Comprehensive Guide to Mastering Vim

Table of Contents

Introduction to Vim

Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems.

Vim is often called a “programmer’s editor” and is so useful that many consider it an entire IDE. It’s not just for programmers, though. Vim is perfect for all kinds of text editing, from composing email to editing configuration files.

Basic Concepts

Modes

Vim operates in several modes, each serving a different purpose:

- **Normal Mode**: This is the default mode. Keys are used to perform operations such as moving the cursor, deleting text, and copying text.
- **Insert Mode**: This mode is for inserting text. Enter Insert Mode by pressing `i` in Normal Mode.
- **Visual Mode**: This mode is for selecting text. Enter Visual Mode by pressing `v` in Normal Mode.
- **Command-Line Mode**: This mode is for executing commands. Enter Command-Line Mode by pressing `:` in Normal Mode.

Basic Navigation

- `h`, `j`, `k`, `l`: Move left, down, up, right
- `w`, `b`: Move forward/backward by word
- `0`, `$`: Move to the beginning/end of the line
- `gg`, `G`: Move to the beginning/end of the file
- `Ctrl-d`, `Ctrl-u`: Move down/up by half a screen

Editing Text

Inserting Text

- `i`: Insert before the cursor
- `I`: Insert at the beginning of the line
- `a`: Append after the cursor
- `A`: Append at the end of the line
- `o`: Open a new line below the cursor
- `O`: Open a new line above the cursor

Deleting Text

- `x`: Delete character under the cursor
- `dw`: Delete from the cursor to the end of the word
- `dd`: Delete the current line
- `d$`: Delete from the cursor to the end of the line
- `dgg`: Delete from the current line to the beginning of the file
- `dG`: Delete from the current line to the end of the file

Changing Text

- `cw`: Change from the cursor to the end of the word
- `cc`: Change the entire line
- `c$`: Change from the cursor to the end of the line

Copying and Pasting Text

- `yy`: Yank (copy) the current line
- `yw`: Yank from the cursor to the end of the word
- `y$`: Yank from the cursor to the end of the line
- `p`: Paste after the cursor
- `P`: Paste before the cursor

Using Numbers with Commands

You can repeat any command by prefixing it with a number.

- `3dw`: Delete three words
- `5j`: Move down five lines
- `2dd`: Delete two lines
- `10i-<Esc>`: Insert `-` ten times

Advanced Navigation

Search and Replace

- `/pattern`: Search for `pattern`
- `n`, `N`: Move to the next/previous match
- `:%s/old/new/g`: Replace all occurrences of `old` with `new` in the file
- `:s/old/new/g`: Replace all occurrences of `old` with `new` in the current line

Buffers, Windows, and Tabs

- `:e filename`: Open `filename` in a new buffer
- `:bn`, `:bp`: Move to the next/previous buffer
- `:bd`: Delete (close) the current buffer
- `:split`, `:vsplit`: Split the window horizontally/vertically
- `Ctrl-w w`: Switch between windows
- `:tabnew filename`: Open `filename` in a new tab
- `gt`, `gT`: Move to the next/previous tab

Managing Split Windows

Opening Split Windows

Horizontal Split To open a new horizontal split window:

- Use `:split` or `:sp` to split the current window horizontally.

- Example: `:split filename` opens `filename` in a new horizontal split.

Vertical Split To open a new vertical split window:

- Use `:vsplit` or `:vsp` to split the current window vertically.
- Example: `:vsplit filename` opens `filename` in a new vertical split.

Open a File in a New Split To open a specific file in a new split window:

- `:split filename` or `:sp filename` opens `filename` in a horizontal split.
- `:vsplit filename` or `:vsp filename` opens `filename` in a vertical split.

Examples

1. Open a horizontal split:

   ```
   :split
   ```

2. Open a vertical split:

   ```
   :vsplit
   ```

3. Open a specific file in a horizontal split:

   ```
   :split myfile.txt
   ```

4. Open a specific file in a vertical split:

   ```
   :vsplit myfile.txt
   ```

Closing a Split Window

1. **Close the current window**:
   - Use `:q` or `:quit` to close the current split window.
2. **Close all other windows except the current one**:
   - Use `:only` to close all other split windows, leaving only the current window open.
3. **Close a window without saving changes**:
   - Use `:q!` or `:quit!` to forcefully close the current split window without saving changes.
4. **Close a split window using a key combination**:
   - Use `Ctrl-w c` to close the current window.

Closing a Split Window in a Specific Direction

1. **Close the window to the left**:
   - Use `Ctrl-w h` to move to the window to the left, then use `:q` or `Ctrl-w c`.
2. **Close the window to the right**:
   - Use `Ctrl-w l` to move to the window to the right, then use `:q` or `Ctrl-w c`.

3. **Close the window above**:
   - Use `Ctrl-w k` to move to the window above, then use `:q` or `Ctrl-w c`.
4. **Close the window below**:
   - Use `Ctrl-w j` to move to the window below, then use `:q` or `Ctrl-w c`.

Example Configuration for Quick Window Management

You can add custom key mappings to your `.vimrc` to quickly close split windows:

```
" Map <leader>q to close the current split window
nnoremap <leader>q :q<CR>

" Map <leader>o to close all other split windows
nnoremap <leader>o :only<CR>
```

Working with Vimdiff

Vimdiff is a powerful tool for comparing files.

Basic Usage

- `vimdiff file1 file2`: Open `file1` and `file2` side by side for comparison
- `]c`: Jump to the next change
- `[c`: Jump to the previous change
- `do`: Get the change from the other file (diff obtain)
- `dp`: Put the change to the other file (diff put)

Example

To compare two configuration files:

```
vimdiff config1.cfg config2.cfg
```

Use `]c` and `[c` to navigate between changes and `do`/`dp` to apply changes from one file to the other.

Clipboard Operations

Copying to and from the System Clipboard

Vim can interact with the system clipboard, allowing you to copy and paste between Vim and other applications.

- `"+y`: Yank to the system clipboard
- `"+p`: Paste from the system clipboard

Managing the Clipboard in Shared Environments

To avoid overwriting clipboard contents, you can use different registers.

- `"_d`: Delete without affecting the clipboard
- `"+y`: Yank to the system clipboard
- `"+p`: Paste from the system clipboard

Dealing with Clipboard Overwriting Issues

Problem Explanation When using a shared clipboard, if you copy something from the host system, then enter Vim and delete text (e.g., with `:%d` to delete the whole file), the deleted text might overwrite the clipboard contents. This is because Vim uses the default register for deletions, which can interfere with the system clipboard.

Solution To avoid this, you can use the black hole register (`"_"`) for deletions, ensuring that deletions do not affect the clipboard.

Example:

- `:%d` becomes `:%d _`
- `dd` becomes `"_dd`

Example Configuration for Shared Clipboard

Add the following to your `.vimrc` to manage clipboard operations:

```
set clipboard=unnamedplus " Use the system clipboard

" Custom delete command to use the black hole register
nnoremap <leader>d "_d
vnoremap <leader>d "_d
```

Downloading Clipboard Program

To enable clipboard sharing between Vim and the system clipboard, you might need to install `xclip` or `xsel` on Linux:

```
sudo apt-get install xclip
```

or

```
sudo apt-get install xsel
```

Customization

Configuration Files

Vim can be customized using the `.vimrc` file. This file contains settings and commands that are executed when Vim starts.

Example `.vimrc`:

```
" Enable syntax highlighting
syntax on

" Set the number of spaces a <Tab> counts for
set tabstop=4
set shiftwidth=4
set expandtab

" Enable line numbers
set number

" Highlight search results
set hlsearch

" Use the system clipboard
set clipboard=unnamedplus
```

Plugins

Vim's functionality can be extended using plugins. Popular plugin managers include `vim-plug`, `Vundle`, and `Pathogen`.

Example using `vim-plug`:

1. Install `vim-plug`:

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
    https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

2. Add plugins to `.vimrc`:

```
call plug#begin('~/.vim/plugged')

Plug 'preservim/nerdtree'
Plug 'junegunn/fzf', { 'do': { -> fzf#install() } }

call plug#end()
```

3. Install the plugins:

```
:PlugInstall
```

Scripting and Automation

Macros

Macros can record a sequence of commands and replay them.

- `q{register}`: Start recording a macro into `{register}`

- `q`: Stop recording
- `@{register}`: Play back the macro from `{register}`
- `@@`: Repeat the last played macro

Custom Commands

Custom commands can be created using the `command!` directive in the `.vimrc` file.

Example:

```
command! WQ wq
```

Custom Functions

Custom functions allow you to automate more complex tasks in Vim.

Example: Building C/C++ files with specific flags when pressing `Ctrl-p`

1. Add the following to your `.vimrc`:

```
function! BuildFile()
  let l:filetype = &filetype
  if l:filetype == 'c'
    execute 'make CFLAGS=-Wall -Wextra'
  elseif l:filetype == 'cpp'
    execute 'make CXXFLAGS=-Wall -Wextra'
  else
    echo "Unsupported file type: " . l:filetype
  endif
endfunction

nnoremap <C-p> :call BuildFile()<CR>
```

Vimscript

Vimscript is the scripting language used to extend Vim. It can be used to create custom functions, commands, and more.

Example:

```
function! SayHello()
  echo "Hello, Vim!"
endfunction

command! SayHello call SayHello()
```

Advanced Editing Techniques

Registers

Registers store text that has been yanked, deleted, or copied.

- `"{register}y`: Yank into `{register}`
- `"{register}p`: Paste from `{register}`

Common registers: - `""`: Default register - `"+`: System clipboard

Marks and Jumps

Marks allow you to set a position in the text and jump back to it later.

- `m{a-z}`: Set mark `{a-z}`
- `'{a-z}`: Jump to the beginning of the line with mark `{a-z}`
- `` `{a-z}` ``: Jump to the exact position of mark `{a-z}`

Text Objects

Text objects are a way to operate on regions of text defined by syntactic elements.

- `ciw`: Change inside word
- `di"`: Delete inside quotes
- `vi(`: Visual select inside parentheses

Integration with External Tools

Compilers and Build Systems

Vim can be integrated with compilers and build systems using the `:make` command.

Example:

1. Add a build command to `.vimrc`:

```
set makeprg=make
```

2. Run the build command:

```
:make
```

3. Navigate through errors:

```
:cnext
:cprev
```

Version Control Systems

Vim can be integrated with version control systems like Git using plugins such as `fugitive.vim`.

Example using `fugitive.vim`:

1. Install `fugitive.vim`:

```
Plug 'tpope/vim-fugitive'
```

2. Use Git commands within Vim:

```
:Gstatus
:Gcommit
:Gpush
```

Performance Tuning

Optimizing Vim

1. Disable swap files for faster performance:

```
set noswapfile
```

2. Reduce the time Vim waits after an ESC key press:

```
set timeoutlen=300
```

3. Disable the mouse if not needed:

```
set mouse=
```

Resources

1. **Vim Documentation**: Official Vim Documentation
2. **Vim Adventures**: Interactive Learning Game
3. **Vim Awesome**: Vim Plugins
4. **Practical Vim**: Book by Drew Neil
5. **Vimcasts**: Screencasts for Vim Users

Conclusion

Vim is an incredibly powerful and flexible text editor. By mastering its features, you can significantly increase your productivity and efficiency. This guide provides a comprehensive overview of Vim's capabilities, from basic commands to advanced scripting and customization. With practice and exploration, you can harness the full potential of Vim to suit your specific editing needs. Happy Vimming!