.
- globals Save information about global variables. Only variables starting with an uppercase letter and of the type String or Number will be saved.
- **help** Save the help window.
- localoptions Save info about local options and mappings you have created for a single window.
- **options** Save all options, both local and global.
- resize Save info about the size of the UI window (lines and columns).
- sesdir If set, the current directory is the place where the session file is saved (cannot be used when `curdir` is also set).
- slash Change backslashes in all paths to slashes (make Windows paths Unix compatible).
- **tabpages** Save information about all tab pages and not only the active one, which is default without this option.
- unix Use Unix line endings, even on Windows systems.
- winpos Save information on where the UI Window was placed on the screen.
- **winsize** Save the size of all open windows.

The options marked with **bold** are the ones Vim has turned on as per default. Instead of setting the entire list of options, whenever you want to add or remove a single one, you can instead use the '**+=**' and '**-=**' operators. If you, for example, have the default options but would like to have *winpos* added to the options, and also have *folds* removed, then simply do:

```
:set sessionoptions+=winpos
:set sessionoptions-=folds
```

You can see which options you have in your sessions options with:

```
:echo &sessionoptions
```

You can in fact see any of the settings in Vim by simply using the **:echo** and adding an **&** in front of the setting's name, for example, **:echo &somesetting**

[Vim]$^{6+}$ [GVim]$^{6+}$ Sessions as a Project Manager

You might sometimes want to use session files as a sort of primitive project file with information about some project you are working on. So when working on a project and having a lot of files and windows open, you simply use:

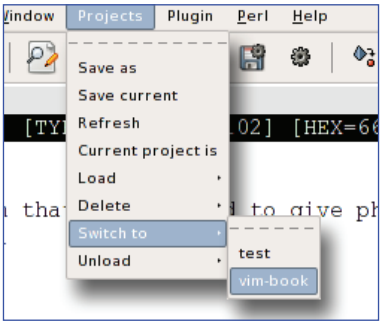
```
mksession!
```

to save the current session to the `Session.vim` file in the folder that you are working from. It would then be ideal if Vim automatically loaded the session for the project if there was a project file (`Session.vim`). So why not make this possible? Simply add the following to your `vimrc` file:

```
silent source! Session.vim
```

Now if there is a `Session.vim` file in the folder where you start Vim, then it will load it. So as long as you keep the session file in the project directory, you can easily reload the project in Vim over and over again—just remember to save the the session again if you open new files or change the windows/buffers.

This is just a simple way of using sessions as a project manager, but it can be made a lot more advanced. Wenzhi Liang has created a practical script that adds a *Projects* menu item to Gvim. In this menu, it is possible to save the current session as a named project. Later, you can restore the project (which is now available directly in the menu) or you can switch between projects with a single click of a menu item. If you don't need a project anymore, then you simply choose to delete it via the menu.



You can find the latest version of the script and read more about it on this homepage:

`http://www.vim.org/scripts/script.php?script_id=279`

Note that the script demands that you have both Perl and Bash installed on your computer.

Registers and Undo Branching

You might know the feeling when you delete or cut something from your text, and realize later that you needed that text elsewhere. If you have already copied/cut another piece of text, then normally the old text is gone because the clipboard normally only has room for one piece of text—not in Vim.

In Vim, you have two tools that can help you when you modify your text and need to keep track of deleted/copied text or changes to the text in general:

- **Registers**
  Registers is sort of an advanced clipboard with multiple buffers for storing your clippings, deleted, and copied text.
- **Undo Branching**
  Undo Branching is a simple form of version control built into Vim. It gives you the possibility to roll back changes to a file until a certain time or number of changes. If you later regret undoing some changes, you can go back and find an undo branch containing those exact changes.

The next two sections will tell you more about how to use both registers and undo branching in your daily work. After reading them, you will easily see why these tools are very strong tools and how they can help you in your daily work in Vim.

[Vim]$^{6+}$ [GVim]$^{6+}$ Using Registers

In many programs and operating systems, you only have access to a single clipboard for text you cut or copy. This is not the case with Vim, because here you have access to not one, not two, but nine different clipboards—or register types as they are called.

Some of the register types overlay each other's working area while others have a very unique purpose. You can use the registers in connection with a range of commands and movements like yank, delete, and paste. The registers are all named with **"** in front of the name like **"x**. So let's look at how to use a register. Let's just say that we use the register called **"x** in the examples. What **x** actually should be will be explained later.

To store a piece of text that you want to copy into a register, you can use the normal y for yank, except that you start out by telling it where to yank it to:

"x y (or `"x yy` if you want to copy the entire line)

the same is the case when cutting text with the x command:

"x x

or when deleting text with the d command:

"x d

So now you have the text stored in the register **"x** and you want to paste it again. You can simply use the `p` (before cursor) and `P` (after cursor) commands to paste the text and just start out by telling Vim which register to paste:

"x p or **"x P**

If you have forgotten which register you used, then you can simply type in the command:

:registers

So now you know how to use the registers with the basic commands in Vim, and it is time to look a bit further at the different register types. The following sections describe each of the nine types of registers.

The Unnamed Register

The unnamed register is called so because it is accessed via `""`, hence resembling the empty string or no name. Vim automatically fills this register whenever some text is yanked with `y` or deleted with one of the following commands: `d` (delete), `c` (delete and go into insert mode), `s` (substitute), or `x` (cut). What this register does is point to the last used register, which also means that it will still work even if you use a specific register when deleting/yanking a text—for example `"xdd` will fill both register x and the unnamed register.

If you paste some text with `p` or `P` without specifying any register, then it actually gets the text it pastes from the unnamed register.

The Small Delete Register

Whenever you delete less than one line of text, Vim will move it into a very specific register—the Small Delete Register (`"-`), only exception where this is not the case is when you specify another register to use.

The Numbered Registers

The numbered registers are named `"0`, `"1`, `"2` and so on up to `"9`. They can be split into two types. The first type is register `"0`, which always contains the last deleted (`d` or `x`)/changed (`c`) text. When you delete or change something new, then register `"0` is overwritten with the new text.

Like register `"0`, register `"1` also contains the last changed/deleted text. There is, however, the difference that register `"1` will not be updated if another register is specified, or if the text is less than one line long (the small delete register is then used). For compatibility with vi, there is, however, an exception where register `"1` is used, no matter what the length. This is if one of the following movement commands is used in your change/delete: `%`, `(`, `)`, `{`, `}`, `` ` ``, `/`, `?`, `n`, `N`

Unlike register `"0`, the contents of register `"1` is not deleted whenever new text is added to it. Instead, it is moved to register `"2`. If register `"2` was already full, then this text is moved to register `"3` first, and so on until register `"9`. The contents of register `"9` will be overwritten whenever any new content is added to it. This way the registers `"1` to `"9` can function as a delete/change history such that you can get access to earlier deleted text, even if you have deleted new text more recently.

The Named Registers

There are two types of named registers—`"a` to `"z` and `"A` to `"Z`.

If you use the lowercase registers like `"a`, then they work like a normal register that you can copy deleted or changed data into. When new text is added to a register, then the old contents is discarded.

If you instead use the uppercase registers (for example, `"A`) then the previous contents of the register is not deleted, but the new text is instead appended to the register.

If you add the value '>' to your *coptions*, then the appended text in uppercase registers is split by a newline: `:set coptions+='>'`

Since you have the complete control over the named registers, they are most likely the type of registers you will get familiar with first.

The Read-Only Registers

There are four different read-only registers. What makes these registers so special is that only Vim has access to them. You only have access to pasting them with the normal `P`, `p`, or `:put` commands. The contents of the read-only registers are quite different:

`"%` : This register always contains the name of the file in the currently active buffer.

`"#` : This register always contains the name of the previous file in the currently active buffer—also called the alternate file.

`".` : This register always contains the last inserted text. You will therefore be able to repeat the last inserted text by executing the normal mode command `".P`.

`":` : This register contains the command you last executed on the command-line. If you repeat a command from the history, then this register will not be overwritten with the command. You will have to write at least one character of the command in order to get it stored in the register.

The Selection and Drop Registers

This register type consists of three registers: `"*`, `"+`, and `"~`. The registers are used to store and retrieve the text you have selected in Gvim. The `"*` register actually accesses the clipboard of your windowing system. If you use Microsoft Windows, then you wont feel any difference between using `"*` and `"+`. On Linux, however, there is a difference between the two registers because the Clipboard in **X11** (the

windowing system) has not only one selection register, but three. The contents of the **"+** register is any text you have selected. It is typically inserted by pressing the middle button on the mouse. The contents of the **"*** register is, however, only altered if you actually tell Vim to yank the text.

These registers can be accessed from any GUI application, and are part of the normal copy-paste procedure you know from your daily work.

The last register in this group is the drop register **"~**. This register contains the last selection that has been dropped into Vim. So, if you select some text in another application and drag it to your Gvim window to drop it there, then **"~** will contain this text.

The Black Hole Register

As the name of this register indicates, this register works like a black hole—everything that goes into it never comes out again. This register is used if you want to completely delete some text and don't even want a record of it in any register. The black hole register is **"-** and can be used like for example **"-x** or **"-dd**. If you try to read out the text you have just written to this register, you will see that no matter what you do, it doesn't return anything.

Search Pattern Register

Whenever you do a search with the **/** command, the pattern you are searching for is automatically added to the search pattern register. The register name **"/** is easy to remember because it resembles the search command **/** and just has the quotes added to show that it is in fact a register. Vim uses this register when you have **hlsearch** (highlight search pattern) turned on. You can use this to your advantage because, as this is a register, you can just change its contents to get **hlsearch** to highlight something different. To change the contents of the register without doing a new search, you can simply use:

```
:let "/ = PATTERN
```

where **PATTERN** is what you want **hlsearch** to highlight.

The Expression Register

The expression register is the last register type in Vim. Calling it a register is, however, not exactly true because it does not store text as the normal registers do. You can not even write to it. Instead, it opens up the possibility to get access to the command line, execute an expression, and the get the result returned as if it were already stored in the register.

You get access to the expression register by simply typing its name **"=.** After pressing the equal sign, the cursor will be moved to the command line. You can see that you are working inside the expression register if there is an equal sign as the first character on the command line. You can now write the expression you want the result of, and then end by pressing the *return* key, or alternatively press *Esc* to return without executing the expression. If you press the *return* key without writing any expression, then Vim will find the latest expression executed and then use this instead. The expression needs to be valid and return a string. If the result of the expression is a number, then Vim automatically converts it into a string, but if you are unsure what type the result has, then simply use the `string()` function to convert the result before returning it.

Look at **:help expression** to see how to put together a valid Vim expression.

[Vim]$^{7+}$ [GVim]$^{7+}$ Using Undo Branching

We all know the feature where we can undo changes that we have done to a text. Vim has taken it a bit further and added the concept of branching to this. In this recipe, we will look at just what undo branching is, and how it can help you in your work.

Let's start by defining what an undo branch is, in Vim. Let's say that you have a file where you have applied a range of changes. At some point, you realize that the last four changes were wrong and that you do not need them at all. You execute undo (**u** in normal mode) four times in a row (or press **4u**) as you normally would, and then the last four changes are gone. Now you see that you need to make an extra change to the file, and you add this also. Normally, your four undone changes would be gone, but not in Vim.

When you reverted the four changes, and added the new changes, you actually added another branch to your undo branch tree.

In one branch you have the four changes you reverted, and in the other branch you have your most recent change. You could go on like this and add other branches to your undo branch tree. At this point, it might be nice to get an overview of the branches your currently have. This is done with the following command:

```
:undolist
```

This will get you a list that shows you three pieces of information about each branch—the change number (used to identify a branch), number of changes in a branch, and the time of the branch creation. It could look like this:

```
number changes   time ~
6      5         12:12:11
11     8         14:01:15
```

If you want to go to one of the specific change numbers, then simply use:

:undo *N*

where *N* is the change number.

You could also move backwards in the list of changes using the following normal mode command:

g- (use g+ if you want to move forward instead).

So what is the difference between using **g-** and **u** to go back through the changes? Let's visualize it with an example.

Write the following text in Vim:

```
My name is Jim
```

Then go to the **J** and press **x** three times to delete the name Jim. You now have:

```
My name is Jim
My name is im
My name is m
My name is
```

Now you realize that your name is in fact Jimmy, so you undo the change:

```
My name is m
My name is im
My name is Jim
```

You now have one branch with the deletion of the name Jim. Now change the name to Jimmy:

```
My name is Jimm
My name is Jimmy
```

But wait! Your name is actually Kim and not Jimmy. Jim was close to Kim, so let's undo back to this place using `u` and change the **J** to **K**

```
My name is Jimm
My name is Jim
My name is im
My name is Kim
```

Another branch was added for when you were undoing the change from Jim to Jimmy.

Now let's go back through the changes with multiple executions of `g-`

```
My name is Kim
My name is im
```

(Vim changes to new branch)

```
My name is Jim
My name is Jimm
My name is Jimmy
```

(Vim changes to new branch)

```
My name is
My name is m
My name is im
My name is Jim
```

Now let's compare this to using `u` for undoing the changes:

```
My name is Kim
My name is im
My name is Jim
My name is Jimm
My name is Jimmy
My name is Jim
```

As you can see, the `u` command only takes you directly through the changes that are not in branches while `g-` takes you through every single change in every branch.

So basically, undo branches can give you access to any text state your file content has had.

Instead of going step by step through the changes in all the branches, you could instead jump a time slice back in your 'edit time.' For this Vim has two commands that jump different time slices, back and forth in the undo history depending on the argument. The commands are like this:

```
:earlier Ns
:earlier Nm
:earlier Nh

:later Ns
:later Nm
:later Nh
```

Where *N* is a number of seconds (`s`), minutes (`m`) or hours (`h`) you want to jump back/forward in time. If you use the `:undolist` function, then you can see the time of the changes and from this you can calculate how far back/forward to jump.

It might take some time to get used to having the undo branches, but when you have gotten used to them, they will help you a lot in your work.

[Vim]$^{6+}$ [GVim]$^{6+}$ Folding

Often when you work with large files, especially code, it can be hard to get a good overview. In Vim there is a special feature that helps you get around this—folding text blocks into folds. In this recipe we will look at how to use folds to make your code easier to overview.

A fold is a way of 'folding' a range of lines (e.g. a function scope) into one single line without losing the contents. An example could be the following code:

```
function myFunction(){
   var a = 1;
   var b = 0;
   var c = a+b;
   return c;
}
```

If this is folded, then it could be set to look like:

```
+-- 6 lines: function myFunction(){ ------------------------------
```

In this case the folding follows the syntax of the code and uses the { } to figure out where to do the folding. Besides using syntax, Vim can also do folding according to the following:

- Manual fold marks—manually mark fold (see `:help fold-manual`)
- Indent folds—use indentation as fold indication (see `:help fold-indent`)

- Expression folds—use an expression to find folds (see **`:help fold-expr`**)
- Syntax folds—use syntax as fold indication (see **`:help fold-syntax`**)
- Diff folds—fold unchanged text (see **`:help fold-diff`**)
- Marker folds—insert markers in text as fold indication (see **`:help fold-marker`**)

Which type of fold indication to use depends on the type of the text you are working on, and also what you find to be the best for you.

So let's look at how you actually do the folding. The first thing to do is to actually activate the functionality:

```
:set foldenable
```

Now Vim knows that it should watch out for folding commands when in normal mode. There is a range of commands you can use to open and close folds, but the primary ones are:

`zc` : close a fold

`zo` : open a fold

`zM` : close all folds

`zR` : open all folds

If we take the Syntax folding method as an example, then you just need to place the cursor somewhere in the area you want to fold (e.g. inside a function scope) and then you go into normal mode and press **`zc`** to close the fold. Now you will see the function get folded into a single line. In the following figure you can see both folded and unfolded code mixed together:

```
 79 void show_save_graph_as_dialog( GtkMenuItem* menu_item, gpointer user_data ){
 80     char* filename_selected;
 81     gint response;
 82     GtkWidget* save_graph_as_dialog = glade_xml_get_widget( xml, "save_graph_as_dialog" );
 83     response = gtk_dialog_run( (GtkDialog*) save_graph_as_dialog );
 84     gtk_widget_hide( save_graph_as_dialog );
 85
 86 -- if( GTK_RESPONSE_OK == response ){...6 Lines... }
 92 }
 93
 94 void show_save_rendering_dialog( GtkWidget* widget, gpointer user_data ){...16 Lines...}
110
111 void show_render_window( GtkWidget* widget, gpointer user_data ){
112     GtkWidget* render_win;
113     GtkWidget* render_drawing_area;
```

If you don't want to remember commands for both opening and closing folds, then simply bind a key to toggle a fold open or closed. This could be for example, space: **`:nnoremap <space> za`**.

If you find that the design of the folded line is not giving you the information you need, then you can change it easily. Simply change the value of the *foldtext* option to point to another function that returns the line you want:

```
:set foldtext=MyFoldFunction()
```

The function could look like the following:

```
function! MyFoldFunction()
   let line = getline(v:foldstart)
   " cleanup unwanted things in first line
   let sub = substitute(line, '/\*\|\*/\|^\s+', '', 'g')
   " calculate lines in folded text
   let lines = v:foldend - v:foldstart + 1
   return  v:folddashes.sub.'...'.lines.' Lines...'.getline(v:foldend)
endfunction
```

This function changes your folded line to look like:

```
+--function myFunction(){...6 Lines...}----------------------------
```

You can see that the function used three different variables whose names start with `v:`. These are variables set by Vim and contain:

- `v:foldstart` Line number of first line in fold
- `v:foldend` Line number of last line in fold
- `v:folddashes` Contains a dash (-) for each level of folding a fold is

The last variable gives a fast indication of how many levels down you are in the folding tree. If you have:

```
if (x != y){
    if (y !=x){
       print "x not y";
      }
}
```

Then `v:folddashes` in the innermost `if` will contain '--' (second level) but in the outermost `if` it will contain '-' (first level).

The dashes at the end of the fold line are automatically added. If you want a different character instead of a dash, then this can of course also be changed. If you, for example, want to change it to equal signs (=) instead of dashes then simply do:

```
:set fillchars=fold:=
```

You might think *why dashes?*, but there is actually quite an obvious explanation. Vim has another fold setting called *foldcolumn* that tells it how many columns to the left of the text it should use for fold information. What it actually uses the columns for is to draw an ASCII fold-tree where the dashes in the folds are the leaves. For example:

```
| some text
+- a first level fold --------------------------------
|
-  beginning of open fold.
2  indication of fold level
2    - do -
- open fold beginning level 1
+-- a second level fold.
| more text
| more text
```

As you can see, it basically draws a tree which looks like:

```
|
+-
|
+--
|
+-
+--
```

To set how wide the tree should be, simply use:

```
:set foldcolumn=N
```

where *N* is a number between 0 and 12. A value of 1 or 2 is recommended only if you have a few levels of folds, else you should use 3-5.

You can execute a command on all folds that are either open or closed:
`:folddoopen cmd` – execute cmd on all line not in a closed fold.
`:folddoclose cmd` – execute cmd on all lines in closed folds.

[Vim]$^{6+}$ [GVim]$^{6+}$ Simple Text File Outlining

Sometimes when you write a simple text file in Vim and you suddenly realize that it has grown from simple to long and chaotic, and you could really use a good way to do outlining of the text. In this recipe, we look at how to use folding in Vim to do just that, especially if you think quite a bit about how you structure your text.

Let's say you have a text like:

```
Chapter 1
Section 1 - Vim help
here is some text about the vim help system.
Section 2 - vim scripts
this section contains  info about vim scripts.
```

Now you would like to fold the text such that only the section headers are shown. If you use manual folding (**`:set foldmethod=manual`**) this is quite simple. You simply need to mark all lines in a section (including header line) and then press **`zf`**. Now you have created a fold containing those lines. If you start from the outside and go inwards, then you can have say 'Chapter 1' as first level, and each section as fold level 2. If you then closed the section folds, it would look like:

```
-Chapter 1
+Section 1 - Vim help  (2)
+Section 2 - vim scripts  (4)
```

To make it look exactly as the above example, you will need the following settings:

```
:set foldcolumn=1
:set fillchars=fold:\  "there is a space after the \
:set foldtext=getline(v:foldstart).'    ('.v:foldstart.')'
```

As you can see, this looks a lot like the Table of Contents in a book, but the difference is that this is just a simple text file. As long as you add new text to the sections by appending it to the previous lines (newlines are OK), Vim will still know that your added text is part of the fold.

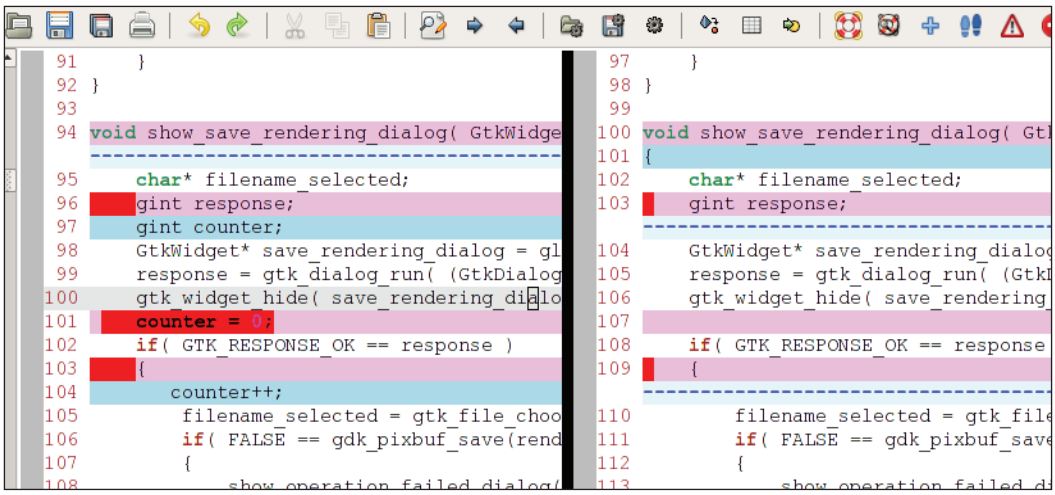
If you later want to delete a fold, then you simply mark the text again in visual mode and then press **`zd`**.

If you want to use a different formatting of your text (e.g. having '= =' around the section headers), you can do that—as long as you mark your own folding areas, then you won't feel a difference.

[Vim]$^{6+}$ [GVim]$^{6+}$ Using vimdiff to Track the Changes

Sometimes you have multiple versions of the same file—maybe they are the same and maybe they are not. On Unix systems, there has been a program called *diff* available for many years (first release in 1974), but on other operating systems you most likely do not have this. This program gives the user an output that shows the differences between two files. Vim has a solution for you that gives this functionality and even presents it in an easy-to-overview format, *vimdiff*. This recipe will show you how to use Vim to get an overview of changes to your file compared to other versions of the same file.

Vimdiff is actually a built-in diff program, which uses colors to show the differences between two files (shown in two windows in Vim split vertically or horizontally). The following figure shows how a vimdiff session could look like.



There are several different ways to activate vimdiff. On many systems, a program shortcut is made that is called vimdiff. In those cases, you can simply use:

```
vimdiff file1 file2
```

This is actually a shortcut for writing

```
vim -d file1 file2
```

You will need to supply at least two versions of the file, but up to four versions of the file are supported.

If you are already in an open Vim session, then you can of course also activate the diff mode in Vim. To do so, you will have to use one of the following commands:

- **`:diffsplit filename`**
  Split the window horizontally and show the file "filename" in both of the windows. All diff-specific settings are set for both windows.
- **`:vert diffsplit filename`**
  Split the window vertically and show the file 'filename' in both of the windows. All diff-specific settings are set for both windows.
- **`:diffthis`**
  Add the current window to the existing set of diff-enabled windows. This could be used if you want to diff a file again with yet another version.

An example could be to see which changes you have made to the currently active file since the last time Vim saved a backup of the file (those copies of the files saved with `~` after the name). If you are working on a file called `main.c`, then you could at any time save the file and then execute the following command:

```
:vert diffsplit main.c~
```

Now the current window is split vertically into two, and you will be able to see the changes you have made to the file marked with colors (depending on your color theme).

Maybe it wasn't actually you who made the changes to one of the versions of the file and maybe another developer just sent you a patch for your file. But don't worry, Vim can still give you a nice diff view of the changes the patch makes to your version of the file. Simply open your version of the file and then with it in the active window, execute this command:

`:vert diffpatch patchfile` (or just **`:diffpatch patchfile`**)

where *patchfile* is the patch the other developer sent you. Now Vim will open another window with your file in, and then apply the patch to it. Then it will set up all the diff settings for the windows, such that the changes are colored.

Navigation in vimdiff

Navigation in the vimdiff windows is a bit different than in the normal Vim windows.

When for instance you scroll through the file in one of the windows in the diff split, you will see that the other part of the diff windows is also scrolled. In fact, the windows follow each other in a way such that the current line in one window is

the same line (if available) as in the other window. This is called *scrollbind* and can be turned on and off with:

```
:set scrollbind
```

and:

```
:set noscrollbind
```

You can edit the files in the diff windows and you will see that the diff colors are updated accordingly; if not, then try executing:

```
:diffupdate
```

When you are in one of the diff windows, you might want to jump fast between changes done to the file. This is done with the following commands in normal mode:

`[c` : Go to start previous change

`]c` : Go to start of next change

This way you can navigate between relevant areas in a file and get a good idea of which changes have been made to the file.

When the cursor is placed in one of the changes you can see in the vimdiff window, then you might realize that this change is also needed in the other version of the file. So now you could copy the lines in the change, and insert them into the other file in the right place. But why use time on this, when Vim has made it a lot easier for you? Vim has a function that simply puts the change from one file version into the correct place in another version of the same file. The command is:

```
:diffput
```

This command should be executed when the cursor is placed in the change you want to move to the other file version. If on the other hand you are in the file without the change, then you could either move to the other file and then put the change back into the first file—or use the Vim way of doing it, with the following command:

```
:diffget
```

Alternatively, you can use `do` in normal mode for getting a change and `dp` for putting a change.

Read more about vimdiff in `:help vimdiff`

[Vim]$^{6+}$ [GVim]$^{6+}$ Using Diff to Track Changes

From the previous section, we know how to use vimdiff to make a diff of different versions of the same file. But what if you really just want to know what you have changed in the current buffer before saving it? This recipe gives you a little trick to check for changes between the version of the file you have on your hard drive and the one you have in the buffer. It will, in other words, show you what you have changed in the buffer since the last time you saved.

What you have to do is to add the following function to your `vimrc` file:

```
function! DiffWithFileFromDisk()
  let filename=expand('%')
  let diffname = filename.'.fileFromBuffer'
  exec 'saveas! '.diffname
  diffthis
  vsplit
  exec 'edit '.filename
  diffthis
endfunction
```

The function stores a temporary copy of the file you have in the current buffer (including latest changes) and then it diffs this file against the version of the file you have on your hard drive.

To call the function you use the following command:

```
:call DiffWithFileFromDisk()
```

or if you want to access it faster, then you can bind a key to the command like this:

```
:nmap <F7> :call DiffWithFileFromDisk()<cr>
```

This binds the *F7* key to the function call, and you then just have to go into normal mode and press *F7* to see the changes marked in diff mode.

Now you can quickly and easily go through the changes that you have made to the file and check if all of them are important.

[Vim]$^{6+}$ [GVim]$^{6+}$ Open Files Anywhere

System administrators and web developers have one thing in common, which can be a big point of annoyance for both of them, if they don't have an easy way to get around it. They both work with files that are most often placed on remote servers of some sort.

The system administrator mostly gets around the problem by logging in on the remote server via for example an SSH (secure shell) connection and then edits the configuration files etc. directly on the server.

The web developer on the other hand gets around the problem by uploading and downloading the files between the remote computer and his or her local computer using an FTP client or by using systems like Webdav.

But what if it didn't have to be that way? What if they could simply edit the files on the remote system directly from their local system? In Vim, this is in fact possible, without any further extensions besides what normally comes with it. Vim has a system called **netrw** (for net read/write), which comes in handy whenever you want to work with files on remote servers. Let's get right to it and start with an example.

Imagine a web developer, John, who has his homepage placed on a remote system called `remote.server.com`. He wants to edit his `index.html` file, which resides in the `public_html/` directory in his home directory on the server. In this case, the web developer could simply open the file in Vim like this:

```
vim ftp://john@remote.server.com/public_html/index.html
```

Vim recognizes that it needs to use the FTP protocol and then connects to the FTP server on *remote.server.com* using *john* as username. If a password is needed to log in, then Vim will prompt you for it. Vim transfers a temporary copy of the file to the local machine and then lets you edit it like any other file. Only difference is that whenever you save the file, it is saved onto the remote server also.

If he had already opened Vim, then John could instead open the site with one of the following commands:

```
:Nread ftp://john@remote.server.com/public_html/index.html
:Nread remote.server.com john PASSWORD public_html/index.html
```

Change **PASSWORD** to be the password you want to use for the FTP server. Besides reading a file from a remote server, you can also open a local file and write it to the remote server—or go about opening a file on one remote server and save it onto another. The command for writing a file to a remote FTP server is:

```
:Nwrite ftp://user@server/path/to/filename
:Nwrite server user password path/filename
```

The format of the arguments from :**Nread** and :**Nwrite** can differ from protocol to protocol. Use **:Nread ?** and :**Nwrite ?** to get help on the exact syntax.

Besides the FTP protocol, Vim supports many other protocols:

- SCP
- SFTP
- RCP
- HTTP (read-only)
- DAV
- rsync (read-only)
- fetch (read-only)

To use these, you simply change the 'ftp' part of the previous example to one of the other protocol names (in lowercase).

There is, however, a catch. Vim is dependent on external command-line programs in order to use the different protocols. On Linux systems most of these programs are available by default, but on Mircosoft Windows, only FTP is available. You can find a list of the external programs that Vim uses as a default and explanation of how to change them in the help system:

```
:help netrw-externapp
```

Besides reading and writing files, Vim is also able to give you a directory listing, such that you can use it for finding the right remote files to edit. You just have to point your **`:Nread`** to a directory instead of a file. For example:

`:Nread scp://`*`user@server/some/directory/`*

You can select any file in the directory listing and it will then be opened in Vim as if it was a local file.

If you use Linux, then you can store usernames and passwords for remote sites in a **`.netrc`** file in your home directory. See **`:help netrw-netrc`** for more information.

[Vim]$^{6+}$ [GVim]$^{6+}$ Faster Remote File Editing

So now you have learned how to work with remote files directly, and you will at some point get in a situation where you have several remote files open at the same time. But then you suddenly hit an annoying situation—you need to re-log in every time you move to another buffer with for example **`:bufferprev`** and **`:buffernext`**.

As a default, Vim tries to reload the contents of a buffer whenever it is shown in a window. This means that if the file in the buffer is a remote file, then Vim will need to log in again, in order to check if the file should be reloaded.

But is that actually necessary? If you can live with the fact that a remote file can be edited by another person while you are editing it remotely (without you being notified about it), then you can trick Vim into not reloading the file.

Each buffer has a set of options that tells Vim what to do with the particular buffer in different situations. One of the options is *bufhidden*, which tells Vim what to do when a buffer is hidden (not shown in a window). This option is normally not set to anything, but if you set it to 'hide', then you tell Vim to just hide the buffer when it is not in a window, and then just show it again when you show it again in a window. Simply, add the following to your `vimrc` file:

```
:set bufhidden=hide
```

And that's basically it. Now you won't have to re-log in whenever you switch buffer and will feel just as if you were editing a local file.

Summary

In this chapter we have been looking at how to improve our daily work in Vim. Many approaches have been touched, each with a specific area to optimize.

We started out by looking at how to use templates to minimize the amount of text to enter. The first time, our templates were simple ones that used the abbreviation functionality in Vim to emulate the insertion of a template. Next, we improved on our template system by creating template files for specific file types and thereby made it possible to insert an entire skeleton into a programming file.

After the templates, we moved on and looked at another way to minimize word entry time—auto-completion. Different approaches for auto-completion were discussed and a function was proposed for binding all auto-completion to a single key—*Tab*.

By recording a list of commands, you can get around the boring task of entering the same commands over and over again. We looked at how to do Vim macro recording and use this to change a simple text file into a fine HTML file in a matter of minutes.

Next up, was Vim Sessions and how to use these for everything from saving the look of a window, to using sessions as a full-fledged project manager.

With registers, you have the possibility to use not one, not two, but nine different registers/clipboards.

Using folding of the text, you can get a better overview of a file because every unnecessary part is hidden in a fold. You can even use folding for creating simple outlines of a text file.

So now the file is changed, but what has actually been touched in the file? We looked at how to use the built-in diff functionality in Vim. This gives you an improved overview of where and what you have changed in the active files and you can even undo or add new changes while still maintaining the good diff overview.

Working with files on the local machine is one thing, and working on remote files, another. In Vim, it is possible to work in and navigate remote files directly. This way you won't feel whether the file is local or remote.

After reading this chapter and playing around with the recipes, you should soon feel how your workday has improved.

5
Advanced Formatting

Often the simplest modification to a text or a piece of code is what changes it from being obscure to being easily readable. In this chapter, we will look at some of the simple tricks you can use to format the text you are working on—no matter whether it is plain text or code.

This chapter will have recipes in three categories:

- Text formatting
- Code formatting
- Using external formatter programs

After reading this chapter, you should have a good idea about what is possible and what is not, when it comes to text formatting in Vim.

Formatting Text

Even though most people prefer graphical word processors like Microsoft Word or OpenOffice Writer when they want to write plain text, there are still times where an editor like Vim will do it just as well. In the following sections, we will look at how to use the strengths of Vim when formatting normal text.

[Vim]$^{6+}$ [GVim]$^{6+}$ Putting Text into Paragraphs

This recipe is probably one of the simplest in this book, but at the same time one of the most versatile when it comes to formatting plain text. Imagine that you are writing a piece of text and just keep on writing without bothering about changing lines or formatting the text. At some point, you might end up with one or more very long lines and conclude that you should start formatting the text. At this point, you have two choices:

1. Go through the text and add the formatting manually
2. Use the strength of Vim and format the entire paragraph with one command

Obviously, the latter option is the fastest one and at the same time the formatting will be consistent. So let's look at which command to use for this:

`gqap`

This command is actually a combination of a command and a movement; specifically:

`gq` : format everything the next movement moves over

`ap` : 'a paragraph' moves over the current paragraph

In other words, the command simply tells Vim to move over the current paragraph and format it. A paragraph is defined as all lines between two empty lines. In order to change to another paragraph, you simply add an empty line.

The formatting that Vim adds to the text is basically nice line breaks such that the lines are not longer than a specific length (split correctly between words).

The text formatting width is defined in the Vim option *textwidth* such that, if you want a maximum of 80 characters on each line, then you would need to have the following in your `vimrc` file:

```
:set textwidth=80
```

If the option is set to `0`, then Vim sets it to the width of the window—however, never more than the number of characters defined in the `textwidth` setting.

How Vim formats a paragraph can be set in the Vim option called *formatoptions*. See **`:help 'formatoptions'`** and **`:help 'fo-table'`**.

`gq` can be used together with any movement command, and after performing the formatting, it will place the cursor where it ends (typically at the end of the last line in that particular area). If instead you want the cursor to go back to the place where it originally was before executing the command, then simply change **`gq`** to **`gw`**. If you have the cursor placed at the beginning of the first line and do **`gwap`**, then the cursor will remain there even though the paragraph is formatted.

You can repeat the formatting multiple times by prepending the command with the number of times it should be repeated, for example **`5gqap`** will format the current and the next four paragraphs. If you want to format all the paragraphs in a file, you can do it with the command **`1gqG`**.

This formatting command does not only apply to plain text, but also to any other type of content—and you can decide what formatting it should apply.

You can set any function to be the 'formatter' for any given file format, simply by setting it in the Vim option called **`formatexpr`**. If for instance you work with a C-code source file, then you should simply have the following in your `vimrc` file:

```
:set formatexpr=c#Formatter()
```

This tells Vim that when it opens a file of type c, it should use the function called `Formatter()` in the autoloaded file for the C filetype.

Autoloaded files can be found in your **VIMHOME** in a folder called **autoload.** Files are named as the file type and appended with **.vim**. For example, **VIMHOME/autoload/c.vim** for the C filetype.

A formatting function has three variables that you can use to find the text you have told it to format:

- **`v:num`** The line number of the first line to format.
- **`v:count`** The number of lines to format.
- **`v:char`** This variable holds a character that is going to be inserted. This can be empty.

A simple formatting function could look like this:

```
function! MyFormatter()
   let first = v:num
   let last = v:num + v:count
    while(first<=last)
       call setline(first, '> '. getline(first))
       let first = first+1
    endwhile
endfunction
```

This formatting function takes all the lines it is set to work on, and then prepends the lines with '>' like quoted text in e-mails.

The above formatting function is a very simple one. If it needs to be a bit more advanced, the complexity of the function rises quite fast. This is why the number of publicly available formatting functions is limited to a very few (that are created for very specific purposes).

[Vim]$^{6+}$ [GVim]$^{6+}$ Aligning Text

One of the most basic formatting options in most word processors is the ability to align the text left, right, or center. Some of them can even align justified, such that the text is spread equally along the lines, so that the line endings of all lines are as near to the margins as possible.

Even though this kind of formatting is quite common in word processors, there are very few plain text editors that have this functionality, and Vim is one of them.

Vim supports three types of alignment—left aligned, right aligned, and centered. But before we look at how they work, we have to realize something about why this type of alignment is uncommon in plain text editors.

The fact is that in common text editors, there is no hidden information. The text you see is what you have—no page width, no alignments, nothing.

In word processors on the other hand, there is lots of information hidden in the document, and this tells the editor how to format text as the user wants it.

As this is not possible in text editors like Vim, the user will have to supply the editor with information about the preferred text width, for example, in order to give the editor margins to align against.

With this in mind, let's take a look at the commands starting with how to center a text.

```
:[range]center WIDTH
```

Here **range** is the range of lines you want to center, and width is the number of characters you maximum want on each line. Typically, you select the lines you want to center in visual mode (use *Shift-v* and move the cursor to select lines) and then, start typing in the command. You will see that after pressing `:`, Vim automatically adds the range you have selected as `'<,'>`. This basically means from first selected line (`'<`) to the last selected line (`'>`).

Then, you just have to write center and the width you want the text to have. You can leave out the width if you have set the Vim option *textwidth.*

If the option is set to `0` and you still leave out the width, then Vim just expects you to want a text width of 80 characters. It probably won't take you long to realize that the text is not centered as in a word processor, but simply indented the correct amount, with whitespaces. This also means that whenever you change the text in the centered line, you will have to re-center the text.

The next command is the left-align command:

```
:[range]left INDENT
```

Again, this command needs a range of lines to work on, and if needed, a number of characters to indent all the lines with. This means that you can set the left margin of those lines exactly where you want it.

Finally, there is the right-align command, which aligns the lines with the right margin. Again, there is the problem that Vim does not necessary know the width of the text, and hence you have to supply it with this information. The command is:

```
:[range]right WIDTH
```

The lines are again indented with white spaces such that the line ends are all aligned according to the width you define. As with centered text, you will have to realign the text whenever you change anything in a line because it will grow beyond the right margin.

[Vim]$^{6+}$ [GVim]$^{6+}$ Marking Headlines

When you write documents in a plain text editor, you sometimes need to create your own formatting and markup in order to make the text more readable.

To improve readability, one of the major things that you can do is mark the strings that act as headlines for the sections of text.

In word processors, this is normally done by making the font larger and bold, but in Vim this is not possible because only one font size is allowed in the document. So, Vim users will have to mark the headlines another way.

My personal way of marking a line as a headline is by adding a line underneath it.

An example could be:

```
My Headline
===========
This is the text on the document. It could contain one
or more lines of text.
```

Different types of marks could be used for different levels of headlines:

```
Level1
======
Level2
------
-Level3-
```

To make it easier to add the underlining of the headlines, it can be wrapped in a macro in Vim. This way you don't have to worry whether you have added less/too much underlining of the headline.

A macro for the first two levels of headlines could look like:

```
yypVr=o
```

Broken down in parts this says:

`yy` : Yank current line
`p` : Paste copied line
`V` : Select entire line
`r` : Replace selected characters with the following characters (in this case =)
`o` : Add a new line below the cursor and place cursor on it in insert mode

This macro basically takes the current line (the headline line) and duplicates it. Then it takes the duplicate and replaces all characters in it with some character (`-` or `=` in this case). Finally, it inserts a new line and goes back into insert mode.

In the case of the third headline level, we have to take another approach as it is not an underlining, but rather a appending/prepending of a dash (-headline-). For this a simple substitution could be used.

```
:s/\(.*\)/-\1-/
```

If we again break it into pieces, it consists of three main parts:

`:s///` : The substitution command.

`\(.*\)` : Regular expression that takes all characters in the current line and remembers that this is the search pattern.

`-\1-` : This is the replacement pattern. It tells Vim to insert a dash followed by the first matched sub-pattern (everything between `\(` and `\)`) from the above search, followed by another dash.

Remembering these macros can be a bit hard, but you can easily create Vim mappings such that you can have a shortcut for each of the headline markings, for example:

```
:map h1 yypVr=o
:map h2 yypVr-o
:map h3 :s/\(.+\)/-\1-/<cr>o
```

Now, you can just go into normal mode an press *h1*, *h2*, or *h3* to add the appropriate headline level formatting. If you don't want it to insert an empty line under the headline and go into insert mode, then simply remove the `o` from the end of each mapping.

[Vim]$^{6+}$ [GVim]$^{6+}$ Creating Lists

Bullet lists and numbered lists are common structures in documents. In this recipe, we will look at how to make the task of creating these lists in Vim a lot easier.

Let's start by looking at how we can create a Vim function that takes a range of selected lines and converts them into a bullet list. In this case, a bullet list looks like:

```
* first item
* second item
* third item
```

So, a function that adds * to the beginning of all the selected lines could look like:

```
function! BulletList()
    let lineno = line(".")
    call setline(lineno, "   * " . getline(lineno))
endfunction
```

Looking at this quite simple function, you will see that all it does is get the current line and replaces it with a copy of itself, prepended with a couple of spaces, a bullet (in this case an *), and then a tab space.

Obviously, it only does its work on one line, but if you select a range of lines, Vim will call this function over and over again for each selected line—starting from the top down. This works fine when you need to add the same to every line and don't have to tell them apart.

For numbered lists, however, this is not the case because you have to know how far in the numbers you have gotten.

So let's take a look at a function that converts a range of selected lines into a numbered list—one line for each item:

```
function! NumberList() range
  " set line numbers in front of lines
  let beginning=line("'<")
  let ending= line("'>")
  let difsize = ending-beginning +1
  let pre = ' '
  while (beginning <= ending)
      if match(difsize, '^9*$') == 0
          let pre = pre . ' '
      endif
    call setline(ending, pre . difsize . "\t" . getline(ending))
    let ending=ending-1
```

```
      let difsize=difsize-1
    endwhile
  endfunction
```

This function is a bit more complex, without losing the simplicity of the task it should solve—adding numbers in front of each selected line.

This function does, however, add a little extra to that—it right-aligns the numbers like so:

```
  1 item1
  2 item2
  ...
 10 item10
 11 item11
  ...
100 item100
  ...
```

In order to perform this alignment, it needs to have two issues taken care of:

1. It needs to know the largest number in the list.
2. It needs to be able to work on all lines at once.

The first issue is handled by looking at the line number of the first and the last line in the selected range, and the difference is then the number of lines—and hence the largest number in the numbered list. This is only possible because the second issue is also taken care of, because else the function would only know about the current line.

The solution for this is simply to add the keyword *range* after the function name, and thereby tell Vim that the function will work on the entire range and not just one line.

The function goes through the lines in the range from the last line to the first. Whenever it hits a number that contains only the number 9 (like 99 or 9999) it knows that it has one character less in the number (e.g. going from line 1000 to line 999). Instead of the character it now misses, it simply prepends an extra space to the indentation. This way the numbers are kept right-aligned all the time, no matter how many lines you select to have in your range.

[Vim]$^{6+}$ [GVim]$^{6+}$ Formatting Code

Formatting code often depends on many different things. Each programming language has its own syntax and some languages rely on formatting like indentation more than others. In some cases, the programmer is following style guidelines given by an employer so that code can follow the company-wide style.

So how should Vim know how you want your code formatted? The short answer is, it shouldn't! But by being flexible, Vim can let you set up exactly how you want your formatting done.

The fact is, however, that even though formatting differs, most styles of formatting follow the same basic rules. This means that in reality, you only have to change the things that differ. In most cases, the changes can be handled by changing a range of settings in Vim. Among these, there are a few especially worth mentioning:

- `formatoptions` This setting holds formatting-specific settings (see **`:help 'fo'`**)
- `comments` What are comments and how they should be formatted (see **`:help 'co'`**)
- `(no)expandtab` Convert tabs to spaces (see **`:help 'expandtab'`**)
- `softtabstop` How many spaces a single tab is converted to (see **`:help 'sts'`**)
- `tabstop` How many spaces a tab looks like (see **`:help 'ts'`**)

With these options, you can set nearly every aspect of how Vim will indent your code, and whether it should use spaces or tabs for indentation. But this is not enough, because you still have to tell Vim if it should actually try to do the indentation for you, or if you want to do it manually. It you want Vim to do the indentation for you, then you have the choice between four different ways for it to do it. The following sections we will look at the options you can set to interact with the way Vim indents code.

Autoindent

Autoindent is the simplest way of getting Vim to indent your code. What it does is simply stays at the same indentation level as the previous line. So if the current line is indented with four spaces, then the new line you add by pressing *Enter* will automatically be indented with four spaces too. It is then up to you as to how and when the indentation level needs to change again. This type of indentation is particularly good for languages where the indentation stays the same for several lines in a row. You get autoindent by using **`:set autoindent`** or **`:set ai`**.

Smartindent

Smartindent is the next step when you want a smarter indent than autoindent. It still gives you the indentation from the previous line, but you don't have to change the indentation level yourself. Smartindent recognizes most common structures from the C programming language and uses this as a marker for when to add/remove indentation levels. As many languages are loosely based on the same syntax as C, this will work for those languages as well. You get smart indent by using `:set smartindent` or `:set si`.

Cindent

Cindent is often called clever indent or configurable indent because it is more configurable than the previous two indentation methods. You have access to three different setup options:

- **`cinkeys`** This option contains a comma-separated list of keys that Vim should use to change the indentation level. An example could be: `:set cinkeys="0{,0},0#,:"`, which means that it should reindent whenever it hits a `{`, a `}`, or a `#` as the first character on the line, or if you use `:` as the last character on the line (as used in switch-constructs in many languages). The default value for `cinkeys` is `"0{,0},0),:,0#,!^F,o,O,e"`. See `:help cinkeys` for more information on what else you can set in this option.

- **`cinoptions`** This option contains all the special options you can set specifically for cindent. A large range of options can be set in this comma-separated list. An example could be `:set cinoptions=">2,{3,}3"`, which means that we want Vim to add two extra spaces to the normal indent length, and we want to place {and} three spaces in compared to the previous line. So, if we have normal indent to be four spaces, then the above example could result in code looking like this (dot marks a space):

```
if( a == b)
...{
......print "hello";
...}
```

  The default value for cinoptions is this quite long string:
  `">s,e0,n0,f0,{0,}0,^0,:s,=s,l0,b0,gs,hs,ps,ts,is,+s,c3,C0,/0,(2s,us,U0,w0,  W0,m0,j0,)20,*30" .`
  See `:help 'cinoptions'` for more information on all the options.

- **cinwords** This option contains all the special keywords that will make Vim add indentation on the next line. An example could be: `:set cinwords="if,,else,do,while,for,switch"`, which is also the default value for this option. See `:help 'cinwords'` for more information.

Indentexpr

Indentexpr is the most flexible indent option to use, but also the most complex. When used, indentexpr evaluates an expression to compute the indent of a line— hence you have to write an expression that Vim can evaluate. You can activate this option by simply setting it to a specific expression:

```
:set indentexpr=MyIndenter()
```

Where `MyIndenter()` is a function that computes the indentation for the lines it is executed on.

A very simple example could be a function that emulates the autoindent option:

```
function! MyIndenter()
   " Find previous line and get its indentation
   let prev_lineno = s:prevnonblank(v:lnum)
   let ind = indent( prev_lineno )
   return ind
endfunction
```

Adding just a bit more functionality than this, the complexity increases quite fast. Vim comes with a lot of different indent expressions for many programming languages. These can serve as inspiration if you want to write your own indent expression. You can find them in the folder **indent** in your **VIMHOME**.

You can read more about how to use indentexpr in `:help 'indentexpr'` and `:help 'indent-expression'`.

[Vim]$^{6+}$ [GVim]$^{6+}$ Fast Code Block Formatting

After you have configured your code formatting, you might want to update your code to follow these settings. To do so, you simply have to tell Vim that it should reindent every single line in the file from first line to the last. This can be done with the following Vim command:

```
1G=G
```

If we split it up it simply says:

`1G` : Go to first line of file (alternatively you can use `gg`).
`=` : Equalize lines, in other words indent according to formatting configuration.
`G` : Go to the last line in file (tells Vim where to end indenting).

You could easily map this command to a key in order to make it easily accessible:

```
:nmap <F11> 1G=G
:imap <F11> <ESC>1G=Ga
```

The last `a` is to get back into insert mode as this was where we originally were. So now you can just press the *F11* key in order to reindent the entire buffer correctly.

Note that if you have a programmatic error, like for example missing a semicolon at the end of a line in a C program, the file will not be correctly indented from that point on in the buffer. This can sometimes be useful to identify where a scope is not closed correctly (for example a `{`not closed with a `}`).

Sometimes you might just want to format smaller blocks of code. In those cases, you typically have two options—use the natural scope blocks in the code, or select a block of code in visual mode and indent it.

The last one is simple. Go into visual mode with e.g. *Shift-v* and then press = to reindent the lines.

When it comes to using code blocks on the other hand, there are several different ways to do it. In Vim there are multiple ways to select a block of code, so in order to combine a command that indents a code block, we need to look at the different types and the commands to select them:

- `i{` 'Inner block', which means everything between `{` and `}` excluding the brackets. This can also be selected with `i}` and `iB`.
- `a{` 'A block', which means all the code between `{` and `}` including the brackets. This can also be selected with `a}` and `aB`.
- `i(` 'Inner parenthesis', meaning everything between `(` and `)` excluding the parentheses. Can also be selected with `i)` and `ib`.
- `a(` 'A parentheses, meaning everything between `(` and `)` including the parenthesis'. Can also be selected with `a)` and `ab`.
- `i<` 'Inner `<>` block', meaning everything between `<` and `>` excluding the brackets. Can also be selected with `i>`.

- **a<** 'A <> block', meaning everything between < and > including the brackets. Can also be selected with **a>**.
- **i[** 'Inner [] block', meaning everything between [and] excluding the square brackets. Can also be selected with **i]**.
- **a[** 'A [] block', meaning everything between [and], including the square brackets. This can also be selected with **a]**.

So we have defined what Vim sees a block of code as; now we simply have to tell it what to do with the block. In our case, we want to re-indent the code. We already know that **=** can do this, so an example of a code block re-indentation could look like:

```
=i{
```

Which, if executed at the following code (| being the place where the cursor is):

```
if( a == b )
  {
print |"a equals b";
}
```

would produce the following code (with default C format settings):

```
if( a == b )
  {
    print |"a equals b";
}
```

If on the other hand we choose to use **a{** as the block we are working on, then the resulting code would look like:

```
if( a == b )
{
   print "a equals b";
}
```

As you can see in the last piece of code, the **=a{** command corrects the indentation of both the brackets and the print line.

In some cases, where you work in a code block with multiple levels of code blocks, you might want to re-indent the current block and maybe the surrounding one. No worries, Vim has a fast way to do this. If for instance you want to re-indent the current code block, and besides that want to re-indent the block that surrounds it, you simply have to execute the following command while the cursor is placed in the innermost block:

```
=2i{
```

This simply tells Vim that you will equalize/re-indent two levels of inner blocks counting from the "active" block and out. You can replace the number '2' with any number of levels of code blocks you want to re-indent. Of course, you can also swap the inner block command with any of the other block commands, and that way select exactly what you want to re-indent.

So, this is really all it takes to get your code to indent according to the setup you have.

[Vim]$^{6+}$ [GVim]$^{6+}$ Auto Format Pasted Code

The trend among programmers tells us that we tend to reuse parts of our code—so called patterns. This could mean that you have to do a lot of copying and pasting of code.

Most users of Vim have experienced what is often referred to as the 'stair effect' when pasting code into a file. This effect occurs when Vim tries to indent the code as it inserts it. This often results in each new line to be indented another level, and you ending up with 'a stair':

```
code line 1
   code line 2
         codeline 3
               code line 4
                     ...
```

The normal workaround for this is to go into **paste**-mode in Vim, which is done by using:

```
:set paste
```

After pasting your code, you can now go back to your normal insert mode again:

```
:set nopaste
```

But what if there was another workaround? What if Vim could automatically indent the pasted code such that it is indented according to the rest of the code in the file? Vim can do that for you with a simple paste command.

```
p=`]
```

This command simply combines the normal paste command (**p**) with a command that indents the previously inserted lines (**=`]**). It actually relies on the fact that when you paste with **p** (lowercase), then the cursor stays on the first character of the pasted text. This is combined with **`]**, which takes you to the last character of the latest inserted text, and gives you a motion across the pasted text from first line to the last.

So all you have to do now is to map this command to a key and then use this key whenever you paste a piece of code into your file.

If you want to use the normal *p* pasting key but with the new functionality, then you can use the following mapping:

```
:nnoremap p p=`]
:nnoremap <c-p> p
```

It maps *Ctrl-p* to what *p* normally did (past with no formatting) and then maps *p* to the new paste with automatic formatting.

[Vim]$^{6+}$ [GVim]$^{6+}$ Using External Formatting Tools

Even though experienced Vim users often say that Vim can do everything, this is of course not the truth, but is close. For those things that Vim can't do, it is smart enough to be able to use external tools.

In the following sections, we will take a look at some of the most used external tools that can be used for formatting your code, and how to use them.

Indent

The Indent program is probably one of the most used external programs for Vim. It has been around since the late 80s for various Unix platforms, and has also later been imported to other platforms including Microsoft Windows.

As the name indicates, this program indents code—especially code that resembles C code in syntax. What you may wonder is why you would use an external program for this, when Vim can handle this task just fine. This is a good question because Vim can do this very well, but the Indent program does it better—and at the same time making it easier to standardize the indentation among multiple editors .

By specializing only in indenting code, indent is able to more effectively indent code than the limited indent functionality included in Vim, for which indenting is a feature and not '*the* feature'. Indent specializes in understanding the code, and indents it according to the code—even if there is a syntactic error in the code.

So how do you use Indent from within Vim? Previously, we have seen several different options for how Vim should indent your code. There is, however, one option that overrules all of them:

```
:set equalprog=PROGRAM
```

What this option does is set the external program Vim should use for indentation when using the commands with **=** . In the case of Indent, you simply change **PROGRAM** to the path to your Indent program. Now, whenever you use one of the indentation commands like **1G=G**, it takes the involved lines and pipes them through the program you have defined in *equalprog*. You can even supply the program with command-line arguments, if needed.

In the case of Indent, there are so many different command line arguments that you will get a better result by configuring its configuration file.

You can always find the latest version of the Indent program at this address: `http://mysite.wanadoo-members.co.uk/indent/beautify.html`.

[Vim]$^{6+}$ [GVim]$^{6+}$ Berkeley Par

In the early 90s, Adam M. Costello began working on a simple command-line program whose only purpose was that when given a text with a paragraph in it, the program would reformat it according to the user's wishes. The program was called Par and, within a year or two, it evolved into a very feature-rich program that can re-format nearly any type of paragraph.

This of course makes Par an ideal external friend for Vim, so let's look at some examples of how it can be used.

If, for instance, you want your text to be nicely formatted in paragraphs with no more than 78 characters on each line, then you could simply use it as:

```
:set formatprg=par\ -w78
```

The `formatprg` option in Vim tells it which program to use for formatting of the text when one of the **gq** commands is used. Notice that the space between the program name and its option is escaped with a backslash. This is needed in order for Vim to see the entire string as one option and not two.

Note that Vim will only use **formatprg** when **formatexpr** is empty. Otherwise the **formatexpr** will be used.

From earlier on, we know that Vim cannot justify the text such that both ends of the lines are aligned with the margins. Fortunately, Par can help us here, and simply by adding a **j** (for 'justify') to our previous **formatprg** value, we can get Par to justify the text:

```
:set formatprg=par\ -w78j
```

Par can not only be used on normal text, but also some parts of the code—the comments.

If you, for instance, have the following comment:

```
/*****************************************************************/
/* This function helps you modify a string and remove all */
/*  unnecessary  characters .  */
/* Don't use this on widechar strings or strings shorter than 10 */
/* characters */
/*****************************************************************/
```

You could select it in Vim and then do:

```
!par 60r
```

(Vim adds `'<,'>` in front of `!` as a range).

This will give you the following result:

```
/*************************************************************/
/* This function helps you modify a string and remove all */
/* unnecessary characters . Don't use this on widechar    */
/* strings or strings shorter than 10 characters          */
/*************************************************************/
```

With a single command, you have transformed an ugly un-formatted comment into a nicely formatted and aligned comment.

The manual page for Par gives a lot of examples on what else it is capable of.

You could easily map different Par commands to different keys in Vim and this way have formatting keys for all text, comments, lists, etc.

You can always find the latest version of Par here:
`http://www.nicemice.net/par/`

Tidy

If you work with web development or XML files, the program Tidy could easily become your next best friend besides Vim. This program cleans up the code that is fed to it and makes it W3C compliant. Being W3C compliant means that the code is constructed such that it follows the HTML guidelines set by the World Wide Web Consortium (W3C).

As a web programmer, I once in a while get in a situation where I have to open someone else's HTML or XML file— just to find that it is one big mess. Because of this, I run all files with the extension .xml, .htm, or .html through Tidy when opening it. This is done using autocommands (au for short) in Vim, which I have added to my `vimrc` file.

For XML this looks like:

```
au FileType xml exe ":silent 1,$!tidy --input-xml true --indent yes -q"
```

or for HTML files:

```
au FileType html,htm exe ":silent 1,$!tidy --indent yes -q"
```

Please note that this will alter the file you open without you knowing anything about what it has changed. In both cases, Vim expects to find a program called Tidy in your path, no matter if you are in Linux or Windows.

As you can see from the arguments I have given for the Tidy program, it can also be used for re-formatting the indentation of the HTML/XML. This option makes the file very readable and it gets a lot easier to get an overview of a file after opening it.

Since Tidy checks for errors in the document, you could assign it to a key, so that at any time you could check if the changes you have made are in fact W3C compliant.

You can always find the latest version of Tidy here:
`http://tidy.sourceforge.net/`

Summary

In this chapter, we have looked at how to get better at formatting both our text and code.

First, we looked at how you can format your text into easily readable paragraphs with the help of a couple of simple Vim commands. We also looked at how to justify the text and why this is not normally so easy to achieve in a plain text editor like Vim. Next, we created functions for marking headlines and generating both bulleted lists and numbered lists. We learned that Vim is very flexible and you can tell it, for instance, whether you want to get it to run your function once for each line you have selected, or whether it should simply let you handle it, and feed your function with all the lines at once.

From here we moved on to looking at how to format your code in Vim—and especially how to indent code. We learned that since we all have our own special coding style, it is often hard to make a generic functionality for code formatting. Vim handles this by giving the user a flexible interface for setting up exactly how he or she wants it to format the code. We also looked at a couple of recipes for how we can format a block of code fast, and even how to format the code that you paste into Vim from other places.

Finally, we took a look at how we can use an external tool to give Vim that extra edge it needs to be the perfect editor. External tools can help you format both text and code, and we took a short look at some of the most popular, to see how they can work together with Vim.

6
Vim Scripting

One of Vim's most powerful features is the extensibility it offers by allowing power users to write scripts. With this feature, you can add nearly any feature to Vim and easily share it with other Vim users.

In this chapter, we will look at some of the aspects of script-writing for Vim. The chapter will contain recipes focusing on the following subjects:

- Creating syntax-coloring scripts for Vim
- How to install and use scripts in Vim
- Different types of scripts
- How to develop scripts in Vim
- Basic syntax of a Vim script
- How to structure a Vim script
- Tips for when you develop Vim scripts
- How to debug a Vim script
- How to use other scripting languages when writing Vim scripts

After you have read this chapter you should have a clear idea about how to use the script functionality in Vim. You should also be able to write your own scripts for Vim and, thereby, be able to add features to Vim.

[Vim]$^{6+}$ [GVim]$^{6+}$ Syntax-Color Schemes

For many programmers, the ability to get the code colored according to the syntax is one of the most important features in Vim. Syntax coloring gives both a better overview of the code and can help the user to discover errors in the code. In Vim, the syntax-coloring system uses script files that very much resemble a Vim script files—they just define colors rather than functionality. In the following section, we will take a look at how to create such a syntax color scheme.

Your First Syntax-Color File

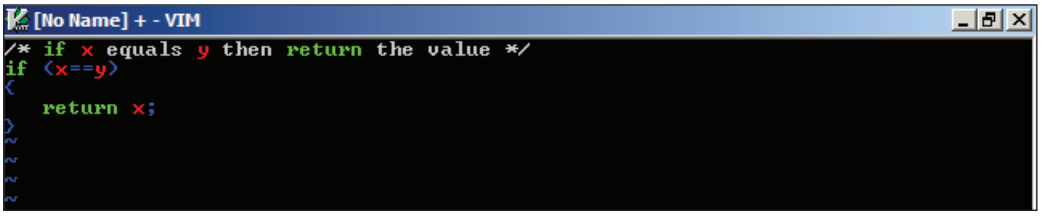
Looking at syntax coloring in a simple way, it's all about identifying certain words and structures in the text, and then giving them the correct color according to what they are. In most cases, however, it is a bit more advanced than that. The syntax-coloring needs to be context-aware in order to be usable. Let's look at an example where we have the following code that we want to syntax-color:

```
/* if x equals y then return the value */
if (x == y)
{
   return x;
}
```

If we simply match on words and symbols, we could get quite a good result. This is done with the following match strings in Vim (as previously described in Chapter 2):

```
:syntax keyword myVars x y
:syntax match mySymbols "[{}();=]"
:syntax keyword myKeywords if return
:highlight myVars ctermfg=red guifg=red
:highlight mySymbols ctermfg=blue guifg=blue
:highlight myKeywords ctermfg=green guifg=green
```

The result is shown in the following screenshot:

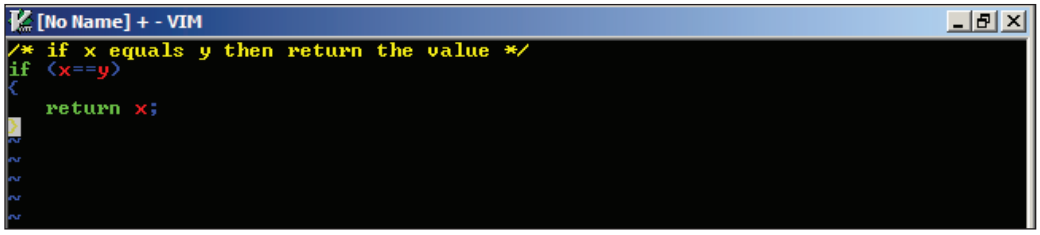


As you can see, the code part is acceptable syntax-colored, but what about the comment part of the code? Because we just match on single words, the same words in the comment are matched too, and hence get the same color as the code. This makes it very hard to distinguish the comments from the code.

So what have we learned from this little example? There is in fact more to syntax coloring than finding words and giving them colors. So let's add a bit of context awareness by saying that it should look for anything between `/*` and `*/` and mark it as a comment, and then go on and syntax-color the rest. The parts of the code that have been colored once will not be colored again by other rules—hence the order of the rules matter. The code for doing this in Vim could look like:

```
:syntax match myComments "/\*.*\*/"
:syntax keyword myVars x y
:syntax match mySymbols "[{}();=]"
:syntax keyword myKeywords if return
:highlight myVars ctermfg=red guifg=red
:highlight mySymbols ctermfg=blue guifg=blue
:highlight myKeywords ctermfg=green guifg=green
:highlight myComments ctermfg=yellow guifg=yellow
```

This results in the code looking like:



So now we actually have a decent syntax coloring for this little piece of code. Of course, this is just a small example and it uses a very limited amount of the syntax-coloring functionality in Vim. Let's move on and take a look at some of the other possibilities you have.

Syntax Regions

In our previous example, we selected the comment-line using the match option for the syntax command. In some cases, however, it is hard to create a suitable match and other simpler approaches are needed.

In Vim, you can select entire regions of the code and color it, simply by setting what the region starts with and what it ends with. To build on our previous example, let's make a region syntax command to substitute our old match command:

```
:syntax region myComments start=/\/\*/ end=/\*\//
```

With this command, I can easily match any of the following comment blocks:

```
/* single line comment */

/************************************
 *  multi line comments
 ************************************/

/* multi line comment
 */
```

But the region option can do more than just setting what a region starts and ends with. It also allows you to set which other things inside the region you want to be colored by other syntax rules. A thing I often use is the ability to give keywords in my function comments like FIXME, OBSOLETE, TODO etc., so I could have code like:

```
/*  function: splitString()
 *  args    : string
 *  OBSOLETE
 */
function splitString(string){
...
```

All we have to do now is make a keyword group that colors the specific keyword like:

```
:syntax keyword myKeywords OBSOLETE FIXME TODO
```

We have to modify the region command, in order to tell that it is allowed to contain other syntax elements. The command will then look like:

```
:syntax region myComments start=/\/\*/ end=/\*\// contains=myKeywords
```

If more than one syntax group needs to be containable inside your region, then you simply add them in the 'contains' list separated by a comma.

You can tell Vim that a region is correct only if both start and end are on the same line, by adding the option *oneline* to your syntax command. Without this option, Vim will start syntax-coloring the code from when it hits the start, until it hits the matching end (or the end of the file).

At some point, you might want a region to be able to be nested inside another region. In that case you will have to tell Vim that is should make this possible. You do so by adding the option **`contained`** to the end of your region command:

```
:syntax region myComments start=/\/\*/ end=/\*\// contains=myKeywords
contained
```

In some cases a block could be anything else in the code, and then of course you don't want to write all the syntax groups. Here, you simply set **`contains`** to be **`ALL`**.

Other keywords like **ALL** exist in Vim:

- **ALLBUT** If this is first item in the list, then all subsequent syntax groups in the list will not be able to be colored in this region.
- **CONTAINED** If this is in the list, then only syntax groups that have the **contained** option are able to be syntax-colored in this region.
- **TOP** If this is in the list, then all the syntax groups except those having the **contained** option are included.

With these you can easily select a large range of syntax groups without having to write them all down. An example could be to select all but the **myComments** group. This is done with the following:

```
:syntax region myCodeblock start=/{/ end=/}/ contains=ALLBUT,myComments
```

If you know that some syntax groups are often used together, then you can join them into a cluster: **:syntax cluster myCluster contains=myKeywords,mySymbols,myConditions**.
A cluster can be used by adding a **@** in front of the name: **:syntax region myComments start=/\/*/ end=/*\// contains=@myCluster**.

Now all you have to do is to combine everything in one file and place in a directory called **syntax** in your **VIMHOME**. The name of the file needs to be the name of the file type you use for your programming language files appended with `.vim`. This means that files with the C programming language have the file extension `.c` and their syntax file is called `c.vim`.

In the above examples, all my syntax group names are prepended with `my`, which is because the syntax for my programming language is called `my` (imaginary of course). Were it the syntax for the C programming language instead, then it would be a good idea to give all the groups names that begin with c (cKeywords, cConditions, cSymbols, etc.).

Just to follow the example, we say that the files in the programming language are named with a `.my` file extension. For simplicity, I want Vim to know my files as having the file type `my`.

If the file type you are using for your files is not known by Vim, then you have to register it for the syntax file to work. This is done by adding a couple of lines to your `filetype.vim` file in your **VIMHOME**. If the file does not exist, then simply create it. In the case of the `My` programming language, I would need to add the following lines:

```
augroup filetypedetect
autocmd BufNewFile,BufRead *.my    setfiletype my
augroup END
```

What this code does is tell Vim that it should add everything between the two `augroup` lines to the auto-command group `filetypedetect`. This is the group of commands that Vim uses to figure out which file type to give a file that it opens or works on.

In our case, it simply adds a line that makes Vim set the file type to `my` whenever it creates or opens a file with the extension `.my`. Many other `autocmd` lines could be added in between the `augroup` lines, if you need it to distinguish between other files types as well.

Now that Vim detects that we are working on files with the file type `my`, it automatically looks for a matching syntax file. It does this by looking in your `VIMHOME/syntax/` directory for a file with a name matching the file type—in this case `my.vim`.

This is all you need to get started on creating your own syntax files that Vim can automatically load whenever you open one of your files.

The best way to learn about how to create a syntax file is by looking at other people's syntax files. Vim comes bundled with syntax files for a wide variety of file types you can use as inspiration or extend with your own commands.

On the other hand, if you just want to add some extra syntax recognition to an already existing syntax file, you have two options. You could of course find the existing syntax file on your system and modify it with your additions. A better solution, however, would be to use the post-processing feature in Vim, which makes it possible to overwrite existing scripting, syntax, etc. that Vim has already loaded. This way, whenever a script is updated on your system, you don't have to add your changes again because they are completely separated from the script itself.

The secret of how to use the post-processor is all about where you place your script files. In your **VIMHOME**, you have a directory called `after` (if it does not exist, simply create it). Whenever Vim looks for a script, syntax file, or color scheme in your runtime path and finds it, it looks for the same file in the `after` directory. So if it found the file `VIMHOME/syntax/c.vim`, then it looks for a file named `VIMHOME/after/syntax/c.vim` to see if there is anything from the first file it should overwrite. The same is the case for scripts found in any of the following directories:

- **plugin**
- **ftplugin**
- **indent**
- **autoload**
- **syntax**
- **colors**

You just add any of these directories to your `after` directory whenever you need a file to be placed in it—if Vim finds it, it will use it.

Color Scheme and Syntax Coloring

In our previous example, we added our own highlighting color groups with the **`:syntax`** command in Vim. This gives you complete control over the colors, but you do, however, also limit yourself to use only those colors—hence it might not follow the colors defined by the color scheme you use in the rest of Vim.

A better approach is to use the color groups already defined in Vim, and thereby split the color definition and syntax highlighting into two parts. This way, whenever you change the color scheme in Vim, your syntax-coloring will change accordingly.

You can find a list of all defined colors by using the command:

```
:highlight
```

or you can take a look inside one of the color scheme files available for Vim. You will find the color schemes in a folder called `color` in your **VIMHOME** or wherever your Vim has been installed.

To speed up the development of a color scheme, you can start out with an existing color scheme and simply modify it to your needs.

Using Scripts

We all have some particular features that we simply can't live without in our editor of choice. Some features are simple modifications like specific key bindings, while others are large and complex extensions throughout the entire editor. Vim will of course not be able to satisfy everyone's needs, so instead it opens up for developers to extend through scripts.

What if you are not a programmer, or do not have the time to develop your own scripts for Vim? Not a problem. Vim is given away free of charge under the charity license; a lot of script developers have decided that they will also give away the scripts they have developed for free. Many of them even put their scripts for easy download on the online Vim community site `http://www.vim.org`, which means that you can easily search for scripts that do exactly what you want them to do.

Script Types

On the Vim Community site, you will find scripts that solve many kinds of tasks, ranging from simple things like inserting the date in the text, to full programming IDEs inside Vim. But actually, there are some defined groups of script types that Vim knows of.

If we look at the script types that add features to Vim, we can split them into two groups. The first group is the **Global** plugins group, which consists of scripts that will be initialized when Vim starts or when the user executes some specific function call. This kind of script is typically used for adding menus to Gvim, adding extra functionality to already existing functions in Vim, or maybe changing some feature in Vim to fit the user's need.

The second group is the **File-Type** plugins group. Scripts in this group are bound to a particular file type (or multiple file types), and the script is only loaded whenever a file of this type is opened or created. The functionality of the scripts in this group could be adding features specific to a particular file type, or the tools used in relation to it. An example could be adding key bindings that make it easy to call a compiler for a certain programming language, or it could be a function that automatically adds a comment above all functions a programmer writes. The scripts in this group also include the scripts for adding syntax coloring, though these are often installed elsewhere than the rest of the scripts in this group.

Installing Scripts

When you download scripts, they typically come in one of the three formats:

- As a single `.vim` script file
- As a compressed (typically Zip) file that typically contains one or more `.vim` files (both global and file-type dependent) and documentation.
- As a vimball, which is an automatic Vim script installation written in Vim.

If the script you want to install is just a Vim script file, then normally all it takes to install it is to copy it to the `VIMHOME/plugin` directory—or `VIMHOME/ftplugin` if it is a file-type-dependent script.

If you are on a multi-user system, you can install it for all users at the same time by installing it in directories with the same names, but instead of in your **VIMHOME**, located where Vim is installed on your system.

If the script comes in a compressed file instead, it can be hard to tell how to install it. Typically, you will simply have to place the file in your **VIMHOME** and then uncompress it there. This normally places the files in the right folders according to how Vim wants them. In any case, there might be a `README` or `INSTALL` file where you can read how to install the script.

If you have found the script on the online Vim Community at `http://www.vim.org`, then you will find instructions for installation on the page that describes the script.

The third and last way to install Vim scripts is by using the vimball installation system—an installation system created for Vim 7 and newer. This system takes a range of files and combines them into a single Vim script archive with the extention `.vba`—a vimball.

Before you start using vimballs, you will need to have the vimball script installed. This adds the functionality for reading and installing vimballs. As with most other scripts for Vim, you will be able to find the script on the online Vim community site.

The latest version of the vimball script can always be found here:
`http://www.vim.org/scripts/script.php?script_id=1502`

When you have the vimball script installed, you are ready to use vimballs for installing other Vim scripts.

Let's say that you have a vimball called `myscript.vba` and want to install it. You simply open the vimball in Vim. Vim will now tell you how to install the script. This is normally done by executing the following command:

```
:source %
```

This installs the script in the first place found in the runtimepath option in Vim. If you want to install the script elsewhere, you can do so by using the following command instead:

```
:UseVimball PATH
```

You need to replace **PATH** with the path to where you want Vim to install the script. Please note that some scripts only work when they are installed in the correct place.

Sometimes, you do not want to install a script unless you know what it contains. If this is the case, then the vimball script has a command that will give you a list of the files that you will get installed if you decide to install the vimball. To get the list of files in the vimball, execute the following command after you have opened the vimball in Vim:

```
:VimballList
```

If the files and directories listed are the ones you expected, then you are ready to install the file, and can then use either the **:source** or the **:UseVimball** commands to install it.

Uninstalling Scripts

There is normally no automatic way of uninstalling scripts after they are installed, and you will have to go through the files, one by one, and uninstall them manually. Having said that, the Vimball script does in fact have an uninstall mechanism.

If you remember the name of the vimball you used to install a Vim script, then you can later use this very same vimball name to uninstall the script. You just have to execute the following command in Vim:

```
:RmVimball VIMBALLNAME
```

Replace the **VIMBALLNAME** with the name of the vimball you used to install the script. If the script was not installed in the default place (if installed with **:UseVimball**), then you can add the installation path as a second argument to the command, and thereby tell the vimball script where to find the files it should remove.

```
:RmVimball VIMBALLNAME PATH
```

In order to be able to bind a vimball name to the files it needs to delete, the script will create a file called `.VimballRecord` in your **VIMHOME**. Note that if you remove this file, you will not be able to uninstall any of the vimballs you have previously installed unless you do it manually.

Script Development

At some point when using Vim, you might find a feature it does not do, and which you need it to do. So now is probably a good time to learn how to make your own scripts for Vim such that you can add this missing functionality.

Before you start, there are, however, a couple of questions you should consider:

First of all, you should make sure that no one else has already created a script that adds what you need—why invent the wheel again? If someone has created a script that does nearly what you need, then why not just help the developer by adding the extra features to that script, and thereby make it work for both him or her and you? This makes development time shorter and limits the number of similar scripts floating around.

If you did in fact, not find any scripts that matched your needs, you need to get working on the script. In this case, you should consider from the start whether or not you want to distribute your script to others when it is done. Bram Moolenaar released Vim, free of charge, for you to use and other Vim script developers have done the same with their scripts. I would urge you to get into the spirit of sharing and do the same with your Vim scripts.

You can find out more about open-source licences on this addrress:
`http://www.opensource.org/`

In case you do decide to share your scripts, you should (from the beginning) remember that Vim is available for a wide variety of platforms and it would be preferable for your script to work on those too. This basically means that:

- You should never expect that some external program is available.
- You should never expect that an external program is installed where you have it installed.
- You should remember that file systems are different on different platforms.

- You should remember that some Vim functionalities are available only on some platforms.
- You should try to make things as configurable as possible—others might not want things how you want them.

With this in mind, you are ready to start looking at how Vim scripts are put together. So let's move on to look at some actual Vim script code.

Scripting Basics

In the next few sections, we will take a look at all the basic types and structures you will need to know in order to be able to write a good Vim script.

If you are already a programmer who knows one or more other programming languages, or scripting languages, then you will most likely find many similarities between these and how the Vim scripting language is constructed.

Types

In Vim there are, roughly speaking, only two types—strings and numbers. When I say roughly speaking, it is because within those two types there are other subtypes. A number can be represented in three different ways depending on how you want it:

- Decimal number : 1, 2, 3, 10, 100, etc.
- Hexadecimal : 0x01, 0x02, 0x03, 0x0A, 0x64
- Octal : 01, 02, 03, 012, 0144

Decimal numbers are used as they are, but with hexadecimal numbers you need to prepend with 0x, and for octal you need to you prepend with 0. Vim will easily be able to use the numbers in calculations together, no matter if they are of the same kind or not. This means that you could easily make calculations saying:

```
:echo 10 + 0x0A + 012
```

which would result in Vim replying with 30.

In Vim, a string is represented as a normal text string encapsulated in either single quotes or double quotes, for example:

```
:echo "this is a string"
:echo 'this is a string'
```

If you need to use the character you used to encapsulate the string inside the string, then you can escape it with a backslash:

```
:echo "this is a string with a \" double quote"
:echo 'the double quote " does not need escaping here'
```

Whether to use single or double quotes depends on the situation.

In a single quoted string, everything is shown as it is represented in the written string—also known as a literal string. This means that you cannot use special escaped characters in the string, for example,

This will work:

```
:echo "string with\n two lines"
```

But this won't:

```
:echo 'string with\n two lines'
```

Besides the newline special escape character `\n`, there are others available in Vim:

- `\n` Newline, line break
- `\r` Carriage return
- `\t` Tab space
- `\123` Octal numbers (123 can be any number)
- `\x123` Hexadecimal number (123 can be any number)
- `\u` Character encoded as up to 4 hex numbers (e.g. \u01fc34)
- `\f` Form feed
- `\e` Esc
- `\b` Backspace
- `\\` A backslash

Besides these escape characters, you can always insert the Vim-specific key acronyms like `<CR>` and `<ESC>` by prepending them with a backslash `\<CR>`. Even the Vim-specific key shortcuts (like `<C-W>` for *Ctrl-W*) can be inserted by escaping them with a backslash.

Variables

In Vim, there are five types of variables, which, even though are defined the same way, can be used very differently. The five types are:

- String A simple string like "this is a string"
- Number A numeric value like 123 or 0x123
- List An ordered sequence of items (an ordered array)
- Dictionary An unordered associative array holding key-value pairs
- Funcref A reference to a function

The name of the variable can include alphanumeric characters and underscore. It cannot, however, start with a number and hence must start with a letter or underscore.

Always use meaningful variable names if possible—remember that others might need to read and understand your code afterwards. To prevent your variables from conflicting with others variables, you can make your variable names unique by, for example, prepending them with your initials like **KSmyvariable**. If there is more than one developer on the script, then you can instead use an abbreviation of the script name—for example, Vim sort script could have variables like **VSmyvariable** or **VSSmyvariable**.

All of the variable types are defined with the `:let` command as follows:

```
:let myvar = VALUE
```

where `VALUE` depends on the type of the variable. In the case of strings and numbers, the value is simply defined as the types in Vim, for example,

```
:let mystringvar = "a string"
:let mynumbervar = 123
```

When working with string and number variables in Vim, there is an automatic conversion going on between the two types depending on how you use them. This means that even though you do:

```
:let mystringvar="123"
```

then you can still do :

```
:let mynumbervar=mystringvar-23
```

and `mystringvar` is automatically converted in place, when the other number is subtracted from it.

You can force a string to become a number by adding 0 to it, e.g.:
`:let mynumber=mystringvar+0`

To force coversion from a number to string, you can use the `string()` function:
`:let mystring=string(mynumber)`

This automatic conversion, however, stops when we move to lists and dictionaries, because these can contain different types within their value. In the following table, you can see examples of how the automatic conversion works:

Input (types)	Result (type)
`"hello" . "world"` (string.string)	`"hello world"` (string)
`"number" . 123`	`"number 123"` (string)
`"123" + 10`	`133` (number)
`"123" - 10 . "hits"`	`"113 hits"` (string)
`"123" - 10 + "hits"`	`113` (number)

To define a list, you use square brackets to enclose a comma-separated list of values:

```
:let mylistvar1 = [1,2,"three",0x04, myfivevar]
```

A list can contain other lists and hence be a list of lists:

```
:let mylistvar2 = [[1,2,3],["four","five","six"]]
```

As you can see, the examples above contain strings, numbers, and lists as item types in the list. This makes this type of variable very suited as a storage container for various values.

Later, we will look more at how to use the values in a list variable, and how to work with multiple lists together.

If you want to create a variable of the type dictionary, it is done with the following let command:

```
:let mydictvar1 = {1: "one", 2: "two", 3: "three"}
```

This creates a dictionary with three items, where the key is the number and the value is the number spelled out with letters (for example, 1 is the key and "one" is the value).

It does not matter whether you write a number key (as above) or a string, Vim will always convert it into a string. So the above example will actually define key-value pairs like *1:one*.

You can also create dictionaries with nested dictionaries. This is done as follows:

```
:let mydictvar2 =  {1: "one",2: "two","tens":{0: "ten",1: "eleven"}}
```

As you can see, the key does not have to follow any strict order, and does not have to be a number (see `"tens"` in the example).

Later, we will look at how to access the values in a dictionary, and how to move from dictionary to list and back.

The final variable type is the funcref type. This type can contain a reference to a function and can, in contrast to the other types, be executed. To define a funcref variable you use the following:

```
:let Myfuncrefvar = function("Myfunction")
```

This ties the function `Myfunction` to the variable `Myfuncrefvar`. Notice that the variable name starts with a capital letter. This is because all user-defined function names in Vim need to have a capital first letter in the name, and hence all variables that can be executed as functions should have the same restriction.

To use a funcref variable later on, you simply use it as a normal variable name, except that you add parentheses after the name like:

```
:echo Myfuncrefvar()
```

Alternatively, it can simply be called with the `:call` command:

```
:call Myfuncrefvar()
```

If the function tied to the variable takes arguments, then these are simply added in the parentheses like `Myfuncrefvar(arg1, arg2,...,argN)`.

When you work with variables in Vim, there are different scopes you can make them available in. This means that you can have some variables available only in a function, while others are global in Vim.

As a Vim script developer, you have to mark the variable yourself to tell Vim in which scope the variable should be available. This is done by adding a scope marker at the beginning of the variable name.

If you define a variable in Vim without specifying which scope it belongs to, then it belongs as a default to the global scope—unless it is defined in a function, which causes it to only be available in the function itself. The following eight scopes are available:

- `v:` Vim predefined global scope
- `g:` Global scope

- **`b:`** Buffer scope—only available in the buffer where it was defined
- **`t:`** Tab scope—only available in the Vim tab where it was defined
- **`w:`** Window scope—only available to the current Vim window (viewport)
- **`l:`** Function scope—local to the function it is defined in
- **`s:`** Sourced file scope—local to a Vim script loaded using **`:source`**
- **`a:`** Argument scope—used in arguments for functions

Did you know that comments in Vim scripts are created by having a quote as the first non-space character on the line: `" this is a comment`.

So for an example that uses some of the scope names, we could look at the following function:

```
let g:sum=0
function SumNumbers(num1,num2)
    let l:sum = a:num1+a:num2
    "check if previous sum was lower than this
    if g:sum < l:sum
       let g:sum=l:sum
    endif
endfunction
" test code
call SumNumbers(3,4)
" this should return 7
echo g:sum
```

Try to use the correct scopes whenever possible. This way you can prevent the global scope from overflowing with variables you do not control and whose origin you do not know.

Conditions

When creating a script for Vim, it is often necessary to be able to check if some condition is met before executing some code. In most programming and scripting languages today, the structure for doing this conditional check is the If-condition check. This is also the case for Vim. In Vim scripting the simplest format for expressing this is:

```
if condition
   code-to-execute-if-condition-is-met
endif
```

If the *condition* evaluates to true, then the code between the `if` and then `endif` lines is executed. If the condition evaluates to false, then the code is not executed.

So what can we use as condition in this if construct? There at two types of conditions you can use here—conditions using logic operators or string operators. So let's take a look at how these operators look. In general, the format is:

```
value1 OPERATOR value2
```

Where `OPERATOR` is the operator that compares `value1` with `value2`. An example could be:

```
value1 >= value2
```

This evaluates to true if `value1` is higher than or equal to `value2`. This is just one of the logical operators available. The following is a full list of the logical operators:

- `val1 == val2` true if `val1` equals `val2`
- `val1 != val2` true if `val1` is not equal to `val2`
- `val1 > val2` true if `val1` is higher than `val2`
- `val1 < val2` true if `val1` is lower than `val2`
- `val1 >= val2` true if `val1` is higher than or equal to `val2`
- `val1 <= val2` true if `val1` is lower lower or equal to `val2`

These operators can be used on both string values and numeric values because Vim can automatically convert back and forth between those types. In the case of strings, the operators work on the letters of the string, one by one, to see if their ASCII value is higher, lower or equal to the one in the other string. For example `"bbb">"aaa"` is true while `"abc">"abd"` is false (because `c` has a lower ASCII value that `d`).

When you work with strings only, there are some more conditions you would want to have available. These are the partial matches which you would want to use if you want to check if a string contains a certain substring or character. In Vim, the operators for this look like:

- `str1 =~ str2` true if `str1` contains the substring `str2` or is equal to `str2`
- `str1 !~ str2` true if `str1` does not contain, and is not equal to, substring `str2`

When using these operators, `str2` is typically a pattern and can use Vim's regular expressions (see **`:help regexp`** for more info). This means that you can not only match simple strings but actually do advanced matches.

All these conditional operators can be used in the if construct and as you will see later, they can also be used elsewhere.

Let's look at some other cases where conditions are useful and can help make your code more structured.

In some cases, you might want to execute one piece of code if the condition evaluates to true, but another piece of code if it evaluates to false. Here, you could of course have two if conditions—one checking if the condition is true, and one that checks if the condition is false. There is, however, another method.

With the if-else-endif construction, you can do just that. The format for expressing this in Vim scripting is:

```
if condition
   code-to-execute-if-condition-is-true
else
   code-to-execute-if-condition-is-NOT-true
endif
```

Another case could be when you have a range of conditions, and depending on which one evaluates to true, the correct piece of code should be executed. This could be done with the following:

```
if condition1
   code-to-execute-if-condition1-is-true
else
   if condition2
       code-to-execute-if-condition2-is-true
    endif
endif
```

As you can see, only one of `condition1` and `condition2` can be evaluated as true, but both can be evaluated as false. This code is, however, cluttered and the extra `endif` can lead to wrongly ended if constructs, if placed wrongly.

A better way to write this is with the if-elseif-else construct, which is formatted as:

```
if condition1
   code-to-execute-if-conition1-is-true
elseif condition2
   code-to-execute-if-condition2-is-true
endif
```

This code does exactly the same as the previous example, except that it is a lot more readable. You can have more than one `elseif`, which means that you can have multiple conditions in the same structure without problems.

Later, when we come to working with loops, we will see how conditions can also be used there.

Lists and Dictionaries

Previously, we have looked at how to create lists and dictionaries. Now let's move on a bit, and look at how to use the data we have stored in them.

When you have a list variable and you want to use one of the values it contains, you simply have to use the name of the variable with square brackets after it, and with the index of the value you want. Index means the place in which the item is placed in the list—starting with first item being placed at index 0. So if we wanted to echo the value "three" from the following list:

```
:let mylistvar1 = [1,2,"three",0x04, myfivevar]
```

it has index 2 and hence could be done like:

```
:echo mylistvar1[2]
```

If we have a list of lists like `mylistvar2`:

```
:let mylistvar2 = [[1,2,3],["four","five","six"]]
```

and want to echo the value `four`, then we need to access the index `0` of the inner list which is placed at index `1` of the outer list. This is done with:

```
:echo mylistvar2[1][0]
```

A point to note about lists in Vim is the possibility to negative indices. Whenever a negative index is used, it will count from the back rather than from the front. So to echo the value `four` from `mylistvar2`, it would look like:

```
:echo mylistvar2[-1][-3]
```

Notice that `-0` does not exist and hence the last item in a list is index `-1`.

If you try to access a non-existing index of a list, Vim will give you an error. You can prevent this error from showing up by using the `get()` function instead. `:echo get(mylistvar1, 2)` where 2 is the index you want to try to access.

If you want to add another item to an already existing list in Vim, you have multiple choices. The simple way to do it, is by using the `add()` function. An example of how this works is:

```
:let mylistvar3 = [1,2,3,4]
:call add(mylistvar3, 5)
:echo mylistvar3
```

This adds the item '5' to the list and then prints the entire list, which now holds 5 items. Another way to do it is to use the list concatenation functionality in Vim. To concatenate two lists in Vim, you just have to use the `+` operator. An example of this is:

```
:let mylistvar4 = [1,2,3,4]
:let mylistvar4 = mylistvar4 + [5,6,7,8]
:echo mylistvar4
```

This creates a list with four items (values 1-4), takes its value list and concatenates it with another list (containing the values 5-8), and then puts the concatenated list back into the **`mylistvar4`** variable. Finally, it echoes the list, which is now eight items long.

Instead of concatenating list one with list two before putting the value back into list one, you can do the entire thing in one move by using the combined equal operator **`+=`**. This takes the right-hand side of the operator and adds it (concatenates) to the left-hand side. For example:
`:let mylistvar4 += [5,6,7,8]`

If the list you concatenate with is only one item, then it basically works like using the add function.

Besides using the + operator for concatenation, you can also use the `extend()` function. An example of how this works could look as follows:

```
:let mylistvar5 = [1,2,3,4]
:call extend(mylistvar5, [5,6,7,8])
:echo mylistvar5
```

Note that there is a very big difference between using `add()` and `extend()` functions for adding elements to the list. If instead of using `extend()` you used `add()` in the above, then you would have added a list in the list resulting in **`mylistvar5`** looking like:

```
mylistvar5 = [1,2,3,4,[5,6,7,8]]
```

which only has five items, the fifth of which is a list containing another four items.

To remove an item from a list, you do it as if you were adding an item to the list—now the function name is, however, `remove()`. For example:

```
:call remove(mylistvar5, 3)
```

This removes the item with index 3 from the list in **`mylistvar5`**.

So let's move on and look at how we can access and modify a dictionary variable. Previously, we created a dictionary variable looking like this:

```
:let mydictvar1 = {1: "one", 2: "two", 3: "three"}
```

Accessing this particular dictionary actually looks very much like accessing a list. If for instance, we want to have the value "two" from the dictionary, we use the [ ] appended to the end again—as with the lists:

```
:echo mydictvar1[2]
```

This looks a lot as if it was just a list we accessed, but notice what happens if we change the keys in the dictionary from being numbers to being some string:

```
:let mydictvar4 = {'banana': 'yellow', 'apple': 'green'}
```

And now we want to access it again to get the color of the apple:

```
:echo mydictvar4['apple']
```

This will print the word '**green**' to the screen. An alternative way to do the same if your key is all alphabetic (ASCII) letters, numbers, or underscores is the following:

```
:echo mydictvar4.apple
```

The first character of the key must always be an ASCII letter.

So compared to the list where everything was ordered and every item had an index, the directory is unordered and the key from the key:value pair is used to get the value instead of the index.

To change one of the values in the directory variable you just use:

```
:let mydictvar4['apple'] = 'red'
```

To add another item to the dictionary, you do exactly as in the above example, except that you use a key that is not already there in the dictionary.

As something special, you can attach a function to the dictionary variable and use it to make distinct things on or with the contents of the dictionary variable. This is better explained with an example, so let's see how that works.

Say we want to be able to take a number and convert each of its digits into the same number written with letters. Let's call the function "convert" and the dictionary variable it is attached to look likes:

```
let mynumbers = {0:'zero',1:'one',2:'two',3:'three',4:'four',
                 5:'five',6:'six',7:'seven',8:'eight',9:'nine'}
```

The function could look like:

```
function mynumbers.convert(numb) dict
   return join(map(split(a:numb,'\zs'),'get(self, v:val,"unknown")'))
endfunction
```

If we look at the function, there are a couple of things you need to notice.

The first thing is the that this function is built like a normal function except that the function name contains the name of your dictionary variable, and we have the keyword "dict" after the function argument.

This keyword is what tells Vim that it should treat this function as a dictionary function and open up for the usage of a special variable—"self". From now on the "self" variable refers to the dictionary to which this function is bound. This means that we could basically do `self[1]` to get the value "one" and so on. The contents of the function itself are a combination of four functions:

- `split` Splits the argument stored in a:numb into an unnamed list
  e.g.:
  ```
  :let a = split("one two")
  :echo a;   " this prints "one"
  ```
- `map` Maps a given command to every element in a list (the one from split)
  e.g.:
  ```
  :let mylist = ["one", "two", "three"]
  :call map(mylist, "<" . v:val . ">")
  :echo mylist[0]   " this prints <one>
  ```
- `get` Gets value from "self" where key is equal to v:val (value from "map")
  e.g.
  ```
  :let mylist2 = ["one","two", "three"]
  :echo get(mylist2, 2,"none") " prints three
  :echo get(mylist2, 3, "none")  " prints "none"
  ```
- `join` Joins all the elements returned by the map-get combination
  e.g:
  ```
  :let mylist3 = ["one", "two", "three"]
  :let mystring = join(mylist3, "+")
  :echo mystring   " prints one+two+three
  ```

(see `:help split()`, `:help join()`, `:help map()`, `:help get()` for more information.)

So translated into a more understandable description, the function `mynumbers.convert` takes a range of digits (a:numb) and splits it into individual digits, then it uses each digit as a key to look up the value in the dictionary variable (`mynumbers`, known as *self*), and joins all the returned values into a string by putting a whitespace between the values, and finally the string is returned.

So now you can use your dictionary variable as a converter from numbers to written number names like:

```
:echo mynumbers.convert(12345)
```

This prints "one two three four five", which is the argument '12345' in words. This is a functionality that opens up a lot of new possibilities.

Loops

When you are working with lists and dictionaries, it is often a needed functionality to be able to go through the some or all of the items in the list/dictionary. For this a programmer would typically use loops—and this is also the case in Vim.

In Vim you have two available looping types:

- for loop
- while loop

In the next sections we will look at how to work with these loops:

For Loops

Let's start by taking a look at the for loop in different situations.

The for loop can be constructed in several different ways, of which this is the simplest:

```
for var in range
    do-something
endfor
```

In this case, we go through all values one by one and for each one of them, the variable *var* is updated to hold the value. An example could be:

```
for myvar in range(1,10)
    echo myvar
endfor
```

This uses the function `range()` to construct a range of numbers from 1 to 10 and then the for loop goes through them beginning with 1. The variable `myvar` will be updated for each iteration, with the value found at the current number from the range. After executing this for loop, it will have printed the numbers 1 through 10.

This does not use a list, but can easily be changed so that it does. So let's look at how that is constructed:

```
for var in list
   do-somthing
endfor
```

Very simple, but let's take an example to see how it works:

```
let mylist = ['a','b','c','d','e','f','g','h','i','j','k']
for itemvar in mylist
   echo itemvar
endfor
```

After executing this, it will have printed the letters a through k one by one. And there really isn't more to it, when using a list.

When you use dictionaries, it takes a bit more work to get the values printed. This is because the value is bound to a key and not to an index. But we have a helper function that just extracts the keys from the dictionary and serves them to the for loop as if they were a list. The functions name is `keys()`. So let's take a look at how we can use this in a for loop:

```
let mydict = {a: "apple", b:"banana", c: "citrus" }
for keyvar in keys(mydict)
   echo mydict[keyvar]
endfor
```

In this case, we get a list of keys from the dictionary `mydict` and go through them, one by one, giving `keyvar` the value of the current key. The value of `keyvar` is then used to look up the matching value in the dictionary `mydict`.

As the items in a dictionary are not ordered, you could, in the above example, end up getting the values printed as "banana citrus apple". There is, however, a function that will help you sort the keys and hence put the values in order. You simply have to use the function `sort()` on the list you get out of the `keys()` function.

```
let mydict = {a: "apple", b:"banana", c: "citrus" }
for keyvar in sort(keys(mydict))
   echo mydict[keyvar]
endfor
```

This, of course, requires that the names of the keys can be ordered individually by using a normal sort algorithm. In the above case, there is no problem because a is before b, which is before c.

The sort function can actually take another argument, which is a function name. This way you can make your own sort algorithm to use when sorting special values. See **`:help sort()`** for more info and an example.

While Loops

The next type of loop we will look at is the while loop. This type of loop, as the name indicates, runs for as long as some condition is true (remember how we previously defined what a condition is in *Conditions*). The basic construction for a while loop is:

```
while condition
   execute-this-code
endwhile
```

where the code between the *while* and the *endwhile* lines is executed as long as the condition evaluates to true. So let's look at an example of how that could work:

```
let x=0
while x <= 5
   echo "x is now " x
   let x+=1
endwhile
```

This example defines a variable `x` with the value of 0 and then goes into a loop that runs as long as `x` is lower than 5. For each iteration in the loop, it prints the value of `x` and then increments the value of `x` by 1. This will result in Vim printing:

```
x is now 0
x is now 1
x is now 2
x is now 3
x is now 4
x is now 5
```

When you use a while loop, there are some extra features available by using some specific statements inside the loop scope.

The first statement is the *break*, which makes the loop end right where it is, and Vim then jumps to the line right after the *endwhile*. An example could be:

```
let x=0
let y = 1000
while x <= 1000
   let y -= 10
   if y <= x
          break
   endif
   let x += 1
endwhile
```

This creates two variables `x` and `y` and gives them the values `0` and `1000`. Then it goes into a while loop where it subtracts `10` from the `y` for each loop iteration. And now the `break` statement comes into the picture, because we now check with an if condition to see if the `y` variable is smaller than or equal to the `x` variable. If this is the case, we don't want to loop anymore and call the `break`. This ends the loop and takes us to just after the loop.

The second statement you can use in the while loop is the `continue` statement.

This statement takes you back to the while line in the while loop without executing the remaining lines under it.

An example of this could be the following:

```
let x=0;
while x <= 5
   let x+=1
    if x == 2
       continue
    endif
    echo "x is now " x
endwhile
```

This example basically loops `x` through the values 1-5. We want it to print out the value of `x` every time except when `x` equals 2. So in order to leave out the loop iteration where x equals 2, we check to see what the value is, and if it is 2, then we use the `continue` statement.

The output from this will be:

```
x is now 1
x is now 3
x is now 4
x is now 5
```

And that's all you need to know about loops to get you started with writing your own scripts.

The *break* and *continue* statements can also be used in the for loop in the same way as shown in the examples with the while loop.

Creating Functions

Throughout this book, we have already used different pieces of code that were constructed as a Vim functions. We have, however, never actually gotten to look into how to actually construct a function and what exactly a function is.

Let's start by looking at the structure of a simple function:

```
function Name(arg1, arg2,...argN) keyword
   code-to-execute-when-function-is-called
endfunction
```

The *Name* is the name you want to call your function. It has to start with a capital letter and can only contain letters, numbers, and underscores.

The *arg1-argN* are the arguments you can require the user of the function to give when calling the function. If you don't need any arguments from the user, then you can simply leave them out (having an empty () after the function name). You can have as many as 20 arguments for a function in Vim, and simply name them as you would like them to be used later in the code inside the function.

After the argument parentheses, it is possible to add a keyword that tells Vim something about what this function is used to do, and how it should call it. We have already seen in this book that the keyword can have the following values:

- `dict` Tell Vim that this function is bound to a dictionary (this chapter)
- `range` Tell Vim that this function is called once for a range of lines and not once for each line it is called on (Chapter 5 when learning to create lists)

In most cases, you do not, however, need this keyword and hence you can just leave it out.

Inside the function, you have all the code that you want executed when the function is called. All variables in this code are local to the function and will not be reachable later when the function has finished executing. If you need to use a variable from outside the function, you can either get it into the function as an argument, or you can use the variable name directly inside the function by adding the global scope marker `g:` to the variable name.

If you get a variable into a function as an argument, then the value is merely copied into the function, and the variable outside the function will not be updated when the value is changed inside the function.

Variables that you get into the function as an argument will have the scope marker `a:` in front of the name when used in the code.

An example of how a simple function could look is the following function that takes two arguments and prints their sum:

```
function PrintSum(num1, num2)
   let sum = a:num1 + a:num2
   echo "the sum is ".sum
endfunction
```

This shows how to use the arguments in the function, but you might also want to be able to update a global variable with the sum. So let's update the example to do this:

```
let sum = 0
function PrintSum(num1, num2)
   let sum = a:num1 + a:num2
   echo "the sum is ".sum
   let g:sum = sum
endfunction
```

Now, you can still use the variable `sum` after the function has ended. So if you did:

```
let sum = 1
call PrintSum(4,5)
echo sum
```

Then this will print the value 9 since the function `PrintSum` has updated the global sum variable.

Updating global variables directly is, however, not considered good programming practice and Vim also gives you another way of updating the global variable—the `return` statement.

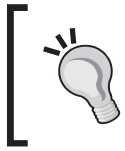
The `return` statement makes the function return a value and end execution of the function, whenever Vim hits the statement. The returned value can then be assigned to the global variable `sum` directly when the function is called. So let's once again update our example to use the `return` statement:

```
function PrintSum(num1, num2)
   let sum = a:num1 + a:num2
   echo "the sum is " sum
   return sum
endfunction
```

Now you can change the function call to be as follows:

```
let sum = PrintSum(4,5)
echo sum
```

If the function had additional lines after the line with the `return` statement, they would never be executed because the function finishes execution after returning. This also means that you can only return one value from a function.



You can have multiple return statements in your function, but the function will return when the first one is reached. Some, however, consider multiple return points to be bad programming practice, unless it is not possible to get around without making the code less understandable.

Variable Argument List

In the previous example, we only had two arguments for the functions, but what if you wished to calculate the sum of more than just those two? In Vim, you can have a variable length argument list by defining your function to have '...' as the last argument in the argument list.

So let's rewrite our sum function once again to take as many arguments as you like (but at least 2 to have something to sum):

```
function PrintSum(num1, num2,...)
   let sum = a:num1 + a:num2
   let argnum = 1
   while argnum <= a:0
       let sum += a:{argnum}
       let argnum+=1
   endwhile
   echo "the sum is " sum
   return sum
endfunction
```

This new function introduces some new variables.

The `argnum` is a counter that we use to go through all the arguments after the `num1` and `num2`.

The number of arguments that the function has been given is stored in the special variable `a:0`, so we use this as a stop for our `argnum` counter when we increase it.

To access each of the variables we now use our `argnum` variable value as index to look up the argument in the list of optional arguments by using the `a:{argnum}` variable. You can see the `a:{}` as a list of the optional arguments (those after `num2`) given to the function, and then `argnum` is used as the index to look up.

For each extra argument there is in the argument list, we add the value to our sum and when done we print the sum and return the value to the global scope.

This means that now we can do:

```
let sum = 0
let sum = PrintSum(4,5,6)
echo sum
let sum = PrintSum(4,5,6,5,4,3,2,1)
echo sum
let sum = PrintSum(1234,5432,3333)
echo sum
```

The result of this will be the following values:

```
15
30
9999
```

If you would rather have all the optional arguments as a list variable in the function, then Vim has another special variable called `a:000`, which acts as a list. With this we can rewrite our function to look like this:

```
function PrintSum(num1, num2,...)
   let sum = a:num1 + a:num2
   for arg in a:000
       let sum += arg
   endfor
   echo "the sum is " sum
   return sum
endfunction
```

This time the values of the optional arguments are passed into the `arg` variable, one by one, in the for loop and then we use the `arg` variable to add to the sum.

When you have created a function but do not want to use it anymore, you can remove it from your Vim session by using the command:

```
:delfunction function-name
```

where `function-name` is the name of the function you want to delete.

To see what a function does, you can use the **`:function`** command to show it to you. In case of the above example you could use:
`:function PrintSum`
If you leave out the function name, you will instead get a list of all available functions.

Besides the functions you create yourself, Vim also comes pumped full of functions for miscellaneous tasks. You can read more about the different functions in the help system under:

```
:help 'function-list'
```

Script Structure

We have been through all the basics of Vim scripting, so now let's take a look at how to put it all together to form a complete script.

Since a Vim script is often just a single file, let's take this as being the goal of our example too. We also want to prepare the script for being made available to others and hence make the code very readable.

The basic outline of a script could be constructed as follows:

Header
The file should begin with a header that states what this script is all about, who the maintainer is (you), the version, when it was last updated, and most importantly a notice about which license you have released the script under.

This could look like:

```
" myscript.vim  : Example script to show how a script is structured.
" Version       : 1.0.5
" Maintainer    : Kim Schulz<kim@schulz.dk>
" Last modified : 01/01/2007
" License       : This script is released under the Vim License.
```

Note how each line is prepended with a `"`. This means that the line is a comment from the quote until the end of the line.

Other information could be placed in the header, like maybe that the script depends on another script, or needs to be used with at least some specific Vim version.

Script-Loaded Check
It is a good practice to check if the script has already been loaded once, and if it has, then unload the function before moving on. This is good because the script is not only installed globally on the system but also in user's own **VIMHOME**.

An example on how this could look is:

```
if exists("loaded_myscript")
    finish "stop loading the script
endif
let loaded_myscript=1
```

If the script has never loaded, the if condition will be false and we will go on and set the `loaded_myscript` variable.

The next time we try to load the script, the if condition will evaluate to true because `loaded_myscript` now exists and the script will then stop loading.

In some cases, it is not the best choice to just stop loading the script, because the user might have changed the version of the script in his or her **VIMHOME**. So instead of just calling *finish*, you could instead unload the functions and then just let the script create the functions again. This could look like:

```
if exists("loaded_myscript")
   delfunction MyglobalfunctionB
   delfunction MyglobalfunctionC
endif
let loaded_myscript=1
```

As you don't know if the user is in a compatible mode (more like vi, less like Vim), it is a good idea to store the user's compatible mode, while in your script. This makes it possible for your script to use Vim-specific functionality without problems. Add the following after the loaded check:

```
:let s:global_cpo = &cpo  " store current compatible-mode
                          " in local variable
:set cpo&vim              " go into nocompatible-mode
```

and the in the end of your script you set it all back as before:

```
:let &cpo = s:global_cpo
```

Configuration
As other users open the script and start looking at it from the top, this is a good place to put all configurable settings. This can be things like path to external program, names of specific files the script needs, file types it should work on, etc.

A user of the script might want to change the settings in his or her `vimrc` file, and you should therefore make sure that you do not overwrite his or her settings. This can be done by checking that the setting does not already exist and only setting it if it doesn't.

An example of settings for this script could be:

```
" variable myscript_path
if !exists("myscript_path")
    let s:vimhomepath = split(&runtimepath,',')
    let s:myscript_path = s:vimhomepath[0]."/plugin/myscript.vim"
else
   let s:myscript_path = myscript_path
   unlet myscript_path
endif

" variable myscript_indent
if !exists("myscript_indent")
   let s:myscript_indent = 4
else
   let s:myscript_indent = myscript_indent
   unlet myscript_indent
endif
```

The example sets two configuration variables—`myscript_path` and `myscript_indent`. We check to see if the variable exists, and if it does not, then we set the default value in the script-scope variable name (e.g. `s:myscript_path`).

If the user has already set this variable, then the value is assigned to the script-scope variable of the same name.

Finally, the user-defined variable is removed with `unlet`, so it does not float around in global scope with no purpose—configuration is only needed in the script and not in global scope.

Mappings
Now it is time to add your key mappings, if you have any. These could be for calling functions, setting variables, and other things. As with configuration variables, mappings is an area where the user might not want the same settings like you—or maybe some other script has already made the same mappings that you want. So let's look at a mapping example with a check to see if a mapping already exists:

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

We have several different pieces put together here when we construct the mapping and mapping check:

- `hasmapto()` Function to check if a mapping to your function exists.
- `<unique>` This tells Vim that it should give an error if a similar map exists.
- `<Leader>` Lets the user decide which map leader to use. `<Leader>` is replaced by the contents of the global variable mapleader.
- `<Plug>` This is a way to make a unique global identifier for a function, such that it will not clash with other functions in global scope.

After putting it all together, we have a check that checks to see if some mapping is already made to the unique function identifier `<Plug>MyscriptMyfunctionA`. If a map does not exists, then `<Leader>a` is mapped to the identifier—unless `<Leader>a` is already used and Vim instead gives an error.

But you may wonder how do we get from `<Plug>MyscriptMyfunctionA` to the actual function `MyfunctionA()` in the script. Well, we have to do some extra mappings to get this done.

```
noremap <unique> <script> <Plug>MyscriptMyfunctionA <SID>MyfunctionA
noremap <SID>MyfunctionA :call <SID>MyfunctionA()<CR>
```

The first mapping maps our unique `<Plug>MyscriptMyfunctionA` identifier to `<SID>MyfunctionA`. We use `<SID>` here, because this little tag is exchanged with Vim's own unique ID for the current script, and this is needed if we want to make a global mapping to a function that is only available in script scope (e.g. `s:MyfunctionA`).

The second mapping binds the actual function (`<SID>MyfunctionA()`, which is `s:MyfunctionA()`) to the global mapping `<SID>MyfunctionA`.

So what actually happens is that when you press **`\a`** (having mapleader set to `\`), then your first mapping translates this into `<Plug>MyscriptMyfunctionA`. This is defined in the script and hence the `<SID>` is now has the right value. Therefore `<Plug>MyscriptMyfunctionA` is again translated further into `<SID>MyfunctionA` which is finally mapped into the actual call of the local function `s:MyfunctionA()`.

You might find this complicated and a bit too much, and you might be right with a relatively unique function name like `MyfunctionA()`. But what if the function was called `Add()`, `Delete()`, `Convert()` or some other function name that many scripts could have implemented. In those cases the function names would clash and Vim would not know which one to use. You could of course just give your functions some weird unique names, but that will in the end just make your script code cluttered and pollute global scope with unnecessary functions.

For more info see:

```
:help <SID>
:help <Plug>
:help 'script-local'
```

Functions

It is now time to add the functions to the script file. We have already seen how a function is added and noticed that it might be better that all functions that are not directly needed in global scope are put into the script scope with the `s:` scope marker. So an example of this could be:

```
" this is our local function with a mapping
function s:MyfunctionA()
   echo "this is the script-scope function MyfunctionA speaking"
endfunction

" this is a global function which can be called by anyone
function MyglobalfunctionB()
   echo "Hello from the global-scope function myglobalfunctionB"
endfunction
" this is another global function which can be called by anyone
function MyglobalfunctionC()
   echo "Hello from MyglobalfuncionC() now calling locally:"
   call <SID>MyfunctionA()
endfunction
```

The first function is a private function, which is only available in the script scope, while the two others are both available in the global scope. Notice, however, how it is possible for one of the global functions to call the local function because it knows the correct `<SID>` for the current script.

Putting it All Together

So let's put it all together and see it as a full script example.

```
" myscript.vim  - Example script to show how a script is structured.
" Version       : 1.0.5
" Maintainer    : Kim Schulz<kim@schulz.dk>
" Last modified : 01/01/2007
```

```
" License       : This script is released under the Vim License.

" check if script is already loaded
if exists("loaded_myscript")
   finish "stop loading the script
endif
let loaded_myscript=1

    let s:global_cpo = &cpo  "store compatible-mode in local variable
    set cpo&vim              " go into nocompatible-mode
" ######## CONFIGURATION ########
" variable myscript_path
if !exists("myscript_path")
   let s:vimhomepath = split(&runtimepath,',')
   let s:myscript_path = s:vimhomepath[0]."/plugin/myscript.vim"
else
   let s:myscript_path = myscript_path
   unlet myscript_path
endif

" variable myscript_indent
if !exists("myscript_indent")
   let s:myscript_indent = 4
else
   let s:myscript_indent = myscript_indent
   unlet myscript_indent
endif

    " ######## FUNCTIONS #########
    " this is our local function with a mapping
function s:MyfunctionA()
   echo "This is the script-scope function MyfunctionA speaking"
endfunction

" this is a global function which can be called by anyone
function MyglobalfunctionB()
   echo "Hello from the global-scope function myglobalfunctionB"
endfunction

" this is another global function which can be called by anyone
function MyglobalfunctionC()
   echo "Hello from MyglobalfuncionC() now calling locally:"
   call <SID>MyfunctionA()
```

```
endfunction

" return to the users own compatible-mode setting
:let &cpo = s:global_cpo
```

And there you have it! Our very first Vim plugin script! It might not have that much functionality, but it shows very well how you should structure your script to make it more understandable. Vim has other types of scripts like file-type plugins, compiler plugins, and library scripts. You can read more about how to modify your script in order to make it these types of scripts in:

```
:help 'write-filetype-plugin'
:help 'write-compiler-plugin'
:help 'write-library-script'
```

On the Vim online community site, you will find thousands of scripts, which you can use as inspiration. Some of these are even library scripts that add functions you can use in your own script to speed up development. See : `http://www.vim.org`.

Scripting Tips

In this section, we will look at a few extra tips that can be handy when you create scripts for Vim. Some are simple code pieces you can add directly in your script, while others are good-to-know tips.

[Vim]$^{6+}$ [GVim]$^{6+}$ Gvim or Vim?

Some scripts have extra features when used in the GUI version of Vim (Gvim). This could be adding menus, toolbars, or other things that only work if you are using Gvim. So what do you do to check if the user runs the script in a console Vim or in Gvim? Vim has already prepared the information for you. You simply have to check if the feature *gui_running* is enabled. To do so, you use a function called `has()`, which returns 1 (true) if a given feature is supported/enabled and 0 (false), if not.

An example could be:

```
if has("gui_running")
   "execute gui-only commands here.
endif
```

This is all you need to do to check if a user has used Gvim or not. Note that it is not enough to check if the feature "gui" exists, because this will return true if your Vim is just compiled with GUI support—even if it is not using it.

Look in **`:help 'feature-list'`** for a complete list of other features you can check with the `has()` function.

[Vim]$^{6+}$ [GVim]$^{6+}$ Which Operating System?

If you have tried to work with multiple operating systems like Microsoft Windows and Linux, you will surely know that there are many differences.

This can be everything from where programs are placed, to which programs you have available and how access to files is restricted.

Sometimes, this can also have an influence on how you construct your Vim script as you might have to call external programs, or use other functionality, specific for a certain operating system.

Vim lets you check which operation system you are on, such that you can stop executing your script or make decisions about how to configure your script. This is done with the following piece of code:

```
if has("win16") || has("win32") || has("win64")|| has("win95")
   " do windows things here
elseif has("unix")
   " do linux/unix things here
endif
```

This example only shows how to check for Windows (all flavors available) and Linux/Unix. As Vim is available on a wide variety of platforms, you can of course also check for these. Find all the operating systems in:

```
:help 'feature-list'
```

[Vim]$^{6+}$ [GVim]$^{6+}$ Which Version of Vim?

Throughout the last decade or two, Vim has developed and been extended with a large list of functions. Sometimes, you want to use the newest functions in your script, as these are the best/easiest to use. But what about the users who have a version of Vim that is older than the one you use, and hence don't have access to the functions you use?

You have three options:

1. Don't care and let it be the user's own problem (not a good option).
2. Check if the user uses a old version of Vim, and then stop executing the script if this is the case.
3. Check if the user has too old a version of Vim, and then use alternative code.

The first option is really not one I would recommend anyone to use, so please don't use that option.

The second option is acceptable, if you can't work around the problem in the old version of Vim. However, if it is possible to make an alternative solution for the older version of Vim, then this will be the most preferable option.

So let's look at how you can check the version of Vim.

Before we look at how to check the version, we have to take a look at how the version number is built.

The number consists of three parts:

- Major number (e.g. 7 for Vim version 7)
- Minor number (e.g. 3 for Vim version 6.3)
- Patch number (e.g. 123 for Vim 7.0.123)

The first two numbers are the actual version number, but when minor features/patches are applied to a version of Vim, it is mostly only the patch number that is increased. It takes quite a bit of change to get the minor number to increase, and a major part of Vim should change in order to increase the major version number.

Therefore, when you want to check which version of Vim the user is using, you do it for two versions—major+minor version and patch number. The code for this could look like:

```
if v:version >= 702 || v:version == 701 && has("patch123")
   " code here is only done for version 7.1 with patch 123
   " and version 7.2 and above
endif
```

The first part of the if condition checks if our version of Vim is version 7.2 (notice that minor version number is padded with 0 if less than 10) or above. If this is not the case, then it checks to see if we have a version 7.1 with patch 123. If patch version 124 or above is included, then you also have patch 123 automatically.

[Vim]6+ [GVim]6+ Printing Longer Lines

Vim was originally created for old text terminals where the length of lines was limited to a certain number of characters. Today, this old limitation shows up once in a while.

One place where you meet this limitation of line length is when printing longer lines to the screen using the "echo" statement. Even though you use Vim in a window where there are more than the traditional 80 characters per line, Vim will still prompt you to press *Enter* after echoing lines longer than 80 characters. There is, however, a way to get around this, and make it possible to use the entire window width to echo on. By window width, the total number of columns in the Vim window minus a single character is meant. So if your Vim window is wide enough to have 120 characters on each line, then the window width is 120-1 characters.

By adding the following function to your script, you will be able to echo screen-wide long lines in Vim:

```
" WideMsg() prints [long] message up to (&columns-1) length
function! WideMsg(msg)
   let x=&ruler | let y=&showcmd
   set noruler noshowcmd
   redraw
   echo a:msg
   let &ruler=x | let &showcmd=y
endfunction
```

This function was originally proposed by the Vim script developer Yakov Lerner on the online Vim community `http://www.vim.org`.

Now whenever you need to echo a long line of text in your script, instead of using the *echo* statement you simply use the function `Widemsg()`. An example could be:

```
:call WideMsg("This should be a very loooong line of text")
```

The length of a single line message is still limited, but now it is limited to the width of the Vim window instead of the traditional 80-1 characters.

[Vim]6+ [GVim]6+ Debugging Vim Scripts

Sometimes things in your scripts do not work exactly as you expect them to. In these cases, it is always good to know how to debug your script.

In this section, we will look at some of the methods you can use to find your error.

Well structured code often has fewer bugs and is also easier to debug.

In Vim, there is a special mode for doing script debugging. Depending on what you want to debug, there are some different ways to start this mode. So let's look at some different cases.

If Vim just throws some errors (by printing them at the bottom of the Vim window), but you are not really sure where it is or why it happens, then you might want to try to start the Vim directly in debugging mode. This is done on the command line by invoking Vim with the `-D` argument.

```
vim -D somefile.txt
```

The debugging mode is started when Vim starts to read the first `vimrc` file it loads (in most cases the global `vimrc` file where Vim is installed). We look at what to do when you get into debug mode in a moment.

Another case where you might want to get into debug mode is when you already know which function the error (most likely) is in, and hence just want to debug that function. In that case you just open Vim as normal (load the script with the particular function if needed) and then use the following command:

```
:debug call Myfunction()
```

where everything after the **`:debug`** is the functionality you want to debug. In this case, it is a simple call of the function **`Myfunction()`**, but it could just as well be any of the following:

```
:debug read somefile.txt
:debug nmap ,a :call Myfunction() <CR>
:debug help :debug
```

So let's look at what to do when we get into the debugging mode.

When reaching the first line that it should debug, Vim breaks the loading and shows something like:

```
Entering Debug mode.  Type "cont" to continue.
cmd: call MyFunction()
>
```

Now you are in the Vim script debugger and have some choices for what to make Vim do.

If you are not familiar with debugging techniques, it might be a good idea to read up on this subject before starting to debug your scripts.

The following commands are available in the debugger (shortcut in parentheses):

- **`cont`** (**`c`**) Continue running the scripts/commands as normal (no debugging) until next breakpoint (more about this later).
- **`quit`** (**`q`**) Quit the debugging process without executing the last lines.
- **`interrupt`** (**`i`**) Stop the current process like quit, but go back to the debugger.
- **`step`** (**`s`**) Execute next line of code and come back to the debugger when it is finished. If line calls a function or sources a file, then it will step into the function/file.
- **`next`** (**`n`**) Execute the next command and come back to the debugger when it is finished. If used on a line with a function call it does not go into the function but steps over it.
- **`finish`** (**`f`**) Continue executing the script without stopping on breakpoints. Go into debug mode when done.

So now you simply execute the different commands to go through the lines of the script/function to see how it jumps through the if conditions, etc. If you want to execute the same command multiple times, you simply press *Enter* without feeding a new command.

You can at any point execute another **`Ex`** command if needed (see **`:help 'ex-command-index'`**), but note that you don't have direct access to the variables etc., in the debugger, unless they are global.

Sometimes, the place you want to get to is many lines into the code, and you really don't want to step all the way through the code until you get to this place.

In that case, you can insert a breakpoint at the exact line where you want to start the real debugging, and then just execute a **`cont`** as first command. A breakpoint is inserted by one of the following commands, depending on how you want it inserted:

```
breakadd func linenum functionname
breakadd file  linenum filename
breakadd here
```

- The first example sets a breakpoint on a particular function. The *functionname* can be a pattern like **`Myfunction*`** if you, for instance, want to break on any function with a name that begins with *Myfunction*.
- Sometimes, however, it is not in a function that the problem resides, but rather around a specific line in a file. If this is the case, then you should use the second command, where you give it a line number and a file name pattern as arguments to tell it where to break.
- The final command is used if you have already stepped to the right place in the file but want to be able to break on it the next time you go through the code in the debugger. This command simply sets a breakpoint on the current line, in the current file, where you currently are in the debugger.
- You can at any point of time get a list of breakpoints with the following command:

  `:breaklist`

- If a breakpoint is no longer needed, you have to delete it. As when adding breakpoints, there are also few different ways to delete them.
- The simplest way to do it is by simply finding the number of the breakpoint in the list of breakpoints, and then using the following command:

  `:breakdel` ***`number`***

- Alternatively, you can delete the breakpoints the same way as you added them—except that you now use **`breakdel`** instead of **`breakadd`**:

  `:breakdel func linenum functionname`

  `:breakdel file linenum file`

  `breakdel here`

- If you want to remove all breakpoints, you can do it in one step by using this command:

  `:breakdel *`

You can add a breakpoint directly on the command line when going into debug mode. Simply use the -c argument like:
`vim -D -c 'breakadd file 15 */.vimrc' somefile.txt`

[Vim]$^{6+}$ [GVim]$^{6+}$ Distributing Vim Scripts

Now that your script is ready, it is time for you to distribute the script (if you have chosen to do so). The online Vim community has become the de facto place to publish scripts for others to find. Because of this, I urge you to do the same. But before you get to this, there are a couple of things you have to get ready.

First of all, you need to figure out whether your script needs to be packed into a compressed file like a ZIP file, or if it should just be distributed as a single `.vim` file. The main reason for choosing the first option is that your script consists of multiple files (like main script, file type plugin, syntax file, documentation, etc.).

How to create ZIP files (or related file types) is beyond what this chapter will look at, but here are a couple of pointers on how I make my ZIP files "install ready":

- Create the ZIP file including the folders where the files are placed relative to your **VIMHOME**. For example, if you have:

  ```
  VIMHOME/plugin/myscript.vim
  VIMHOME/syntax/mylang.vim
  VIMHOME/doc/myscript.txt
  ```

  then, the ZIP file should contain the three folders: plugin, syntax, and doc with one file in each. This makes the installation easy, as you simply have to go into your **VIMHOME** and then unpack the ZIP file.
- Always include a help file for your script. This file should be installed in `VIMHOME/doc/` and contain descriptions of what the script does, which settings it has, and how to use it.

Even though you only have one script file, it can still be a good idea to put it in a ZIP file together with a help file. This makes it easier for you to remember to add documentation. We will look more at how to create Vim documentation in the next section.

[Vim]$^{7+}$ [GVim]$^{7+}$ Making Vimballs

Another, maybe even more interesting, alternative is to make a Vimball. We have previously looked at how to use Vimballs to install scripts, so it could now be interesting to look a bit at how to create one.

The command to create a vimball is constructed as:

```
:[range]MkVimball filename.vba
```

This sure seems simple, right? And it really is. There is, however, a bit of preparation you need to do before calling this function.

The first thing you have to do is to open a new empty buffer in Vim with:
`:enew`

Now you add the paths to all the files (one on each line) relative to your **VIMHOME**. To take the above ZIP file example, it could look like:

```
plugin/myscript.vim
doc/myscript.txt
syntax/mylang.vim
```

When this is done you are actually ready to execute the command across the range of lines. Place the cursor on the first line in the buffer, go into normal mode, and use *Shift-v* to select all the lines with paths on. Now all you have to do is to execute the command:

```
:MkVimball myscript.vba
```

Vim will automatically add the range of the lines you have selected in front of the command. The filename `myscript.vba` can be any name, but if the file already exists, then a warning will be given, but no file is written.

If you really want to overwrite an existing file, then just add an `!` after the **`MkVimball`** to tell Vim that you mean it. There is not more to it. You now have a vimball file called `myscript.vba`, which can be installed as described earlier in this chapter.

Remember that you need to install the vimball script in order to use the vimball functions we have described here. The latest version of the Vimball script can always be found here: `http://www.vim.org/scripts/script.php?script_id=1502`.

Remember the Documentation

Vim has a very comprehensive help system with help for nearly any aspect of using Vim. But what happens when a user installs your script and wants to find help about it? If you haven't added documentation with your script installation, then the user will be out of luck and find no help in the help system. But why not follow the good concept that Vim has started about documenting everything? In other words: "Please include documentation when you release a script for Vim."

So let's take a look at how to create Vim documentation so that it has links, markers, tags, etc. just like the real Vim documentation.

A Vim documentation document is just a plain text file with some special markup. When you create a new document, the first line is the most important of them all. So let's start by looking at that line:

```
*docname.txt* single line of description
```

The first `*` should be the very first character on the very first line of the file.

The `docname.txt` is the name of the file you are currently editing. Vim uses this when linking to the file from the "local additions" list in the Vim help system (see **`:help local-additions`**). The description after the second `*` is a one-line description of what this document is all about. In the case of our example, this line could be:

```
*myscript.txt* Documentation for example script myscript.vim
```

After this line, you place the actual contents of the document.

Typically, this starts with a longer description of what this document is about, and what it will explain. This could include your name and contact info (email address, homepage etc.). After that, a table of contents could be added if the document is long enough to need one (we add it here as an example). It could in the case of `myscript.vim` look like this (including the first line):

```
*myscript.txt* Documentation for example script myscript.vim

Script : myscript.vim - Example script for vim developers
Author : Kim Schulz
Email  : <kim@schulz.dk>
Changed: 01/01/2007
============================================================
                                          *myscript-intro*

1. Overview~
This document gives a short introduction to the example
script myscript.vim.
This script is made as an example for vim users on how to
structure a simple vim plugin script such that it is easy
to read and figure out.
The following is covered in this document:

   1. Overview                 |myscript-intro|
   2. Mappings                 |myscript-mappings|
   3. Functions                |myscript-functions|
   4. Todo                     |myscript-todo|
============================================================
```

In this example, we use most of the formatting tags you have available for Vim documents. Let's go through them one by one to see what they mean.

The first markup is the `*...*` which marks keywords that the Vim help system can jump to. In this case, we have `*myscript-intro*`, which makes it possible to jump directly to this section in the documentation with:

```
:help 'myscript-intro'
```

The next quite simple marker is the `~` after the `Overview` headline. This marks the line with a different color from the rest of the text.

Then we come to the `|...|` around some keywords next to each item in the table of contents. This creates a link to a particular section with a matching keyword (marked with `*...*`). The lines with equal signs are just a good way to mark section borders; they are not actually a markup type.

The following sections would be formatted in the same way, except that if they had a part that shows a piece of Vim code, then another markup would be used. An example of this could be in the function section. So let's take that as an example:

```
============================================================
                                        *myscript-functions*
3. Functions~
Besides the functions available via mappings (as described
in |myscript-mappings|) there are some extra global func-
tions available.

MyglobalfunctionB()~
This function is one of the global functions in this script.
An example of usage could be: >
    :call MyglobalfunctionB()
<
   Vim returns:
    Hello from the global-scope function myglobalfunctionB~

MyglobalfunctionC()~
This function is a global function that also calls one of
the internal functions ("s:MyfunctionA()") in the script.
An example of usage could be: >
    :call MyglobalfunctionC()
<
    Vim returns:
    Hello from MyglobalfuncionC() now calling locally:~
    This is the script-scope function MyfunctionA speaking~
============================================================
```

The special markup in this section is the >...< around the code examples. This is used to mark the code, while we use the ~ colored lines to mark the return from Vim.

That is basically all you need to know in order to create readable documentation for your script.

When the user wants to install the documentation, he or she places it in `VIMHOME/doc/` and then he or she uses the following command:

```
:helptags docdir
```

where *docdir* is the path to **VIMHOME/doc/**. If any of the keywords you have added is already used, then Vim will give you a warning, and you have to change it before distributing the documentation.

Want to distribute your documentation in multiple languages? Take a look in the help system for more information:
:help 'help-translated'

Using External Interpreters

Even though you can do nearly everything with Vim scripts, there are, however, some things that might be smarter or faster to do in other languages. The developers of Vim have recognized this need, and therefore implemented the possibility to interface with other scripting languages from within Vim. There are in particular three languages that you have access to:

- Perl
- Python
- Ruby

In the following sections, will we take a peek at how to interface with these scripting languages and which variables you have access to.

The support for these language interfaces is not included in Vim by default, and you will have to either compile Vim yourself to get it, or find a precompiled version that supports them.

To check if your version supports one of the languages, you simply run Vim on the command line with `-version` argument:

```
vim –version
```

Then, you look through the list of features to see if it has one of the following in the list

```
+perl
+python
+ruby
```

It needs to say `+` in front of the language name to show that it is included. If it instead says e.g. `-perl`, then Perl support is not included.

Alternatively you can just open up Vim and then test for the features with the `has()` function:

```
:echo has("perl")
:echo has("python")
:echo has("ruby")
```

It should return 1 for the languages you have support for.

Vim Scripting in Perl

Perl is a very popular scripting language that has been around for quite some time now. It is very powerful when it comes to parsing text and other similar tasks. This also makes it very useful from within Vim.

The simplest way to use Perl in Vim is with the following command:

```
:perl command
```

This executes the Perl command in the argument *command.* Note that the values you set with the Perl command will persist throughout the entire Vim editing session.

Often you would, however, like to execute more than just a single command and hence you have to use another command.

In that case you can use:

```
:perl << endpattern
    perl code here
endpattern
```

This executes all the Perl code between the endpatterns.

In Perl, Python, or Ruby, you can use anything for your *endpattern*, but the last one needs to be the only word on that particular line, and should be at the beginning of the line. If you leave out the first *endpatter*n, then Vim defaults to use a single dot as *endpattern*.

An example that just prints a single line of text to Vim could be:

```
:perl << EOF
   VIM::Msg("this is a text");
EOF
```

Note how `EOF` is used as *endpattern*, and that in the Perl code I used a function called `VIM::Msg()` to print a message into Vim. This is just one among many functions you can use to interface between Vim and Perl. Other examples of Vim functions you can use from Perl are:

- `VIM::buffers()` Return a list of all buffers open in Vim.
- `VIM::SetOption("option")` Set a Vim option from perl.
- `$curbuf->Name()` Returns the name of the current buffer.
- `$curbuf->Set(100, "fooo")` Replace line 100 in current buffer with new text.
- `$curwin->SetCursor(15,8)` Set cursor at line 15, column 8 in current window.

You can find a full list of the Vim-specific functions you can use from within Perl by looking in the help system with:

```
:help perl-pverview
```

If you put Perl code in your script, you should always remember to check if the user has support for Perl in his or her version of Vim.

It is always a good idea to have your Perl code wrapped in Vim functions in your script. This way it is easy to implement your script, and for an inexperienced user, the script will look normal and work as usual. An example of how to wrap Perl in a function could be:

```
function MoveCursor(row,col)
   if has("perl")
       perl << EOF
       ($oldrow,$oldcol) = $curwin->Cursor();
       VIM::Msg("Old position was: ($oldrow,$oldcol)");
       $curwin->Cursor(row,col);
```

```
EOF
   else
      echo "perl not available. canceling function call"
   endif
endfunction
```

This function gets the old position of the cursor in the current window, prints that position, and then moves the cursor to the position that matches the two arguments for the function (row and column).

If the user does not have Perl support, then a message about this will be written. Note how the EOF is placed entirely to the left, even though the rest of the code is indented. This is strictly needed in order for Vim to be able to recognize it as the *endpattern*.

Vim Scripting in Python

Through the recent years, Python has become the favorite scripting language for many programmers. This is mainly for its ease of use and strict rules about indenting (which lead to more readable code).

As with Perl, there is also an interface for Python in Vim, such that you can break out of Vim and use Python in your script. There are three main ways to use Python.

When you only want to execute a single Python statement from Vim:

```
:python statement
```

An example could be :

```
:python print "hello Vim developer"
```

If you want to execute a larger amount of Python code at the same time, you can use the following from Vim:

```
:python << endpattern
      python statements here
endpattern
```

This executes all the Python code between the *endpatterns*.

The third option for using Python from within Vim is by executing an entire Python script file. This is done with the following command from Vim:

```
:pyfile file.py
```

Replace `file.py` with the name of the script you want to execute.

Sometimes your Python script expects to get some command-line arguments. It is, however, not possible to pass these in the `pyfile` command.

There is, however, a workaround where you set the arguments in `sys.argv` before executing the Python script file. An example could be:

```
:python import sys
:python sys.argv = ["argument1", "argument2"]
:pyfile myscript.py
```

To make it easier to interface between Python and Vim, there is a Python module available called vim. This module gives access to some extra functionality in Vim. Here is an example of usage in a Python script:

```
import vim
window = vim.current.window
window.height =200
window.width = 10
window.cursor = (1,1)
```

You can find a complete list of available functions in the help system in:

```
:help 'python-vim'
```

It is always a good idea to wrap your Python code into Vim functions if you use it in a Vim script.

Vim Scripting in Ruby

For many programmers in the western world, Ruby has been an unknown language until just recently. It has, however, been quite a popular programming language in Asia for some time, and after its introduction as a scripting language for web development, it has become quite popular in the rest of the world. A real strength about Ruby is said to be the fact that it is truly object oriented, which makes the language very modular.

Vim has a nice interface for Ruby, such that you can use it from within Vim. This can be done in several different ways. The simplest way is to execute single Ruby commands in Vim is with the following:

```
:ruby command
```

Replace *command* with any single-line Ruby command. An example could be:

```
:ruby print "Hello from Ruby"
```

You might, however, want to execute several lines of Ruby in a sequence. This can be done with the following Vim command:

```
:ruby << endpattern
      ruby commands here
endpattern
```

This executes all the Ruby code between the *endpatterns*. If you set any variables or create any objects in the code, then these will be stored for later use in Vim.

Here is an example of what this could look like:

```
:ruby << EOF
   window = VIM::Window.current
   window.height = 250
   window.width = 35
   window.cursor = (10,10)
EOF
```

If your Ruby code is in a file, then you can even load in this file directly in Vim and execute it. This is done with the following command:

```
:rubyfile file.rb
```

This is basically an alternative to using:

```
:ruby load 'file.rb'
```

Again all objects etc. you create in the script are stored in Vim for later usage, unless you remove them in the script.

To interface with Vim from your Ruby script, there is a Vim module available for Ruby. The module is called VIM and contains methods for a variety of Vim-related tasks.

Here are some examples of what it can be used for:

- `VIM::Set_option('option')` : Set a vim option.
- `VIM::Message("message")` : Print a message in Vim.
- `$curwin.height` : The height of the current window.
- `$curbuf.width` : The width of the current buffer.
- `VIM::Buffer.current.append(10, "line")` : Append a line to current buffer.
- `VIM::Buffer.current.length` : Return number of lines in current buffer.

You can find a full list of available methods and objects in:

```
:help 'ruby-vim'
```

Always remember to check if the user has Ruby available before using it in your script.

Summary

This chapter has been a chapter especially for those who wanted to learn how to extend Vim with scripts. It started out with one of the simplest types of scripts for Vim—the syntax schemes. We learned how to create a syntax file for programming languages (or other syntactic contents), such that we can get some nice coloring of the contents.

After this, we moved on and looked at how one actually uses scripts in Vim. This was done by looking at the different types of script there are for Vim and where to find them. We ended this session by looking at not only how to install Vim scripts, but also how to uninstall them again afterwards—a functionality it turns out you can only easily use after installing a script in Vim.

So now we knew how to install and use scripts, and it was time to move on to the actual development of scripts for Vim.

We started out by looking at the basic types in Vim and learned that Vim is nearly typeless (only two basic types). There were, however, many types of variable, and these are really useful for storing all the data needed when creating scripts.

The variables could also be used in the next thing we looked at—the conditional structure. We learned about the if condition that only executed some command if a condition could be evaluated to true.

A simple if condition was expanded to be an if-else condition that can execute one of two groups of statements, depending on how a condition evaluates.

Then we extended it even further to the if-elseif-elseif condition construct, which makes it possible to check multiple conditions and only execute the group of statements that was grouped by the condition that evaluated to true.

All of this made us look at what conditions actually are and which ones are available in Vim. We learned that, besides the normal logic operators we normally use in our conditions, there are two special operators for when we work with strings and want to see if a string contains or matches another string or pattern.

Two of the variable types were the list and dictionary types. These can store multiple values at once and even store other lists and dictionaries. We learned that a list is similar to an ordered array in other programming languages and that a dictionary is really an unordered associative array storing key-value pairs. Variables work in different scopes, and we learned how to mark our variables such that Vim will know which scope they belong to.

When you work with lists or dictionaries, it is often nice to be able to loop through the contents and do something with each of the values. This made us look at the two looping structures we have available in Vim—the for loop and the while-loop.

We learned that a for loop is the one most suited for working with lists and dictionaries because it can loop directly on the contents. The while loop on the other hand runs until a certain condition evaluates to true. Did you know that you can break out of a loop before it has finished? If you read this chapter you will know that. You can also fast-forward through the values in a loop and thereby skip the execution of some unneeded code.

So now we knew about most of the basic structures of the Vim scripting language, and it was time to learn how to create functions.

We learned that by the use of a single keyword, Vim can be told how to call the function. This made it possible for us to bind a function to a dictionary variable and thereby have some extra functions that work directly on the contents of the dictionary.

Having learned about how to create functions, it was time to see how we put it all together in a script. We took a top-down approach and went through a script example—line by line.

An important lesson learned was that a script programmer does not know how the user has his or her Vim configured and hence has to make sure that the correct settings exist and that the script does not break any of the users settings.

We learned how to use variables and functions that are only available in the scope of the script, such that we don't pollute the global scope with all our variables and functions. Again this was to help the user such that only the relevant functionality is available to him or her.

After looking at the structure of the script example, we ended the session with a couple of tips about how to make your script check for all sorts of things like operating system and version of Vim.

We had now created a script and it was time to learn how to debug it if something went wrong in the script. This took us through the debug mode in Vim and we learned how to step through the code line by line.

The script was ready and it was time to distribute it to others. We therefore took a look at how to distribute your script to others, such that they can also install it and use it.

Documentation is an important part of a distributed script, so we looked at how the Vim documentation markup language works and how we can use it to create documentation for our script. Having documentation for our script also makes our script show up in the Vim help system under local additions. This makes it easy to jump directly to the documentation by the use of tags and keywords.

For some people, Vim scripting is not enough to satisfy all needs. Because of this, we looked at how to use external scripting languages from within Vim.

We looked at three languages as these are some of the most popular ones available in Vim—Perl, Python, and Ruby. We noted that these external languages are not actually available in Vim by default, but had to be enabled at compile time. There are, however, pre-compiled versions of Vim available that contain these languages.

We went through the three different languages and learned how to execute a single command from the language, a range of commands, and in the case of Python and Ruby even entire scripts.

We also learned that all of the languages have their own modules available such that they can interface with Vim. These modules give access to many of the functions you normally find in Vim and you can thereby control Vim directly from the script.

So that's it! After reading this you should now be able to create your own simple Vim scripts, and with time, more advanced scripts.

A Vim Can Do Everything

It was once said that Vim can do everything. This might not be entirely true, but Vim sure can do a lot of different things that you might not have imagined it can.

In this appendix, we will take a look at some of the things that Vim users have made it do via Vim scripting or by combining it with other programs.

This appendix will take you through everything from games and mail clients, to IRC chatting and complete development IDE setups—all done in Vim.

Vim Games

Even though Vim is a text editor, people have spent a lot of time on creating scripts that make it do other things (other than being an editor). Among these are small games that you can play directly inside the Vim editor. Notice that these are not just simple games like '20 questions' where everything can be done in text. No these are actual graphical games! The graphics are not the best because they are created as so-called ASCII-art. It is, however, enough to give reasonably good game-play, and hours of fun.

So let's take a quick rundown on some of the games that Vim users have created for Vim in Vim script.

[Vim]$^{5+}$ [GVim]$^{5+}$ Game of Life

The first game is not really a game, but still a script worth mentioning. The *Game of Life* is what's normally called a zero-player-game because you don't play the game, but rather just watch how the game plays itself. The game has a simple artificial intelligence that emulates the evolution of cells. The cells follow some very basic rules:

1. Any living cell with fewer than two living neighbors dies, as if by loneliness.
2. Any living cell with more than three living neighbors dies, as if by overcrowding.
3. Any living cell with two or three living neighbors lives, unchanged, to the next generation.
4. Any dead cell with exactly three living neighbors comes to life.

In 1996, a guy who calls himself Eli the Bearded, created a Vim script that implements these rules and prints a *Game of Life* in the current buffer. It was not particularly fast, but was meant as a proof-of-concept implementation. For most people this game will be quite boring, but for *Game of Life* enthusiasts, this implementation could be very interesting.

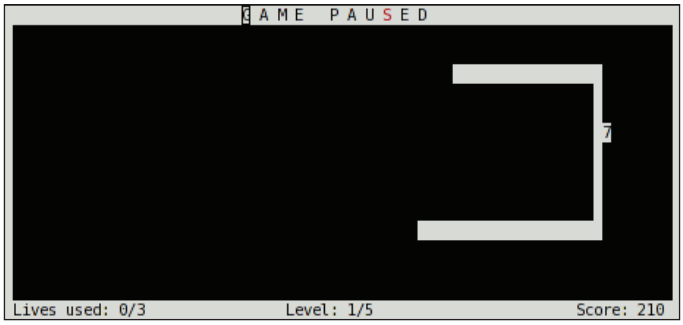
You can find the *Game of Life* at the following site:
`http://www.vanhemert.co.uk/vim/vimacros/life1.vim`

[Vim]$^{5+}$ [GVim]$^{5+}$ Nibbles

When I got my very first PC back in 1986, I only had one game to play. This was a game called *Nibbles*. I sure spent a lot of time playing this game, where I had to control a small worm moving around, in different levels. In each level, it had to eat some things in order to grow. For each thing it ate, another part was added to the end of the worm, and after a while the worm was really long. You could not cross the boundaries or wall of the level and you were not allowed to cross your own tail as well.

In 2004, Hari Krishna Dara recreated this fine game as a Vim script. He only implemented a few levels, but made it possible to easily add more levels, if needed. The game-play was nice and the ran quite smoothly, if you keep in mind that it reprints parts of the text over and over again.

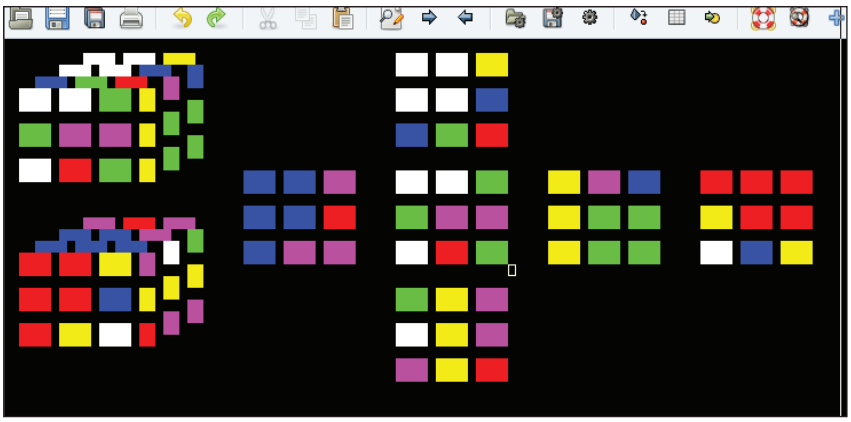
You can find the latest version of the game at the following address:
`http://www.vim.org/scripts/script.php?script_id=916`



[Vim]$^{6+}$ [GVim]$^{6+}$ Rubik's Cube

In 1974, a Hungarian sculptor and professor of architecture, Ernő Rubik created a complex mechanical cube consisting of 27 smaller cubes. The sides had different colors such that each face of the cube had the same color. However, if you turned the layers that the smaller cubes created, you could scramble the colors, and the puzzle of the game was to get it back into an unscrambled setup again.

In 2005, more than 30 years after the of the first *Rubik's cube*, Olivier Vermersch recreated the game of Rubik's cube in Vim script. The Vim version of the cube can still give the player hours of fun and mind puzzle.



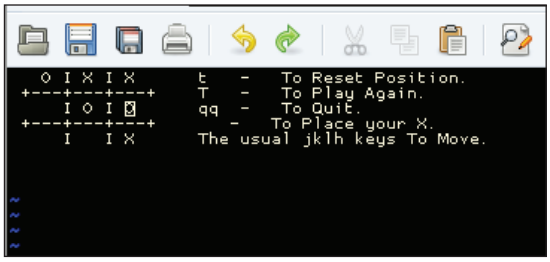
The script that makes it possible to play the game is available via the online Vim community. Instructions on how to download and play this came can be found at this address:

```
http://www.vim.org/scripts/script.php?script_id=1271
```

[Vim]$^{5+}$ [GVim]$^{5+}$ Tic-Tac-Toe

Most of the children of the modern world know the simple game of *Tic-Tac-Toe* and have played it at some point of their life. So, why not implement this game in Vim, such that you can play against your favorite editor to see which of you is the smarter one?

In 1996, Kevin Earls decided to do so, and this ended up as a list of Vim macros that combined to give a nice little *Tic-Tac-Toe* game inside a Vim buffer.



Even though the artificial intelligence for the opponent (Vim) is not that advanced, it can still be quite hard to win against.

You can find the script file you need to use, in order to get this game, at Mr. Earl's home page. Installation and usage instructions are inside the script:
`http://www.vanhemert.co.uk/vim/vimacros/ttt.vim`

[Vim]$^{5+}$ [GVim]$^{5+}$ Mines

Back in 1995, when Microsoft released Windows 95, one of the games that came with it was a little game called *Mines* or *Minesweeper*. The game was a mixture of a simple numeric puzzle, and a gamble game. The player had to get an entire area cleared of mines, by either clicking on areas without mines (guessing) or calculating where the mines were, and marking them with a flag. To calculate the placement of the mines you had to get an area cleared first. This revealed some numbers that said how many mines there were in the squares next to this particular area. When you had calculated where a mine was placed, then you marked it with a flag.

In 2004, Charles E. Campbell recreated this game in Vim script, such that you can get up a minefield in an empty Vim buffer. You can play the game in different difficulty levels. When you can win the easy mode, you can get a headache while trying to get the mines marked on time in the higher levels.

You can find the script that makes this possible on the Vim Online Community by following this link:

`http://vim.sourceforge.net/scripts/script.php?script_id=551`

On this site, you will also find information about how to install and use this game.

[Vim]$^{6+}$ [GVim]$^{6+}$ Sokoban

I have to admit that I just love to play puzzle games, so I obviously also like the game *Sokoban*. The game play is simple and so is the user interface—however, the puzzles are mind bending and quite hard. The task is to be a small man who has to move some boxes around in some mazes/corridors. Sounds easy right? The man can however, only push and not pull the boxes. So, whenever it gets into a corner or blind alley, the box is stuck and you cannot get through that level without restarting.

In 2002, Mike Sharpe recreated this game in Vim using the level definitions from the old Linux game *XSokoban*. He kept the user interface very simple but the game play is still great.

You can find the game XSokoban at the following address:
`http://www.cs.cornell.edu/andru/xsokoban.html`

If you want to play this fine game, then you can find the script and installation instructions on the Vim Online Community site at this address:
`http://www.vim.org/scripts/script.php?script_id=211`

[Vim]$^{6+}$ [GVim]$^{6+}$ Tetris

The final game for Vim that I am going to mention in this appendix is a real classic—*Tetris*, where blocks of different sizes and shapes fall down and need to be placed properly to produce complete rows. This game can be dated back to 1985; the Russian Alexey Pajitnov designed and created it. Since then the game been implemented for almost any platform, and in hundreds of different variations over the same theme.

In 2002, Gergely Kontra decided to implement this game in Vim script and this turned into yet another fine implementation of this classic game. The game even has different modes and keeps a high-score list so that you can play against your own previous records.

You can find the game at the online Vim Community at the following address:
`http://www.vim.org/scripts/script.php?script_id=172`

Here, you will also find instructions about installation and how to play.

[Vim]$^{6+}$ [GVim]$^{6+}$ Programmers IDE

Throughout this book, I have several times talked about features in Vim that could be used when you are a programmer. Personally, I use Vim for nearly all the programming tasks that I do, but I am often met with a skeptical comment from other programmers: "How can you use such a primitive editor?", "How can you live without an IDE?" (Integrated Development Environment) they say? Well, Vim can give you just that if you want it.

So before we look at how, let's look at what an IDE is (or could be).

A typical IDE, if we talk of programs like MS Visual Studio®, basically consists of the following things:

- An editor with automatic indentation, syntax coloring, and auto-completion
- An integrated compiler that makes it possible to jump directly to compile errors in the code
- An integrated debugger that makes it possible to step through the code
- A file explorer such that you can look through the files to add to the project
- A project browser to look through the files included in the project
- A tag browser to look through the tags (defines, functions, methods, classes)
- An easy way to jump between files, definitions, etc.
- Maybe integration with some version/revision control system

So are all these things, at all, possible in Vim? Now, let's go through all the items, one by one, and see how we can get that functionality in Vim.

The first item is obvious, as Vim is an editor it does just that.

The second item in the list is the integration with a compiler. Vim is often used for programming, so it is built with support for compiler integration. For most common programming languages, the settings for compiler integration in Vim are already set. But if they are not, you can learn more about how to set up this feature in:

```
:help compiler
```

This functionality can be linked to a so-called quickfix list, which makes it easy to jump from a compile error to the place in the code where the actual error exists. You can then jump back and forth between the errors, correct them and then compile once again.

To learn more about quickfix lists, look in the help system under:

```
:help quickfix
```

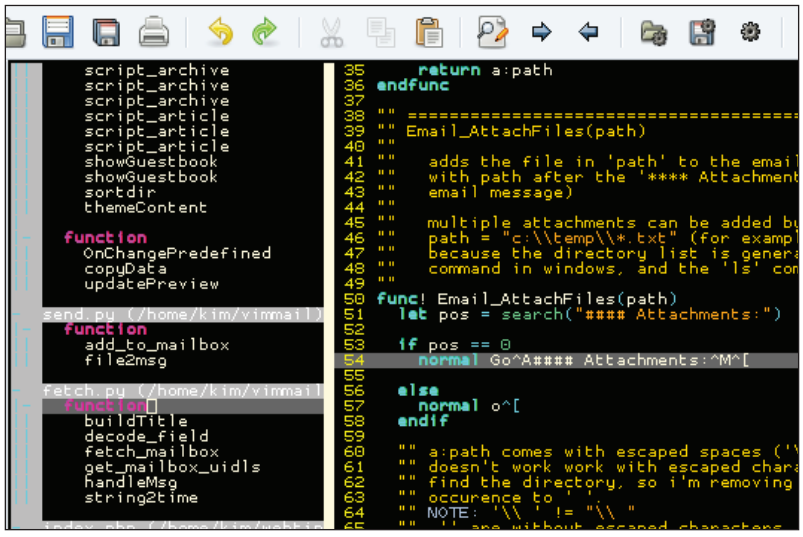
The next item is an integrated debugger. This feature is, unlike the previous ones, not standard in Vim. There are, however, several different scripts that make the integration of debuggers in Vim easy. If you are using Linux, there is a script available called **VimDebug**. This script makes it possible to integrate **gdb** (C/C++ debugger), **jdb** (Java debugger), **pdb** (Python debugger) and the **Perl** debugger. You can find the latest version of this script at `http://www.vim.org/scripts/script.php?script_id=663`.

An alternative and more fully-featured debugger integration is called **Clewn**. This system integrates the gdb debugger with Vim and lets you use all the functionality the debugger would normally have. It even supports remote debugging if you want to debug an external system, without leaving your own system. You can find Clewn with instructions about installation and usage at: `http://clewn.sourceforge.net`.

The next item is the file explorer. Vim comes with its own file explorer that makes it possible to browse directories and files on your system. However, a script has been constructed that makes the file browser even more compatible with an IDE-like setup. The script is called *VtreeExplorer*, and this makes it possible not only to browse the files, but also to see the contents of directories as a tree, which you can fold and unfold. This makes it very fast to navigate. You can find the script and info about how to use it here: `http://vim.sourceforge.net/scripts/script.php?script_id=184`.

Let's move on to the project manager part of the IDE. This component should make it possible to combine files into a project such that when you open a project, all the files in that particular project are also opened. If you are using Gvim, the script *ProjMgr* is a good choice when it comes to managing projects in Vim. It creates a menu item that holds lists of the available projects and the possibility to create new projects. You can find the script here: `http://vim.sourceforge.net/scripts/script.php?script_id=279`.

When it comes to browsing tags, it can be done in two variations. You can do completion of tags, function names, etc. and then browse through the possibilities, or you can have a window that shows all available tags. To create such a window, I will recommend the script called *TagList*. This script supports all programming languages that the Ctags program (which it is also dependent on) supports. It shows a window with a nice list of all defines, functions, methods, classes, etc. and you can easily browse it and jump to the declaration of the tag. You can find the script and info about how to install it at: `http://vim-taglist.sourceforge.net`.



When it comes to moving around in the Vim editor window itself, Vim already has the shortcuts `gf` and `gd`, which takes you to the file or declaration of the tag under the cursor. This makes it very fast to jump around in the files.

Finally, there is the integration with version/revision control systems like **CVS**, **SVN**, and **Perforce**. As with all the other functionality that you need to construct a Vim IDE, this integration is of course also available via scripts. I would recommend the following scripts for the mentioned systems:

CVS and SVN:
`http://vim.sourceforge.net/scripts/script.php?script_id=90`
Perforce: `http://vim.sourceforge.net/scripts/script.php?script_id=240`

So we have all the parts to complete our Vim IDE, and all we have to do now is to put all the parts together and integrate them. If you don't want to go through this task, then some other Vim users have already done the job for you. An example is the **Vim JDE** (Just a Development Environment), which integrates many of the mentioned scripts and combines them into a feature-rich IDE for Java, C, and C++ developers. You can find JDE and how to install it here:

`http://www.vim.org/scripts/script.php?script_id=1213`

Another possibility is to follow one of the Vim tips on the Online Vim Community where you can find tips that tell how to set up the different scripts to integrate them. An example of such a tip is: `http://www.vim.org/tips/tip.php?tip_id=1439`.

[Vim]$^{5+}$ [GVim]$^{5+}$ Mail Program

Vim has for many years been able to be integrated in different programs and among those have been some mail clients—most notable is the open-source mail client **Mutt**.

For some Vim users, however, this was not enough, and they started to implement entire e-mail clients in Vim script—including support for sending, fetching and organizing mail directly from within Vim.

You can find the latest version of the free open-source mail client Mutt here: `http://www.mutt.org/`.

In 2004 Suresh Govindachar created what he called **The Mail Suite** (TMS) for Vim. It was a mail client implemented partly in Vim script and partly in Perl script (embedded in Vim script). TMS has all you need in order to work with emails, and it even makes it possible to use all the features of Vim at the same time.

You can find The Mail Suite and instructions about installation and usage at:
`http://www.vim.org/scripts/script.php?script_id=1052`

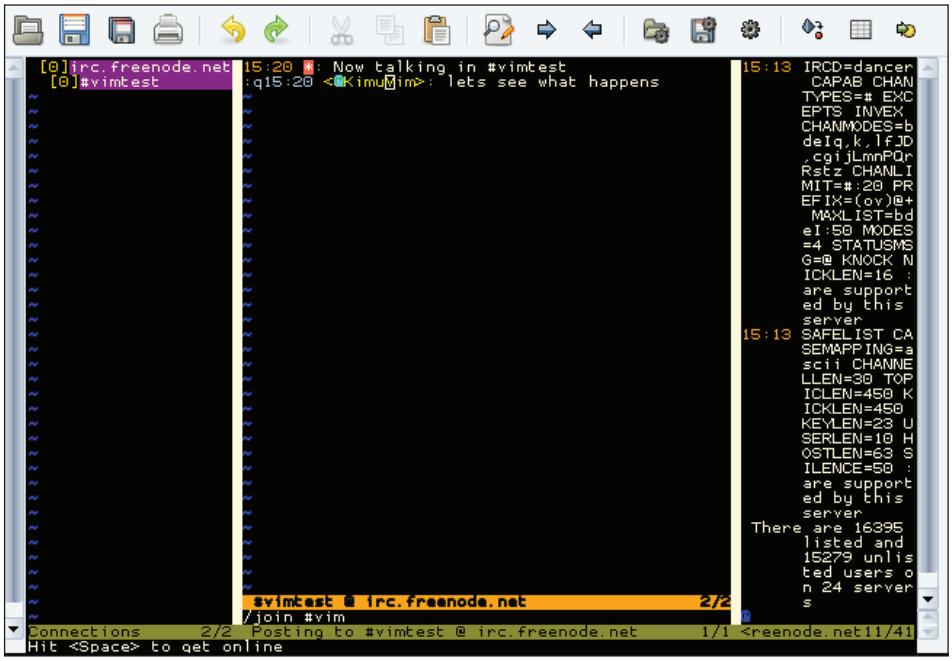
Another person who decided to implement an email client in Vim was David Elentok. In 2005, he started creating an e-mail client for Vim by using a single Vim script and two Python scripts. The script is still in development, but already offers many of the main features you would want from an email client.

You can find Mr. Elentok's scripts at at the following address:
`http://www.ee.bgu.ac.il/~elentok/files/vim/vim-mail/`

Here, you can also find installation instructions and follow the development.

[Vim]$^{6+}$ [GVim]$^{6+}$ Chat with Vim

These days we almost never find a computer without some sort of Chat application installed in it. It has, however, not always been like that. Back in the late 1980s the concept of chatting on the Internet was limited to the Bulletin boards on some BBSs, but in 1988 the Finish guy Jarkko Oikarinen implemented a client-server chat solution he called **Internet Relay Chat** (IRC). During the next decade, IRC became very popular and more and more people joined the chat networks every day. The concept was simple. Get an IRC client, let it connect to an IRC network server, join a chat channel about your favorite subject, and chat with people there.



Many IRC clients exist, and in 2004, the Japanese Vim user Madoka Machitani started implementing an IRC client in Vim. He called the client **VimIRC** and what he constructed was a very feature-rich client that supports chatting in multiple channels on multiple networks. You can even use your normal Vim commands like `I`, `/`, `?` etc.

There is, however, a small problem when chatting from within Vim (besides the fact that it takes time from your actual project). In order for the chat to stay online, it has to create a loop that keeps updating everything. When you move to another buffer that has nothing to do with the IRC client, then this loop is stopped and you are eventually disconnected from the network. So the trick to get this chat client working optimally is to start it in a Gvim or Vim of its own. This can be done directly on the command line with:

```
gvim -i NONE -i .vimircrc  -c VimIRC
```

where `.vimircrc` is the path to a Vim configuration file with VimIRC-specific settings.

You can find the latest version of the VimIRC script at this address:
`http://www.vim.org/scripts/script.php?script_id=931`

Look in the top of the script for a thorough description of how to install and configure the script.

You can chat with other Vim users if you use the IRC network `irc.freenode.net` and join the channel called `#vim`. See you there!

B
Vim Configuration Alternatives

In Chapter 2, we looked at the main configurations files that Vim uses, and throughout the rest of the book, things have been added to the `vimrc` configuration files. In the end, you might have a `vimrc` file full of settings mixed up in one big mess, making it hard to find things in the file.

In this appendix, we take a look at some ideas as to how to keep your `vimrc` file clean and well organized. The recipes range from simple tips to entire configurations systems.

Finally, we will take a look at how it is possible to use the same `vimrc` file on many different computers, simply by storing a copy of it online.

Tips for Keeping your Vimrc Clean

You `vimrc` file is the heart of your personal Vim setup. Without this file, you are bound to have the same setup as the rest of the system. This is also the main reason for keeping this file clean and up to date, such that you always know what it contains, and where in the file it is. It seems like a weird claim that you might not be able to find something in the file. I have, however, found myself in a situation where I had a `vimrc` with more than 2000 lines. Then, I realized that I had to clean up my file in order to get in a state where I could again find things in it. So here I will give some tips on how you can keep your `vimrc` clean and organized.

1. **Always have Vim in nocompatible mode**
   This tip might not make your `vimrc` any cleaner—at least not directly. It is, however, a very important tip. Having Vim in `nocompatible` mode, opens up a lot of features in Vim that other tips and scripts might take advantage of. So the first tip is to always have the following in the beginning of your `vimrc` file: **`set nocompatible`**.

2. **Use Comments**
   At some point, we all have come across a tip about changing something in Vim, and then just added it to our `vimrc`. Later, when we return to our `vimrc` to modify or do a cleanup, we suddenly can't remember what that piece of Vim script did and why we added it. We don't even remember from where we got it. So instead of getting into this problem of not knowing what things do, just add comments to the things you add. What should you write? I would recommend that you write what it does (description), where you got it from (source), and who originally authored it (author). With this information, it is more likely that you can trace back to where you got it from, and in this way you make it possible to decide whether or not to keep the code. Comments are inserted by adding a "(double quote) in front of the comment like: "`This is a comment`

3. **Group data**
   Often scripts need some extra settings or maybe you want to do some extra key bindings (maps) for some functionality in the scripts. To make it clear what belongs together, it is a good idea to group the data. You can group it according to many different things, but I would recommend the following (ordered from the beginning of the file and down):
   - General system-wide setup
   - Key mappings for your own macros
   - Script-specific settings ordered per script
   - 'Playground' with all the script snippets and macros you find and test

4. **Use multiple files**
   Sometimes your `vimrc` file becomes really large, and no matter what you do the file is still cluttered. In that case, it might be better to split it into multiple files. To do so, you simply copy the parts you like (e.g. the mapping group described in previous tip) to a file with a descriptive name, and replace it in your `vimrc` with a line that sources that particular file. In the case of the key mappings, the file could be called `mappings.vim`, and in your `vimrc` you would then have for example: `source $HOME/.vim/mappings.vim`.

5. **Use other files for tests**
   Previously, when I wanted to test some new piece of Vim script, macro, or mapping, I would just add it to the end of my `vimrc` file. Mostly, however, I forgot to remove it, and over time my `vimrc` got filled with my test code. Instead of doing what I did, you should instead have an alternative file to add your code tests, and then simply source it to get the functionality into Vim. If you like the functionality, then you move the code to the right place in your `vimrc` (or other settings file) and if not, then you simply delete the file.

If you follow these tips, you should hopefully be able to keep your `vimrc` file on a clean level, such that you always know where to change the settings for a certain task.

A Vimrc Setup System

When you use Vim, you normally will have to open up a configuration file in order to change the settings permanently. You can change all of them from within Vim, but after closing Vim, all the settings are lost again.

But what if you had a system in Vim such that you could go into a settings menu, and the changes you made there were permanent for the system (until you change them again!)? This is actually possible with a little help from some Vim scripts created by Jos van RisWick.

By using his script, and creating your `vimrc` file using his special syntax, you can make it possible not only to make local changes permanent, but also to change the values in an easy-to-use settings wizard.

This is achieved by using the comments surrounding the setting as a placeholder for the info you are shown in the setup wizard. As I find this system both very smart and very different from what we normally use in Vim, I will here give a short introduction to how you use it.

So let's start by getting the parts we need in order to use this setup system.

The main script and all function scripts it relies on can be found on the folowing address: `http://www.vim.org/scripts/script.php?script_id=1894`.

The package contains the following scripts:

`setup.vim`	The main script.
`array.vim`	Helper functions to work with string arrays.
`arrayg.vim`	Helper functions to work with string arrays (global).
`strfun.vim`	Helper function to manipulate strings.
`tableaf.vim`	Script used to display tab leafs in the setup system.

To install you go into your VIMHOME and unpack the zipfile. This will add all the files to the folder `plugin/`.

Now you are ready to modify your `vimrc` file, but before you do that I urge you to make a copy of your existing `vimrc` file—just in case you don't like the setup system after all.

Before we start adding settings to our `vimrc` (or other settings files), let's take a look at the syntax for using the setup system. As mentioned earlier, the trick is to use the comments surrounding the setting. Such a comment needs to have a very distinct syntax in order for the script to know what to do with it. The basic syntax is as follows:

```
"|ID|tabname|text|command|extra|val1|val2|val3|
```

Let's split it up into parts.

- `"` Begin comment
- `ID` An identifier which the script can use to distinguish the different groups of settings. When the script is started, you tell it which group of settings it should work on.
- `tabname` Name of the tab (called leaf in the script) to add the setting to.
- `text` Text to show in relation to the setting.
- `command` The actual command to execute to set the setting. An example could be `set textwidth=` where the actual value is removed.
- `extra` If the command was for example a map , then the variable part is what keys to map and not the command. In that case it you would also want to be able to set what command to map the keys to. This is what you add in the extra field. If you don't need it, simply leave it out (like `||`).
- `va1-val3` These are the default values you can use as argument for the command. When in the setup system, you can cycle through the different values by using tab the key.

Now we have the basic syntax, so let's take an example of how this could look like:

```
"|SETTINGS|Layout|Set the text width|set tw=||50|70|90|
set tw=60
"|SETTINGS|Layout|Show a ruler?|let &ruler=||0|1|
let &ruler=1
```

This example shows of two typical ways of setting up Vim—setting an option with `set`, and setting an option with `let`. By using `&` before the variable name, you tell Vim that the right-hand side of the equal sign is an expression that it should evaluate before setting it. This also means that if the type of the result of the expression is not the right one for the option, it will be converted automatically. The line below the comment is the actual setting as you want it per default.

If you simply want to add a comment in one of the tabs, you can do so my using a `%` instead of the command like:

```
"|SETTINGS|Layout|Here you will find layout settings|%|
```

You can also use this to add empty lines as spacers by replacing the text with a white space.

Now that you know how to create your settings you need to set up the script itself.

This is done in the script file `setup.vim` and can be found by simply searching for *HERE2* in the `setup.vim` file. The second time you hit the search word you are in the right place (the first one is in the introduction comments). Here, you will find the following settings:

```
" Import settings from these files HERE:
let setup_files=Arr("settings.vim",”mappings.vim”,”scripts.vim”)
let setup_group="SETTINGS"
```

The first setting, `setup_files`, is an array containing a list of the name of the files you have your settings in. In the example above, the files `settings.vim`, `mappings.vim`, and `scripts.vim` are used.

The second setting tells the script which `ID` (as descibed above) to use in this current setup. Here, we use `SETTINGS`, which means that all settings with a comment line above that starts with `|SETTINGS|` will be used.

Now you are almost ready and set to use the setup wizard, but you just need to change one more setting in the `setup.vim` file.

Beacuse we have the settings split into several files, we need to make Vim aware of their existence. Therefore we set the `vimrc` file to source the different settings files. Because of this we also need Vim to reload our `vimrc` file, in order to load the new changes, and for this, we have a special setting in the file.

Search for *HERE1* in the setup.vim file and once again go to the second hit from the top of the file. Here you will find a line where the script sources some file:

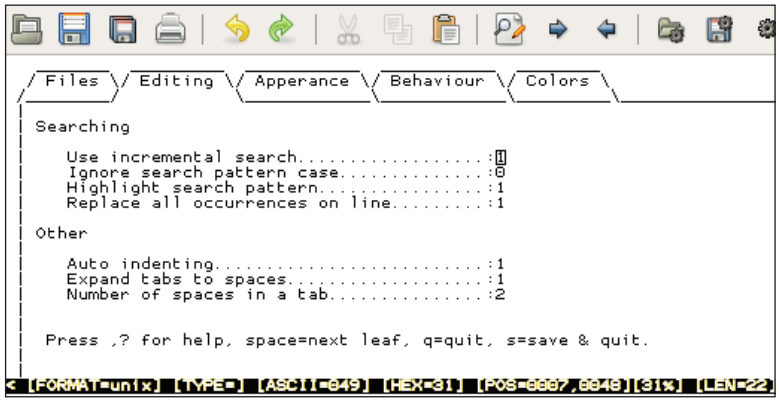
```
autocmd bufleave _setup source ~/.vimrc
```

This is an auto-command which is executed when you leave the setup wizard. You need to set it to source your `vimrc` file (or alternatively all your external setting files), such that Vim will use the new settings.

Now you are all set up and it is time to learn how to start and navigate the Vim setup wizard. To start the setup wizard, you go into normal mode and press `,s`. The cursor is placed on the first setting on the first tab in the settings editor. A number of key maps are now available for you to navigate and use the settings wizard.

- `r` or `R` Enter replace mode to change a setting
- `s` Save changes and quit the editor
- `q` Quit without saving the changes
- `<space>` Cycle through the tab leaves
- `j` Move to next setting on the current tab leaf
- `k` Move to previous setting on the current tag leaf
- `<tab>` Show alternative values for the selected setting
- `<cr>` Accept change on current line
- `<esc>` (normal mode) Redraw current tab leaf
- `<esc>` (insert/replace mode) Discard change

And that is really all you need in order to have an easy-to-use setup wizard built into Vim. So in the following screenshot, let's take a look at how the final result could look like in Vim:



[Vim]$^{7+}$ [GVim]$^{7+}$ Storing Vimrc Online

If you use Vim on many different computers, it is often annoying that you have to set up each Vim as you like it, or have to live with the many different setups.

These days where nearly all computers are connected to the Internet, an obvious solution to accommodate this problem is to store a copy of your `vimrc` online. We already know that Vim can fetch and edit files from online sources like web and FTP servers, but when you have to do it instead for just reading your normal `vimrc` file, you will hit some problems. Normally, when Vim needs to read files on the net, it will use its netrw plugin. However, when Vim is reading the `vimrc` file it has not yet loaded the plugins, and hence cannot read the online version of the `vimrc` file.

This is, however, a problem we can get around with a bit of scripting, so let's take a look at how we can do this:

```
function! GetNetVimrc(vimrc_url)
    source $VIMRUNTIME/plugin/netrwPlugin.vim
    Nread a:vimrc_url
    let tmpfile = tempname()
    save! tmpfile
    source tmpfile
    delete(tmpfile)
    enew
endfunction
```

All of the functionality is combined in the **`GetNetVimrc`** function. The function takes a URL to where your `vimrc` file is stored online. The function starts by sourcing the netrw plugin, such that Vim gets access to the net read and write functions. Next, it uses the `Nread` function to read the `vimrc` file (given by the argument) from the net into the current buffer. The content of the buffer, which is actually the `vimrc` read from the net, is now written to a temporary file, which is then sourced to load the settings. Finally, the temporary file is deleted and a new clean buffer is opened.

So what do you have to do to actually get this to work?

First of all you will have to store your `vimrc` online somewhere where it is accessible. Besides this you will have to add the above function to the `vimrc` on all the computers where you want to use your online `vimrc` version.

To actually activate the function, you could either call it manually as:

```
:call GetNetVimrc("http://www.domain.com/myvimrc")
```

or you can simply add the above line to all the `vimrc` files where you added the function.

Now Vim will act the same way on all computers you use, and if you update your online `vimrc` file, the changes will also be available on all the other computers the next time you use Vim there.

Remember that you can change your online `vimrc` file directly from within Vim. Simply use the `Nread` and `Nwrite` functions to read and write the file.

Index

N

R

S

T

U

V

W