

The Vim Tutorial and Reference

By Steve Oualline

Table of Contents

Introduction.....	26
Chapter 1:Basic Editing.....	28
Chapter 2:Editing a Little Faster.....	42
Chapter 3:Searching.....	59
Chapter 4:Text Blocks and Multiple Files.....	69
Chapter 5:Windows and Tabs.....	83
Chapter 6:Basic Visual Mode.....	99
Chapter 7:Commands for Programmers.....	111
Chapter 8:Basic Abbreviations, Keyboard Mapping, and Initialization Files.....	148
Chapter 9:Basic Command-Mode Commands.....	159
Chapter 10:Basic GUI Usage.....	168
Chapter 11:Dealing with Text Files.....	175
Chapter 12:Automatic Completion.....	192
Chapter 13:Autocommands.....	202
Chapter 14:File Recovery and Command-Line Arguments.....	211
Chapter 15:Miscellaneous Commands.....	226
Chapter 16:Cookbook.....	232
Chapter 17:Topics Not Covered.....	247
Chapter 18:Complete Basic Editing.....	260
Chapter 19:Advanced Searching Using Regular Expressions.....	290
Chapter 20:Advanced Text Blocks and Multiple Files.....	309
Chapter 21:All About Windows, Tabs, and Sessions.....	335
Chapter 22:Advanced Visual Mode.....	351
Chapter 23:Advanced Commands for Programmers.....	363
Chapter 24:All About Abbreviations and Keyboard Mapping.....	418
Chapter 25:Complete Command-Mode (:) Commands.....	427
Chapter 26:Advanced GUI Commands.....	453
Chapter 27:Expressions and Functions.....	487
Chapter 28:Customizing the Editor.....	533
Chapter 29:Language-Dependent Syntax Options.....	563
Chapter 30:How to Write a Syntax File.....	586
Appendix A: Installing Vim.....	598
Appendix B: The <> Key Names.....	605
Appendix C: Normal-Mode Commands.....	608
Appendix D: Command-Mode Commands.....	633
Appendix E: Visual-Mode Commands.....	734
Appendix F: Insert Mode Commands.....	738
Appendix G: Option List.....	742

The Vim Tutorial and Reference

Appendix H: Vim License Agreement.....	770
Appendix I: Basic Vim Quick Reference.....	772
Appendix J: Vim Quick Reference.....	773

Detail Table of Contents

Introduction.....	26
Copyright and License Information.....	26
Part I Tutorial.....	27
Chapter 1:Basic Editing.....	28
Before You Start.....	28
Running Vim for the First Time.....	29
The vim Command.....	29
Modes.....	30
Editing for the First Time.....	30
Inserting Text.....	30
Getting Out of Trouble.....	31
Moving Around.....	31
Aliases.....	32
Deleting Characters.....	33
Undo and Redo.....	34
Getting Out.....	35
Discarding Changes.....	35
Other editing Commands.....	36
Inserting Characters at the End of a Line.....	36
Deleting a Line.....	36
Opening Up New Lines.....	37
Help.....	37
Help Language.....	40
Other Ways to Get Help.....	40
Using a Count to Edit Faster.....	41
The Vim Tutorial.....	41
Summary.....	41
Chapter 2:Editing a Little Faster.....	42
Word Movement.....	42
Moving to the Start or End of a Line.....	43
Searching Along a Single Line.....	44
Moving to a Specific Line.....	45
Telling Where You Are in a File.....	46
Where Am I?.....	47

The Vim Tutorial and Reference

Scrolling Up and Down.....	48
Deleting Text.....	49
Deleting Text Without Visual Mode.....	49
Where to Put the Count (3dw or d3w).....	50
Visual vs. Normal Mode Delete.....	51
Changing Text.....	51
The . Command.....	53
Joining Lines.....	53
Replacing Characters.....	54
Changing Case.....	55
Keyboard Macros.....	55
Digraphs.....	57
Chapter 3:Searching.....	59
Simple Searches.....	59
Search History.....	60
History Window.....	61
Searching Options.....	61
Highlighting.....	61
Incremental Searches.....	62
Searching Backward.....	64
Reverse Search History.....	64
Changing Direction.....	65
Basic Regular Expressions.....	66
The Beginning (^) and End (\$) of a Line.....	66
Match Any Single Character (.).....	67
Matching Special Characters.....	68
Regular Expression Summary.....	68
Chapter 4:Text Blocks and Multiple Files.....	69
Cut, Paste, and Copy.....	69
Character Twiddling.....	70
More on "Putting".....	71
Moving Large Blocks of Text.....	71
Marks.....	72
Where Are the Marks?.....	73
Yanking.....	74
Normal Mode Yanking.....	74
Yanking Lines.....	76
Filtering.....	76
Normal Mode Filtering.....	76
Editing Another File.....	77
The :view Command.....	77
Dealing with Multiple Files.....	78

The Vim Tutorial and Reference

Which File Am I On?.....	79
Going Back a File.....	80
Editing the First or Last File.....	80
Editing Two Files.....	81
Matching.....	81
 Chapter 5:Windows and Tabs.....	83
Opening a New Window.....	83
Vertical Windows.....	84
Opening Another Window with Another File.....	86
Quick Split.....	86
Controlling Window Size.....	87
Split Summary.....	87
The :new Command.....	88
Split and View.....	88
Changing Window Size.....	88
Buffers.....	90
Selecting a Buffer.....	92
Buffer Types.....	94
Buffer Options.....	94
Basic Tabbed Editing.....	96
Selecting a tab.....	97
Finding Files with Tabs.....	98
Editing Multiple Files From the Command Line.....	98
 Chapter 6:Basic Visual Mode.....	99
Entering Visual Mode.....	99
The Three Visual Modes.....	100
Leaving Visual Mode.....	101
Editing with Visual Mode.....	102
Deleting Text in Visual Mode.....	102
Yanking Text.....	103
Switching Modes.....	103
Changing Text.....	103
Joining Lines.....	103
Commands for Programmers.....	104
Keyword Lookup.....	104
Visual Block Mode.....	105
Inserting Text.....	105
Changing Text.....	106
Replacing.....	108
Shifting.....	109
Visual Block Help.....	110

The Vim Tutorial and Reference

Chapter 7:Commands for Programmers.....	111
Syntax Coloring.....	111
Syntax Coloring Problems.....	112
Matching Pairs.....	113
Shift Commands.....	115
Automatic Indentation.....	116
C Indentation.....	116
Smartindent.....	118
Autoindent.....	118
The = Command.....	119
Diff Mode.....	120
Folding.....	122
Locating Items in a Program.....	125
Instant Word Searches Including #include Files ([CTRL-I,]CTRL-I).....	126
Jumping to a Variable Definition (gd, gD).....	126
Jump to Macro Definition ([CTRL-D,]CTRL-D).....	127
Displaying Macro Definitions ([d,]d, [D,]D).....	127
Shifting a Block of Text Enclosed in {}.....	129
Indenting a Block Using Visual Mode.....	130
Finding the man Pages.....	130
Tags.....	131
Help and Tags.....	134
Windows and Tags.....	134
Finding a Procedure When You Only Know Part of the Name.....	135
Shorthand Commands.....	138
The Care and Feeding of Makefiles.....	138
Sorting a List of Files.....	139
Sorting a List in Visual Mode.....	140
Making the Program.....	141
The :make command.....	141
The 'errorfile' Option.....	145
Searching for a Given String.....	145
Vim and outside edits.....	146
Other Interesting Commands.....	147
Chapter 8:Basic Abbreviations, Keyboard Mapping, and Initialization Files..	148
Abbreviations.....	148
Listing Your Abbreviations.....	149
Mapping.....	149
Listing Your Mappings.....	150
Fixing the Way Delete Works.....	151
Controlling What the Backspace Key Does.....	151
Saving Your Setting.....	152
My .vimrc File.....	155

The Vim Tutorial and Reference

Script Files.....	157
Chapter 9:Basic Command-Mode Commands.....	159
Entering Command-Line Mode.....	159
The Print Command.....	160
Ranges.....	160
Marks.....	161
Visual-Mode Range Specification.....	162
Substitute Command.....	162
How to Change Last, First to First, Last.....	164
Reading and Writing Files.....	165
Saving the file under a new name.....	165
The :shell Command.....	165
Printing the file.....	166
Chapter 10:Basic GUI Usage.....	168
Starting Vim in GUI Mode.....	168
Mouse Usage.....	169
X Mouse Behavior.....	170
Microsoft Windows Mouse Behavior.....	170
Special Mouse Usage.....	170
Tear-Off Menus.....	171
Toolbar.....	172
Showing the cursor.....	173
Chapter 11:Dealing with Text Files.....	175
Automatic Text Wrapping.....	175
Text Formatting Commands.....	177
Justifying Text.....	179
Fine-Tuning the Formatting.....	179
The joinspaces Option.....	179
The formatoptions Option.....	180
Formatting and numbered list.....	183
Using an External Formatting Program.....	183
Basic Spelling.....	184
Finding Spelling Errors.....	185
Spelling Language.....	185
Word Lists.....	186
Multiple word lists.....	187
File Formats.....	187
Changing How the Last Line Ends.....	188
Troff-Related Movement.....	189
Section Moving.....	190
Defining Sections.....	191

The Vim Tutorial and Reference

Encrypting with rot13.....	191
Chapter 12:Automatic Completion.....	192
Automatic Completion.....	192
How Vim Searches for Words.....	193
Searching Forward.....	193
Automatic Completion Details.....	193
Automatic Completion Details.....	194
The Include Path.....	194
Specifying a Dictionary.....	194
Controlling What Is Searched For.....	195
Tag Search.....	197
Finding Filenames.....	199
Line Mode.....	200
Dictionary and Thesaurus.....	200
Guessing.....	200
User and Omni completion.....	201
Adjusting the Screen.....	201
Chapter 13:Autocommands.....	202
Basic Autocommands.....	202
Groups.....	203
Events.....	204
File Patterns.....	207
Nesting.....	208
Listing Autocommands.....	208
Removing Commands.....	209
Ignoring Events.....	209
Chapter 14:File Recovery and Command-Line Arguments.....	211
Command-Line Arguments.....	211
Encryption.....	212
Switching Between Encrypted and Unencrypted Modes.....	213
Limits on Encryption.....	213
Executing Vim in a Script or Batch File.....	214
Additional Command-Line Arguments.....	214
Foreign Languages.....	216
Backup Files.....	217
Skipping the backup.....	218
Controlling How the File Is Written.....	218
Basic File Recovery.....	219
Recovering from the Command Line.....	221
Advanced Swap File Management.....	222
Controlling When the Swap File Is Written.....	222

The Vim Tutorial and Reference

Controlling Where the Swap File Is Written.....	223
Advanced File Writing.....	223
Saving Your Work.....	224
The :recover Command.....	224
MS-DOS Filenames.....	225
readonly and modified Options.....	225
Chapter 15:Miscellaneous Commands.....	226
Printing the Character.....	226
Going to a Specific Character in the File.....	226
Screen Redraw.....	227
Sleep.....	227
Terminal Control.....	227
Suspending the Editor.....	227
General Help.....	227
Other Help Commands.....	228
Nvi Compatibility Commands.....	228
Window Size.....	228
Executing commands without changing things.....	228
Signs.....	228
Viewing the Introduction Screen.....	230
Open Mode.....	231
Chapter 16:Cookbook.....	232
Character Twiddling.....	232
Replacing One Word with Another Using One Command.....	232
Interactively Replacing One Word with Another.....	233
Alternate Method.....	234
Moving Text.....	234
Copying a Block of Text from One File to Another.....	235
Method 1: Two Windows with Traditional Vi-Style Commands.....	235
Method 2: Two Windows Using Visual Mode.....	236
Method 3: Two Different Vim Programs.....	236
Sorting a Section.....	237
Sorting the old Vi way:.....	239
Finding a Procedure in a C Program.....	239
Drawing Comment Boxes.....	240
Reading a UNIX man Page.....	241
Trimming the Blanks off an End-of-Line.....	242
Oops, I Left the File Write-Protected.....	243
Changing Last, First to First Last.....	243
How to Edit All the Files that Contain a Given Word.....	245
Finding All Occurrences of a Word Using the Built-in Search Commands.	246

The Vim Tutorial and Reference

Chapter 17:Topics Not Covered.....	247
Interfaces to Other Applications.....	247
Cscope.....	247
MzScheme.....	248
Netbeans.....	249
OLE.....	249
Perl.....	249
Python.....	250
Python Interface Command Reference.....	250
Ruby.....	250
Sniff+.....	250
Sun Visual WorkShop.....	251
Tcl.....	251
Foreign Languages.....	252
Input Method.....	254
Arabic.....	254
Chinese.....	255
Farsi.....	255
Hebrew.....	256
Japanese.....	257
Korean.....	257
Binary Files.....	257
Modeless (Almost) Editing.....	258
Operating System File Modes.....	258
Part II Reference.....	259

Chapter 18:Complete Basic Editing.....	260
Word Movement.....	260
Move to the End of a Word.....	260
Defining What a Word Is.....	261
Special Characters for the iskeyword Option.....	262
Other Types of Words.....	263
There Are "words," and Then There Are "WORDS".....	263
Beginning of a Line.....	264
Repeating Single-Character Searches.....	264
Moving Lines Up and Down.....	265
Cursor-Movement Commands.....	266
Jumping Around.....	266
Using the Change List.....	267
Controlling Some Commands.....	268
Where Am I, in Detail.....	268
Scrolling Up.....	269

The Vim Tutorial and Reference

Scrolling Up Summary.....	270
Scrolling Down.....	271
Define How Much to Scroll.....	272
Adjusting the View.....	272
Delete to the End of the Line.....	275
The C Command.....	275
The s Command.....	275
The S Command.....	276
Deleting Text.....	276
Insert Text at the Beginning or End of the Line.....	277
Arithmetic.....	277
Joining Lines with Spaces.....	278
Replace Mode.....	279
Virtual Editing.....	280
Replace Mode.....	281
Digraphs.....	282
Changing Case.....	282
Other Case-Changing Commands.....	283
Advanced Undo.....	284
Undo Time Machine.....	284
Undo Level.....	284
Change Sets and Branching.....	285
Getting Out.....	288
Chapter 19:Advanced Searching Using Regular Expressions.....	290
Searching Options.....	290
Case Sensitivity.....	290
Wrapping.....	293
Turning Off Search Wrapping.....	293
Interrupting Searches.....	294
Instant Word Searches.....	294
Search Offsets.....	295
Specifying Offsets.....	297
Complete Regular Expressions.....	298
Beginning (\<) and End (\>) of a Word.....	298
Modifiers and Grouping.....	299
Special Atoms.....	300
Character Ranges.....	301
Character Classes.....	302
Repeat Modifiers.....	302
Repeating as Little as Possible.....	302
Grouping (\(\)).....	303
The Or Operator (\).....	303
Putting It All Together.....	304

The Vim Tutorial and Reference

The 'magic' Option.....	304
Offset Specification Reference.....	305
Regular Expressions Reference.....	305
Simple Atoms.....	305
Range Atoms.....	306
Character Classes.....	306
Patterns (Used for Substitutions).....	306
Special Character Atoms.....	306
Modifiers.....	307
Chapter 20:Advanced Text Blocks and Multiple Files.....	309
Additional Put Commands.....	309
Special Marks.....	310
Manipulating Marks.....	311
Multiple Registers.....	311
Appending Text.....	313
Special Registers.....	313
The Black Hole Register (_).	314
The Expression Register (=).	314
The Clipboard Register (*).	315
How to Edit All the Files That Contain a Given Word.....	315
Editing a Specific File.....	316
Changing the File List.....	316
The +cmd Argument.....	317
Defining the file list (arguments).	317
Local and Global argument Lists.....	319
Global Marks.....	319
Advanced Text Entry.....	321
Movement.....	321
Inserting Text.....	321
Inserting a Register.....	323
Leaving Insert Mode.....	324
The .viminfo File.....	324
Dealing with Long Lines.....	327
Wrapping.....	331
Spelling Dictionaries.....	332
Dumping dictionaries.....	333
Customizing the spelling system.....	333
Chapter 21:All About Windows, Tabs, and Sessions.....	335
Moving Between Windows.....	335
Moving Windows Up and Down.....	336
Performing Operations on All Windows.....	338
Other Window Commands.....	339

The Vim Tutorial and Reference

Editing the Alternate File.....	340
Split Search.....	341
Shorthand Commands.....	341
Other Window Commands.....	342
Advanced Buffers.....	342
Adding a Buffer.....	342
Deleting a Buffer.....	343
Unloading a Buffer.....	343
Opening a Window for Each Buffer.....	343
Windowing Options.....	344
Controloing a split.....	345
Tabs.....	346
Executing a command for all tabs.....	346
Other tab commands.....	346
Customzing tabs.....	347
Tabs without the GUI.....	348
Sessions.....	348
Specifying What Is Saved in a Session.....	349
Views.....	350
 Chapter 22:Advanced Visual Mode.....	351
Visual Mode and Registers.....	351
The \$ Command.....	351
Repeating a Visual Selection.....	352
Selecting Objects.....	353
Moving to the Other End of a Selection.....	355
Case Changes.....	357
Joining Lines.....	357
Formatting a Block.....	358
The Encode (g?) Command.....	358
The Colon (:) Commands.....	359
Pipe (!) Command.....	359
Select Mode.....	360
Deleting the Selection.....	360
Replacing Text.....	361
Switching Modes.....	362
Avoiding Automatic Reselection.....	362
 Chapter 23:Advanced Commands for Programmers.....	363
Removing an Automatic Indentation.....	364
Inserting Indent.....	365
Inserting Registers.....	365
To Tab or Not to Tab.....	367
Spaces and Tabs.....	368

The Vim Tutorial and Reference

Smart Tabs.....	368
Using a Different Tab Stop.....	370
No Tabs.....	370
The 'copyindent' and 'preserveindent' Options.....	371
The :retab Command.....	371
Modelines.....	372
Shift Details.....	372
Specifying a Formatting Program.....	373
Formatting Comments.....	373
Defining a Comment.....	374
Customizing the C Indentation.....	376
The 'cinoptions' Options.....	377
The 'cinwords' Option.....	383
Advanced Diff Mode.....	383
Moving from difference to difference.....	384
Moving Differences Around.....	384
Customizing Diff.....	385
Comparing Two Files The Old Fashioned Way.....	385
Advanced Folding.....	387
Additional fold commands.....	389
Toggling folds.....	390
Enabling and disabling folding.....	390
Moving around folds.....	390
Executing a command for all folds.....	390
Customizing folds.....	390
Controlling what opens and closes folds.....	391
Fold Methods.....	391
The Preview Window.....	392
Match Options.....	394
Showing Matches.....	394
Finding Unmatched Characters.....	395
Method Location.....	396
Movement.....	396
Comment Moves.....	396
Dealing with Multiple Directories.....	397
The include Path.....	399
Checking the Path.....	400
Defining a Definition.....	402
Locating include Files.....	402
Multiple Error Lists.....	403
Manipulating the quick fix list.....	403
Local error lists.....	404
Customizing the :make Command.....	406
The Error Format.....	407

The Vim Tutorial and Reference

The 'switchbuf' Option.....	410
Customizing :grep.....	411
Defining How a Tag Search Is Done.....	411
Customizing the Syntax Highlighting.....	412
Black-and-White Terminals.....	412
Color Terminals.....	413
GUI Definition.....	414
Combining Definitions.....	415
Syntax Elements.....	415
Color Chart.....	416
The 'syntax' Option.....	417
 Chapter 24:All About Abbreviations and Keyboard Mapping.....	418
Removing an Abbreviation.....	418
Abbreviations for Certain Modes.....	419
Listing Abbreviations.....	419
Forcing Abbreviation Completion.....	420
Mapping and Modes.....	420
Other :map Commands.....	422
Undoing a Mapping.....	422
Clearing Out a Map.....	423
Listing the Mappings.....	423
Recursive Mapping.....	424
Remapping Abbreviations.....	424
Language Dependent Mappings.....	425
The usual suite of commands applies. :map Mode Table.....	425
 Chapter 25:Complete Command-Mode (:) Commands.....	427
Advanced Command Entry.....	427
Editing Commands.....	427
Other Ways to Specify Ranges.....	429
Deleting with a Count.....	430
Copy and Move.....	431
Inserting Text.....	432
Printing with Line Numbers.....	433
Printing with list Enabled.....	433
Print the Text and Then Some.....	434
Substitute.....	434
Substitute flags.....	437
Making g the Default.....	437
Global Changes.....	438
Commands for Programs.....	438
Include File Searches.....	438
Jumping to Macro Definitions.....	439

The Vim Tutorial and Reference

Split the Window and Go to a Macro Definition.....	440
Listing the Macros.....	440
Listing the First Definition.....	440
Override Option (!).....	440
Directory Manipulation.....	441
Current File.....	442
Advanced :write Commands.....	443
Updating Files.....	444
Reading Files.....	444
Register Execution.....	444
Simple Edits.....	445
Shifting.....	445
Changing Text.....	445
Entering Insert Mode.....	445
Joining Lines.....	445
Yanking Text.....	445
Putting Text.....	445
Undo/Redo.....	446
Marks.....	446
Miscellaneous Commands.....	446
The :preserve Command.....	446
The Shell Commands.....	446
Shell Configuration.....	447
Command History.....	447
Setting the Number of Remembered Commands.....	449
Viewing Previous Error Messages.....	449
Redirecting the Output.....	449
Executing a :normal Command.....	450
Getting Out.....	450
Write and Quit.....	450
Advanced Hardcopy.....	451
 Chapter 26:Advanced GUI Commands.....	453
Switching to the GUI Mode.....	453
Window Size and Position.....	453
Microsoft Windows Size and Position Command-Line Specification.....	455
Moving the Window.....	455
Window Size.....	455
The :winsize Command.....	455
The 'guioptions'.....	456
Changing the Toolbar.....	461
Customizing the Icon.....	462
Mouse Customization.....	464
Mouse Focus.....	464

The Vim Tutorial and Reference

The 'mousemodel' Option.....	464
Mouse Configuration.....	465
Mouse Mapping.....	465
Double-Click Time.....	466
Hiding the Mouse Cursor.....	466
Select Mode.....	466
Custom Menus.....	467
Special Menu Names.....	469
Limiting the Maximum Number of Generated Items.....	469
Toolbar Icons.....	469
Toolbar Tips.....	471
Listing Menu Mappings.....	471
Executing a Menu Item.....	473
No Remapping Menus.....	473
Removing Menu Items.....	473
Tearing Off a Menu.....	474
Translating A Menu.....	474
Special GUI Commands.....	475
The File Browsers.....	475
Finding a String.....	477
Replace Dialog Box.....	477
Finding Help.....	478
Confirmation.....	478
Browsing the Options.....	479
Using the Clipboard.....	481
Coloring.....	482
Selecting the Font.....	482
Customizing Select Mode.....	483
Mouse Usage in Insert Mode.....	483
Microsoft Windows - Specific Commands.....	483
Changing the Appearance of the Cursor.....	484
Line spacing.....	485
X Windows System - Specific Commands.....	486
Selecting the Connection with :shell Commands.....	486
MS-DOS-Specific Commands.....	486
Chapter 27:Expressions and Functions.....	487
Basic Variables and Expressions.....	487
Special Variable Names.....	488
Constants.....	489
Expressions.....	490
Deleting a Variable.....	491
Locking and unlocking a variable.....	491
Locking Arrays and Dictionaries.....	492

The Vim Tutorial and Reference

Entering Commands.....	492
How to Experiment.....	493
The :echo Statement.....	493
Echoing in Color.....	494
Printing error messages using :echoerr.....	494
Echoing message.....	495
Control Statements.....	495
The :if Statement.....	495
Looping.....	496
The :for Loop.....	496
The :execute Command.....	497
Exceptions.....	497
Defining Your Own Function.....	499
Using a Function.....	500
Function Options.....	500
Listing Functions.....	501
Deleting a Function.....	502
Running Functions in a Sandbox.....	502
Debugging a Function.....	502
Other debugging commands.....	505
Redrawing the screen.....	506
Profiling a function.....	506
Other Profile Commands.....	508
Deleting Profile Items.....	508
User-Defined Commands.....	508
The Operator Function.....	510
Built-In Functions.....	511
Obsolete Functions.....	531
Plugins and other scripts.....	531
Chapter 28:Customizing the Editor.....	533
Setting.....	533
Boolean Options.....	533
Numeric Options.....	534
String-Related Commands.....	535
Another Set Command.....	536
Other :set Arguments.....	536
Chaining Commands.....	537
Automatically Setting Options in a File.....	537
Local .vimrc Files.....	538
Customizing Keyboard Usage.....	539
Microsoft Windows.....	539
Customizing Keyboard Mappings.....	540
Confirmation.....	541

The Vim Tutorial and Reference

Customizing Messages.....	541
Showing the Mode.....	542
Showing Partial Commands.....	542
Short Messages.....	543
The 'terse' Option.....	544
The "File Modified" Warning.....	544
Error Bells.....	544
Status Line Format.....	545
Rulers.....	548
Reporting Changes.....	548
Help Window Height.....	549
Preview Window Height.....	549
Defining How 'list' Mode Works.....	549
Changing the line number size.....	550
Changing the Highlighting.....	550
The 'more' Option.....	552
Number Format.....	552
Restoring the Screen.....	552
Pasting Text.....	552
Wildcards.....	553
Customizing Behavior of the Screen Movement Commands.....	557
File Writing Options.....	557
Memory Options.....	557
Function Execution Options.....	557
Terminal Options.....	558
Terminal Name.....	558
Lazy Redraw.....	558
Internal Termcap.....	558
Fast Terminals.....	558
Mouse Usage Inside a Terminal.....	558
How Much to Scroll.....	559
Some More Obscure Options.....	559
Compatibility.....	559
Weirdinvert.....	560
Debugging.....	560
Production.....	561
Keyboard Mapping.....	561
Encoding.....	562
Macintosh Silliness.....	562
Obsolete Options.....	562
Legacy Options.....	562
Chapter 29:Language-Dependent Syntax Options.....	563
Abel.....	563

The Vim Tutorial and Reference

Ada.....	563
Ant.....	564
Apache.....	565
Assembly Language.....	565
ASP.....	566
BaaN.....	566
Basic.....	566
C and C++.....	566
Doxygen.....	569
CH.....	569
Chill.....	569
Changelog.....	570
COBOL.....	570
Cold Fusion.....	570
CSH / TCSH.....	570
CYNLIB.....	570
CWEB.....	570
Desktop.....	570
Dircolors.....	571
DocBook.....	571
DosBatch.....	571
Doxygen.....	571
DTD.....	571
Eiffel.....	572
ERLANG.....	572
FlexWiki.....	573
Form.....	573
Fortran.....	573
FVWM.....	573
Haskell.....	574
HTML.....	574
Inform.....	574
IDL (Interface Definition Language).....	575
Java.....	575
Lace.....	576
Lex.....	576
Lisp.....	576
Lite.....	576
LPC.....	577
LUA.....	577
Mail.....	577
Make.....	577
Maple.....	577
Mathematica.....	578

The Vim Tutorial and Reference

Moo.....	578
MSQL.....	578
NCF.....	578
Nroff.....	579
OCAML.....	579
Papp.....	579
Pascal.....	579
Perl.....	580
Php3/ Php4.....	580
PlainTex.....	581
PPWizard.....	581
Phtml.....	581
PostScript.....	581
Printcap and Termcap.....	582
Progress.....	582
Python.....	582
Quake.....	582
ReadLine.....	582
Rexx.....	583
Ruby.....	583
Scheme.....	583
SDL.....	583
Sed.....	583
SGML.....	584
Shell.....	584
Speedup.....	584
TCsh.....	584
TeX.....	584
TinyFugue.....	585
Vim.....	585
XF86Config.....	585
Xml.....	585
Chapter 30:How to Write a Syntax File.....	586
Basic Syntax Commands.....	586
Defining Matches.....	587
Defining Regions.....	587
Nested Regions.....	588
Multiple Group Options.....	589
Transparent Matches.....	590
Other Matches.....	591
Match Groups.....	591
Match Offsets.....	592
Clusters.....	593

The Vim Tutorial and Reference

Including Other Syntax Files.....	593
Listing Syntax Groups.....	594
Synchronization.....	594
Adding Your Syntax File to the System.....	596
Option Summary.....	596
 Appendix A: Installing Vim.....	598
UNIX.....	598
Unpacking the sources.....	598
Running configure.....	598
Dealing with common installation problems.....	601
Installation for Each UNIX User.....	603
Installing on Microsoft Windows.....	603
Common Installation Problems and Questions.....	603
I Do Not Have Root Privileges. How Do I Install Vim? (UNIX).....	603
The Colors Are Not Right on My Screen. (UNIX).....	604
I Am Using RedHat Linux. Can I Use the Vim That Comes with the System?.....	604
How Do I Turn Syntax Coloring On? (All).....	604
What Is a Good vimrc File to Use? (All).....	604
 Appendix B: The <> Key Names.....	605
The Function Keys.....	605
Line Endings.....	605
Editing Keys.....	605
Arrow Keys.....	605
Keypad Keys.....	605
VT100 Special Keys.....	605
Printable Characters.....	606
Other Keys.....	606
Termcap Entries.....	606
Mouse Actions.....	606
Modifiers.....	606
Mouse Modifiers.....	606
 Appendix C: Normal-Mode Commands.....	608
Motion Commands.....	631
 Appendix D: Command-Mode Commands.....	633
:map Mode Table.....	732
Modes.....	733
 Appendix E: Visual-Mode Commands.....	734
Visual Block Commands.....	736

The Vim Tutorial and Reference

Starting Select Mode.....	736
Select Mode Commands.....	736
Appendix F: Insert Mode Commands.....	738
Appendix G: Option List.....	742
- A -.....	742
- B -.....	743
- C -.....	744
- D -.....	746
- E -.....	747
- F -.....	748
- G -.....	750
- H -.....	751
- I -.....	751
- J -.....	753
- K -.....	753
- L -.....	753
- M -.....	754
- N -.....	756
- O -.....	756
- P -.....	757
- Q -.....	758
- R -.....	758
- S -.....	759
- T -.....	763
- U -.....	766
- V -.....	766
- W -.....	767
Appendix H: Vim License Agreement.....	770
Author's Note.....	771
Appendix I: Basic Vim Quick Reference.....	772
Appendix J: Vim Quick Reference.....	773
Minimum Command Set. Learn This First.....	773
Editing Commands.....	773
Getting Out.....	773
Note: On Linux and Unix you must enable the Vim command set.....	773
Vertical Movement / Scrolling.....	774
Screen Location.....	775
Screen Redrawing.....	776
Virtual movement.....	777

The Vim Tutorial and Reference

Commands for English Text.....	778
Text Movement.....	778
Horizontal Movement.....	779
Changing text.....	781
Windows.....	784
Others.....	787
Multiple Files.....	788
Searching.....	789
Search Pattern Reference.....	790
Simple atoms.....	790
Character Classes.....	790
Modifiers.....	791
Grouping and Repeats.....	791
Sets of Characters.....	791
Anchors.....	792
Repeats and Wildcards.....	792
Choices.....	793
Zero Width Conditionals.....	793
For programmers.....	794
Program searches.....	795
Text selection.....	797
Display options.....	799
Diff mode.....	800
Folding.....	801
Misc commands.....	802

Introduction

I'm not sure this book is a labor of love or a love of labor. It certainly is the longest book I've written. When I first started using *Vim* I noticed that there were a "few" commands that *Vim* had that the old *Vi* editor didn't have. So I decided to write a book which would serve as documentation to this wonderful editor.

As part of my preparation I checked existing books for the *Vi* editor. They were about 150-200 pages long, so I figured that my book would turn out about 250-300 pages. Turns out that *Vim* has a *lot* more features than expected and as you can see the book is over 800 pages.

The goal of this book is to provide a tutorial to show the reader how to use the power of Vim to solve common problems. Also the book attempts to show you visually the operation of very major command and option, and to fully document the rest.

Unfortunately, I've had to impose some limits on the book. This book does not cover editing in any language but English mostly due to the fact that English is the only language I know. Also it does not cover the interfaces to external tools which I do not possess.

Copyright and License Information

This book makes use of material from *The Vim (Vim Improved) Book* published by Newriders and copyright 2001 which was published under the Open Publication License. (<http://www.opencontent.org/openpub/>).

This book is copyrighted by Steve Oualline and published under the same license. Basically what license states is that you can use the material in this book for your own use as long as you give credit to the sources (this book and the original *Vim (Vi Improved) Book*.)

A downloadable version of this book is available from the author's web site (<http://www.oualline.org>.)

1

Part I

2

Tutorial

Chapter 1: Basic Editing

The *Vim* editor is one of the most powerful text editors around. It is also extremely efficient, enabling the user to edit files with a minimum of keystrokes. This power and functionality comes at a cost, however: when getting started, users face a steep learning curve. This chapter teaches you the basic set of 10 *Vim* commands you need to get started editing. In this chapter, you learn the following:

- The four basic movement commands
- How to insert and delete text
- How to get help (very important)
- Exiting the editor

After you get these commands down pat, you can learn the more advanced editing commands.

Before You Start

If you have not installed *Vim*, you need to read *Appendix A: Installing Vim* and install the editor. (If *Vim* came with your Linux system, read *Appendix A: Installing Vim* because you may not get the full editor by default.)

If you are running on UNIX or Linux, execute the following command:

```
$ touch ~/.vimrc
```

By creating a *~/.vimrc*, you tell *Vim* that you want to use it in *Vim* mode. If this file is not present, *Vim* runs in *Vi*-compatibility mode and you lose access to many of the advanced *Vim* features. (Most Linux distributions now come with system initialization files that turn on *Vim*'s advanced features. However, you can't count on this.)

You also can enable the advanced features from within *Vim* at any time with this command:

```
:set nocompatible
```

You can also use the command:

```
:set nocp
```

The Vim Tutorial and Reference

Note: Both these commands have an implied <enter> at the end. Also the '**compatible**' option like most *Vim* options has a long form ('**compatible**') and a short form ('cp'). For clarity the long form is used in the examples in this book.

If you are running on Microsoft Windows, the installation process creates the Microsoft Windows version of this file, *_vimrc*, for you.

Running Vim for the First Time

To start *Vim*, enter this command:

```
$ gvim file.txt
```

Note that the \$ is the default UNIX command prompt. Your prompt might differ. If you are running Microsoft Windows, open an MS-DOS prompt window and enter this command:

```
C:> gvim file.txt
```

(Again, your prompt may differ.) In either case, *Vim* starts editing a file called *file.txt*. Because this is a new file, you get a blank window. Figure 1-1 shows what your screen will look like. The tilde (~) lines indicate lines not in the file. In other words, when *Vim* runs out of file to display, it displays tilde lines. At the bottom of a screen, a message line indicates the file is named *file.txt* and shows that you are creating a new file. The message information is temporary and other information overwrites it when you type the first character.

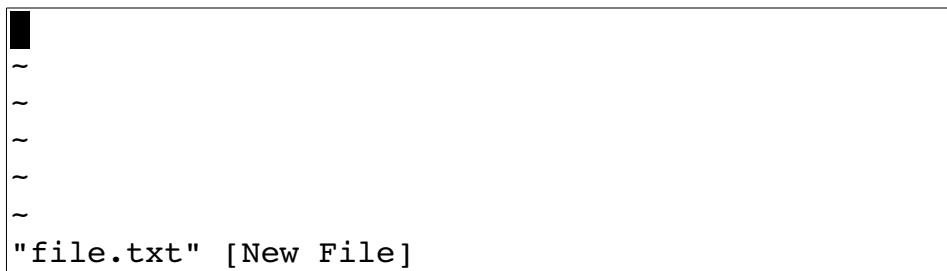


Figure 1-1: Initial Vim window.

The vim Command

The *gvim* command causes the editor to create a new window for editing. If you use the command *vim*, the editing occurs inside your command window. In other words, if you are running inside an *xterm*, the editor uses your *xterm* window. If you are using an MS-DOS command prompt window under Microsoft Windows, the editing occurs inside the window. Figure 1-2 shows a typical MS-DOS command prompt window.

The Vim Tutorial and Reference

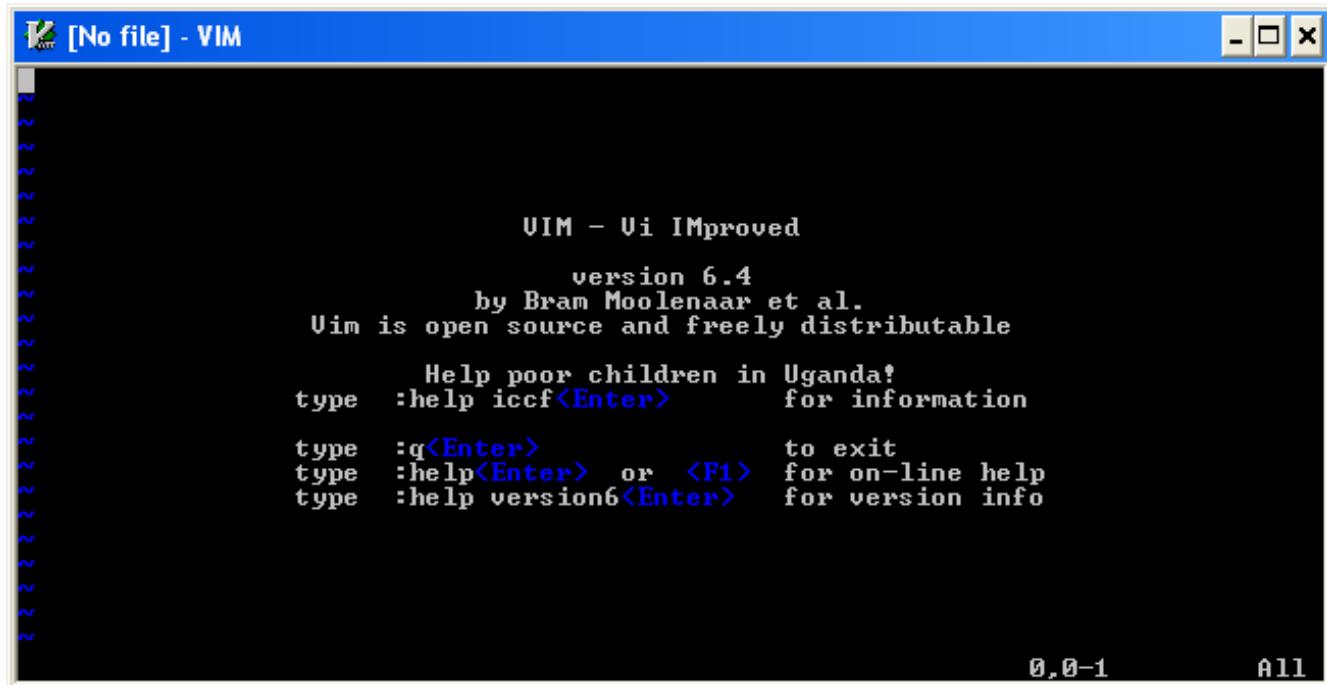


Figure 1-2:Editing with the vim command in an MS-DOS window.

Modes

The *Vim* editor is a modal editor. That means that the editor behaves differently, depending on which mode you are in. If the bottom of the screen displays the filename or is blank, you are in normal mode. If you are in insert mode, the indicator displays **--INSERT--**; and if you are in visual mode, the indicator shows **--VISUAL--**.

Editing for the First Time

The next few sections show you how to edit your first file. During this process, you learn the basic commands that you have to know to use *Vim*. At the end of this lesson, you will know how to edit--not fast, not efficiently, but enough to get the job done.

Inserting Text

To enter text, you need to be in insert mode. Type **i**, and notice that the lower left of the screen changes to **--INSERT--** (meaning that you are in insert mode).

The Vim Tutorial and Reference

Now type some text. It will be inserted into the file. Do not worry if you make mistakes; you can correct them later. Enter the following programmer's limerick:

```
A very intelligent turtle  
Found programming UNIX a hurdle  
    The system, you see,  
    Ran as slow as did he,  
And that's not saying much for the turtle.
```

After you have finished inserting, press the `<Esc>` key. The **--INSERT--** indicator goes away and you return to normal mode. (The indicator for this mode is a blank indicator.) Your screen should now look something like Figure 1-3.

```
A very intelligent turtle  
Found programming UNIX a hurdle  
    The system, you see,  
    Ran as slow as did he,  
And that's not saying much for the turtle.█  
~  
~  
~  
~
```

Figure 1-3: Screen after the text has been inserted.

Getting Out of Trouble

One of the problems for *Vim* novices is mode confusion, which is caused by forgetting which mode you are in or by accidentally typing a command that switches modes. To get back to normal mode, no matter what mode you are in, press the `<Esc>` key.

Moving Around

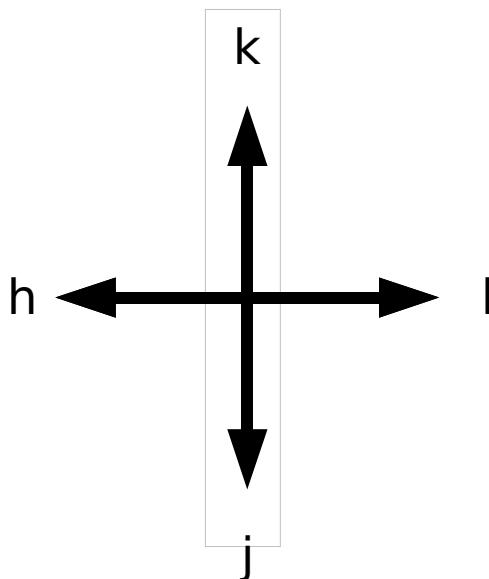
After you return to normal mode, you can move around by using these keys: **h** (left), **j** (down), **k** (up), and **l** (right). At first, it may appear that these commands were chosen at random. After all, who ever heard of using **l** for right? But actually, there is a very good reason for these choices: Moving the cursor is the most common thing you do in an editor, and these keys are on the home row of your right hand. In other words, these commands are placed where you can type them the fastest.

The Vim Tutorial and Reference

Note You can also move the cursor by using the arrow keys. If you do, however, you greatly slow down your editing--because to press the arrow keys, you must move your hand from the text keys to the arrow keys. Considering that you might be doing it hundreds of times an hour, this can take a significant amount of time. If you want to edit efficiently, use **h**, **j**, **k**, and **l**.

Also, there are keyboards which do not have arrow keys, or which locate them in unusual places; therefore, knowing the use of these keys helps in those situations.

One way to remember these commands is that **h** is on the left, **l** is on the right, **j** is a hook down, and **k** points up. Another good way to remember the



commands is to copy this information on a Post-It Note and put it on the edge of your monitor until you get used to these commands. (A quick reference for novices can be found in *Appendix I: Basic Vim Quick Reference*.)

Is there some way of doing the reference above automatically?

Aliases

The alias for the **h** command include **<Left>**, **<BS>**, **CTRL-H** and **CTRL-K**. The aliases for **j** are **<Down>**, **<NL>**, **CTRL-J**, and **CTRL-N**. For **k** we have **<Up>** and **CTRL-P**. Finally for **l** we can also use **<Right>** and **<Space>**.

Deleting Characters

To delete a character, move the cursor over it and type **x**. (This is a throwback to the old days of the typewriter, when you deleted things by typing **xxxx** over them.) Move the cursor to the beginning of the first line, for example, and type **xxxxxxxx** (seven x's) to delete the first seven characters on the line. Figure 1-4 shows the result. To enter a correction, type **iA young <Esc>**. This begins an insert (the **i**), inserts the words **A young**, and then exits insert mode (the final **<Esc>**). Figure 1-5 shows the results.

```
intelligent turtle
Found programming UNIX a hurdle
    The system, you see,
    Ran as slow as did he,
And that's not saying much for the turtle.
~
~
~
~
```

Figure 1-4: Screen after delete (**xxxxxxxx**).

```
A young intelligent turtle
Found programming UNIX a hurdle
    The system, you see,
    Ran as slow as did he,
And that's not saying much for the turtle.
~
~
~
~
```

Figure 1-5: Note Result of the insert.

Note: *Vim* is a text editor. By default, it does not wrap text. If you type past the edge of the screen, the line will appear to wrap, but what you really have is a long line wrapped so that *Vim* can display it. If you want multiple lines you must end each line by pressing the **<Enter>** key. If you don't and just keep typing when you reach the right margin, all you will do is insert a very long line into the editor. You will not automatically go to the next line. To do so, you need to press the **<Enter>** key. (This is the default mode of operation. You can configure the *Vim* editor to word wrap, however, as discussed in *Chapter 11: Dealing with Text Files*)

Undo and Redo

Suppose you delete too much. Well, you could type it in again, but an easier way exists. The **u** command undoes the last edit. Take a look at this in action. Move the cursor to the **A** in the first line. Now type **xxxxxx** to delete A young. The result is as follows:

```
intelligent turtle
```

Type **u** to undo the last delete. That delete removed the **g**, so the undo restores the character.

```
g intelligent turtle
```

The next **u** command restores the next-to-last character deleted:

```
ng intelligent turtle
```

The next **u** command gives you the **u**, and so on:

```
ung intelligent turtle  
oung intelligent turtle  
young intelligent turtle  
  young intelligent turtle  
A young intelligent turtle
```

If you undo too many times, you can press **CTRL-R** (redo) to reverse the preceding command. In other words, it undoes the undo. To see this in action, press **CTRL-R** twice. The character **A** and the space after it disappear.

```
  young intelligent turtle
```

There's a special version of the undo command, the **u** (undo line) command. The undo line command undoes all the changes made on the last line that was edited. Typing this command twice cancels the preceding **u**.

Note: If you are an old *Vi* user, note that the multilevel undo of *Vim* differs significantly from the single level available to a *Vi* user.

Note: Throughout this book we assume that you have turned off *Vi* compatibility. See the first section of this chapter for more information. If compatibility is turned on the **u** command provides one level of undo.

Lets see how the **u** command works. We'll start with the line:

```
A very intelligent turtle
```

Type **xxxxx** to delete “very<space>”.

```
A intelligent turtle
```

The Vim Tutorial and Reference

Move over to the "t" in "turtle and delete "turtle" with the command **xxxxx**.

A intelligent█

Restore line with the **u** command.

A very intelligent turtle

Second U undoes the preceding **u**.

A intelligent█

Getting Out

To exit, use the **zz** command. This command writes the file (if modified) and exits. Unlike many other editors, *Vim* does not automatically make a backup file. If you type **zz**, your changes are committed and there's no turning back. (You can configure the *Vim* editor to produce backup files, as discussed in *Chapter 14: File Recovery and Command-Line Arguments*)

Discarding Changes

Sometimes you will make a set of changes and suddenly realize you were better off before you started. Don't worry; *Vim* has a "quit-and-throw-things-away" command. It is **:q!**. For those of you interested in the details, the three parts of this command are the colon (:), which enters command mode; the **q** command, which tells the editor to quit; and the override command modifier (!). The override command modifier is needed because *Vim* is reluctant to throw away changes. Because this is a command mode command, you need to type **<Enter>** to finish it. (All command mode commands have **<Enter>** at the end. This is not shown in the text.) If you were to just type **:q**, *Vim* would display an error message and refuse to exit:

No write since last change (use ! to override)

By specifying the override, you are in effect telling *Vim*, "I know that what I'm doing looks stupid, but I'm a big boy and really want to do this."

Note: The **:q** command can also be written as **:quit**. From on when we have a command with more than one spelling we will show the alternate spellings in parenthesis after the first mention of the command. For example: "To quit use the **:q** (**:quit**) command. I say again use **:q** to get out."

Other editing Commands

Now that you have gone through a few simple commands, it is time to move on to some slightly more complex operations.

Inserting Characters at the End of a Line

The **i** command inserts a character before the character under the cursor. That works fine; but what happens if you want to add stuff to the end of the line? For that you need to insert text after the cursor. This is done with the **a** (append) command.

For example, to change the line

and that's not saying much for the turtle.

to

and that's not saying much for the turtle!!!

move the cursor over to the dot at the end of the line. Then type **x** to delete the period. The cursor is now positioned at the end of the line on the **e** in turtle:

and that's not saying much for the turtle

Now type **a!!!<Esc>** to append three exclamation points after the **e** in turtle:

and that's not saying much for the turtle!!!

Deleting a Line

To delete a line, use the **dd** command, which deletes the line on which the cursor is positioned. To delete the middle line of this example, for instance, position the cursor anywhere on the line **The system, you see,** as shown in Figure 1-6. Now type **dd**. Figure 1-7 shows the results.

```
A very intelligent turtle
Found programming UNIX a hurdle
    The system, you see,
        Ran as slow as did he,
    And that's not saying much for the turtle!!!
~
```

Figure 1-6: Screen before dd command.

The Vim Tutorial and Reference

```
A very intelligent turtle
Found programming UNIX a hurdle
    Ran as slow as did he,
And that's not saying much for the turtle!!!
~
~
~
~
```

Figure 1-7 Screen after **dd** command.

Opening Up New Lines

To add a new line, use the **o** (lower case) command to open up a new line below the cursor. The editor is then placed in insert mode.

Suppose, for example, that you want to add a line to the sample text just below the third line. Start by leaving the cursor on the *Ran as slow. . .* line, as seen in Figure 1-7. Now type **o** to open up a new line. Enter the text for the line and then press **<Esc>** to end insert mode. Figure 1-8 shows the results. If you want to open a line above the cursor, use the **O** (uppercase) command.

```
A very intelligent turtle
Found programming UNIX a hurdle
    Ran as slow as did he,
█
And that's not saying much for the turtle!!!
~
~
~
-- INSERT --
```

Figure 1-8: Screen after using the **o** command.

Help

Finally, there's one more important command, the **:help** (**:h**, **<F1>**, **<Help>**) command. To get help, enter the following:

```
:help
```

(Remember the implied **<Enter>** for command-mode commands.) This displays a general help window, as seen in Figure 1-9.

The Vim Tutorial and Reference

```
*help.txt*      For Vim version 6.3. Last change: 2004 May 04

          VIM - main help file

Move around: Use the cursor keys, or "h" to go left,           k
              "j" to go down, "k" to go up, "l" to go right.   h l
Close this window: Use ":q<Enter>".
Get out of Vim:   Use ":qa!<Enter>" (careful, all changes are lost!). j

Jump to a subject: Position the cursor on a tag between |bars| and hit CTRL-].
With the mouse:   ":set mouse=a" to enable the mouse (in xterm or GUI).
                  Double-click the left mouse button on a tag between |bars|.
Jump back:        Type CTRL-T or CTRL-O (repeat to go further back).

Get specific help: It is possible to go directly to whatever you want help
                   on, by giving an argument to the ":help" command |:help|.
                   It is possible to further specify the context:
                           *help-context*
                   WHAT           PREPEND     EXAMPLE   ~
                   Normal mode commands (nothing) :help x

help.txt [help][RO]
[No File] [+]
```

Figure 1-9: Help screen.

If you don't supply a subject, **:help** displays the general help window. The creators of *Vim* did something very clever (or very lazy) with the help system. They made the help window a normal editing window. You can use all the normal *Vim* commands to move through the help information. Therefore **h**, **k**, **j**, and **l** move left, up, down, right, and so on.

To get out of the help system, use the same command you use to get out of the editor: **zz**.

As you read the help text, you will notice some text enclosed in vertical bars (for example, |**:help**|). If you are using the GUI this text is colored cyan.¹ This indicates a hyperlink. If you position the cursor anywhere between the bars and press **CTRL-]** (jump to tag), the help system takes you to the indicated subject. (For reasons not discussed here, the *Vim* terminology for a hyperlink is tag. So **CTRL-]** jumps to the location of the tag given by the word under the cursor.)

¹ It's still enclosed in vertical bars but they are colored white, the same color as the background which makes them very hard to see.

The Vim Tutorial and Reference

After a few jumps, you might want to go back. **CTRL-T** (pop tag) takes you back to the preceding screen. Or in *Vim* terms, it "pops a tag off the tag stack." At the top of this screen, there is the notation ***help.txt***. This is used by the help system to define a tag (hyperlink destination). *Chapter 7: Commands for Programmers* explains tags in detail. To get help on a given subject, use the following command:

```
:help subject
```

To get help on the **x** command, for example, enter the following:

```
:help x
```

To find out how to delete text, use this command:

```
:help deleting
```

To get a complete index of what is available, use the following command:

```
:help index
```

When you need to get help for a control character command (for example, **CTRL-A**), you need to spell it with the prefix **CTRL-**.

```
:help CTRL-A
```

The *Vim* editor has many different modes. By default, the help system displays the normal-mode commands. For example, the following command displays help for the normal-mode **CTRL-H** command:

```
:help CTRL-H
```

To identify other modes, use a mode prefix. If you want the help for the insert-mode version of this command, prefix the key with **i_**. This gives you the following command:

```
:help i_CTRL-H
```

The table below lists several other mode prefixes.

When you start *the Vim* editor, you can use several command-line options. These all begin with a dash (-). To find what the **-t** command-line option does, for example, use the command

```
:help -t
```

The *Vim* editor has a number of options that enable you to configure and customize the editor. If you want help for an option, you need to enclose it in single quotation marks. To find out what the **number** option does, for example, use the following command:

```
:help 'number'
```

The Vim Tutorial and Reference

The following table summarizes the special prefixes.

What	Prefix	Example
Normal-mode commands	(nothing)	<code>:help x</code>
Control Character	<code>CTRL-</code>	<code>:help CTRL-u</code>
Visual-mode commands	<code>v_</code>	<code>:help v_d</code>
Insert-mode commands	<code>i_</code>	<code>:help i_<Esc></code>
<i>ex</i> -mode commands	<code>:</code>	<code>:help :quit</code>
Command-line edit	<code>c_</code>	<code>:help c_</code>
<i>Vim</i> command arguments	<code>-</code>	<code>:help -r</code>
Options	' (both ends)	<code>:help 'nowrap'</code>

Special keys are enclosed in angle brackets. To find help on the up-arrow key, for instance, use this command:

```
:help <Up>
```

Appendix B: The <> Key Names provides a complete list of the key names.

Help Language

By default *Vim* will set the language for the help file to the current language of your system. (From the locale settings.) If you wish to use another language, use the '`helplang`' ('`hl`') option. This option contains a series of languages to search for help text.

Note: *Vim* always searches English as a last resort.

So to search for help in German, Italian, and then English use the command:

```
:set helplang=de,it
```

Other Ways to Get Help

You can get to the help screen by pressing the `<F1>` key. This displays the general help screen, and you can navigate from there. If your keyboard has a `<Help>` key, you can use it as well.

Using a Count to Edit Faster

Suppose you want to move up nine lines. You can type **kkkkkkkkk** or you can enter the command **9k**. In fact, you can precede almost all the movement commands with a number. Earlier in this chapter, for instance, you added three exclamation points to the end of a line by typing **a!!!<Esc>**. Another way to do this is to use the command **3a!<Esc>**. The count of **3** tells the **a** command to insert what follows (!) three times. Similarly, to delete three characters, use the command **3x**.

The Vim Tutorial

The UNIX version of the *Vim* editor comes with an interactive tutorial. Lesson 1 covers many of the commands described in this chapter. To invoke the tutorial on UNIX, use the following command:

```
$ vimtutor
```

The tutorial starts by explaining the movement commands so that you can move through the tutorial. After that it gradually introduces more complex commands. If you are on a non-Unix system, execute the command

```
:help tutor
```

for information on how to get the *Vim* tutorial working on your system (it isn't difficult).

Summary

You now know enough to edit with *Vim*. Not well or fast, but you can edit. Take some time to practice with these commands before moving on to the next chapter. After you absorb these commands, you can move on to the more advanced commands that enable you to edit faster and easier.

Chapter 2: Editing a Little Faster

The basic commands covered in *Chapter 1: Basic Editing* enable you to edit text. This chapter covers some additional commands that enable you to edit more efficiently. These commands include the following:

- Additional movement commands
- Quick searches along a single line
- Additional delete and change commands
- The repeat command
- Keyboard macros (how to record and play back commands)
- Digraphs

One of the things I noticed as I wrote this chapter is the amazing number of different ways you can move through a file. Although I have been using *Vi* and now *Vim* as my main editor for the past 15 years, I have never bothered to learn all of them. I get by with the 10% I like.

There are lots of different ways of doing things in *Vim*. This chapter discusses one useful selection of all the possible commands.

Word Movement

Let's start with movement. To move the cursor forward one word, use the **w** command. The **b** command moves backward one word. Like most *Vim* commands, you can use a numeric prefix to move past multiple words. For example, **4b** moves back four words. Figure 2-1 shows how these commands work.

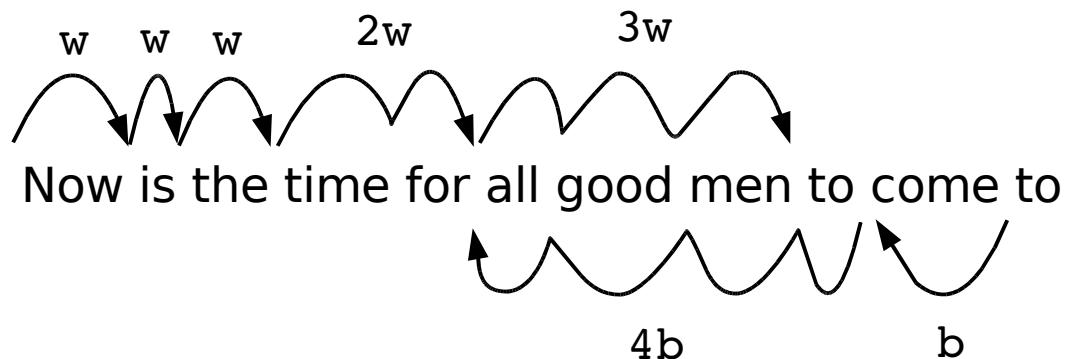


Figure 2-1: Word movement.

Moving to the Start or End of a Line

The `$` command moves the cursor to the end of a line. Actually, a bunch of keys map to the "end-of-line" command. The *Vim* names for these keys are `$`, `<End>`, and `<kEnd>`. (The `<kEnd>` key is *Vim*'s name for the keypad End key.)

The `$` command takes a numeric argument as well. If present, it causes the editor to move to the end of the next line. For example, `1$` moves you to the end of the first line (the one you're on), `2$` to the end of the next line, and so on. Figure 2-2 illustrates how this command works.

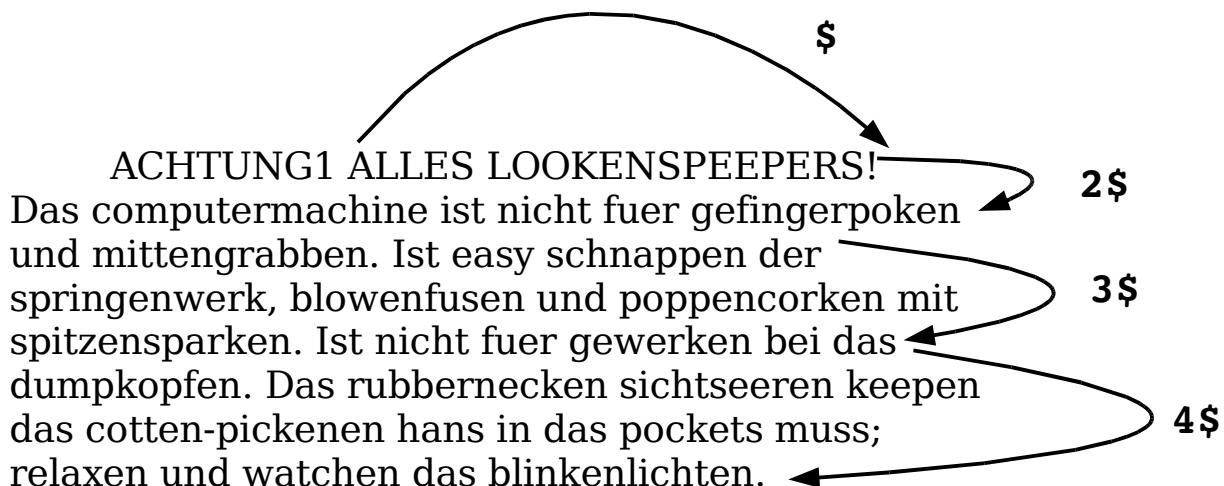


Figure 2-2 The `$` command.

The Vim Tutorial and Reference

The `^` command moves to the first nonblank character of the line. The `<Home>` or `<kHome>` key moves to the first character of the line, as seen in Figure 2-3. (The `0` [zero] command does the same thing.) Like every other command previously discussed, these commands (except `0` [zero]) can take a numeric argument. They do not do anything with it, but you can specify it if you want to.

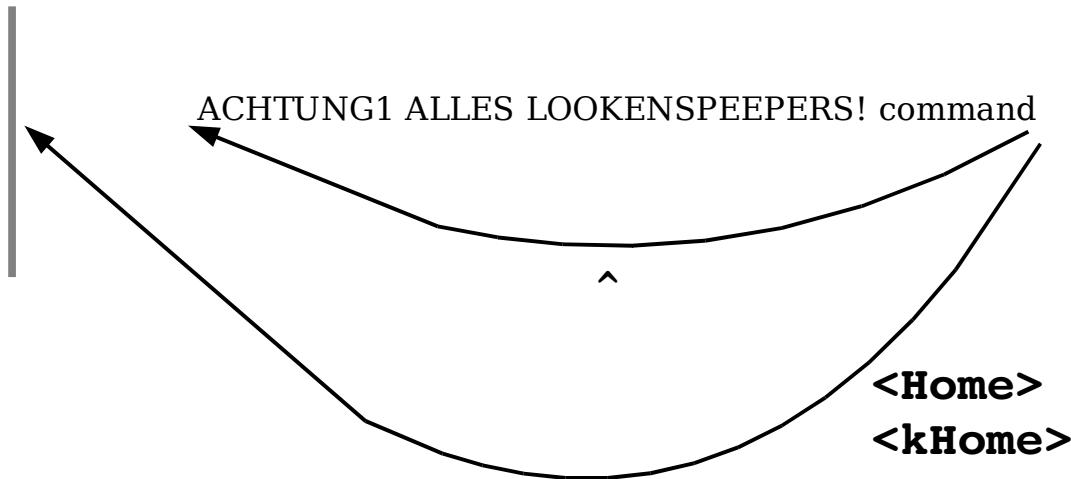


Figure 2-3: The `^` and `<Home>` commands.

Searching Along a Single Line

Moving is the most common editing activity you do. One of the most useful movement commands is the single-character search command. The command `fx` (forward search) searches the line for the single character `x`. Suppose, for example, that you are at the beginning of the following line:

To err is human. To really foul up you need a computer.

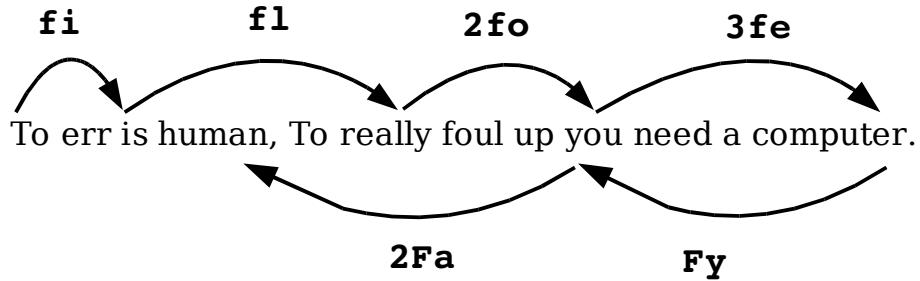
Suppose you want to go to the `h` of `human`. Just execute the command `fh` and the cursor will be positioned over the `h`:

To err is **h**uman. To really foul up you need a computer.

To go to the end of the word “really”, use the command `fy`. You can specify a count; therefore, you can space forward five words by using the command `5f<Space>`. Note: this only moves five space characters, not five words. If there are multiple spaces between words, this will not move five words!

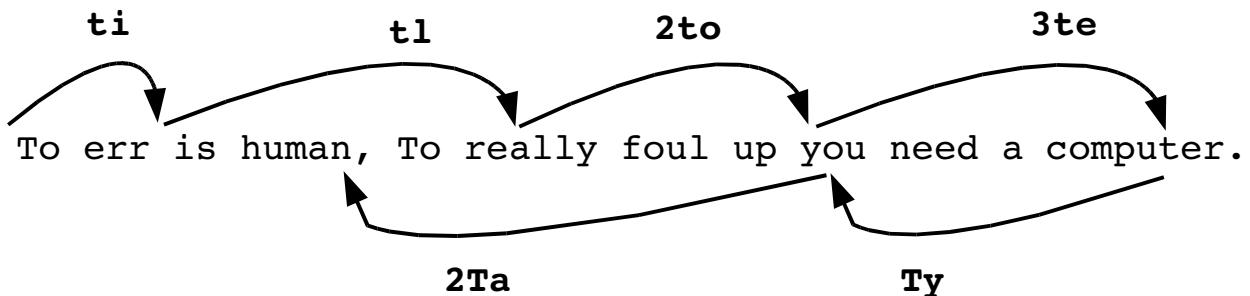
To err is human. To really foul**up** you need a computer.

The `F` command searches to the left. Figure 2-4 shows the effect of the `f` and `F` commands.



*Figure 2-4: Operation of the **f** and **F** commands*

The **tx** (search `til) command works like the **fx** command, except it stops one character before the indicated character. The backward version of this command is **Tx**. Figure 2-5 shows how these commands work.



*Figure 2-5:Operations of the **f** and **F** commands.*

Sometimes you will start a search, only to realize that you have typed the wrong command. You type **f** to search backward, for example, only to realize that you really meant **F**. To abort a search, press **<Esc>** as the search key. So **f<Esc>** is an aborted forward search. (Note: **<Esc>** cancels most operations, not just searches.)

Moving to a Specific Line

If you are a C or C++ programmer, you are familiar with error messages such as the following:

```
prog.c:3: 'j' undeclared (first use in this function)
```

This tells you that you might want to fix something on line 3. So how do you find line 3?

The Vim Tutorial and Reference

One way is to do a **9999k** to go to the top of the file and a **2j** to go down two lines. It is not a good way, but it works. A much better way of doing things is to use the **G** command. With an argument, this command positions you at the given line number. For example, **3G** puts you on line 3. (Likewise, use the **1G** command to go to the top of the file rather than **9999k**.) With no argument, it positions you at the end of the file. (For a better way of going through a compiler's error list, see *Chapter 7: Commands for Programmers*, for information on the **:make** and **:clist** related commands.)

Another way to move through the file is with the **%** command. The **50%** command moves you through the file, **25%** a quarter, **90%** ninety percent, etc. (**%** by itself finds the matching **{}** which is a different command.)

Telling Where You Are in a File

How do you really know where you are in a file? You can do so in several ways. The first is to turn on line numbering with the following command (see Figure 2-6):

```
:set number
```

This causes line numbers to appear at the beginning of each line.

```
1176 Ode to a maintenance programmer
1177 =====
1178
1179 Once more I travel that lone dark road
1180 into someone else's impossible code
1181 Through "if" and "switch" and "do" and "while"
1182 that twist and turn for mile and mile
1183 Clever code full of traps and tricks
1184 and you must discover how it ticks
1185 And then I emerge to ask anew,
1186 "What the heck does this program do?"
1187
1188 ****
```

Figure 2-6: Window with numbering turned on.

Note: These line numbers are for your information only; they are not written into the file when you exit.

The *Vim* editor is highly configurable and has a huge number of options. You can use the **:set** command in many different ways, which are described in *Chapter 28: Customizing the Editor*. The '**number**' option is a boolean option, meaning that it can be on or off. To turn it on, use this command:

The Vim Tutorial and Reference

```
:set number
```

To turn it off, use this command:

```
:set nonumber
```

The numbers disappear from your screen. Figure 2-7 shows what happens we you do a `:set nonumber`.

```
Ode to a maintenance programmer  
=====
```

Once more I travel that lone dark road
into someone else's impossible code
Through "if" and "switch" and "do" and "while"
that twist and turn for mile and mile
Clever code full of traps and tricks
and you must discover how it ticks
And then I emerge to ask anew,
"What the heck does this program do?"

Figure 2-7 Results of `:set nonumber`.

Where Am I?

The **CTRL-G** command displays a status line that indicates where you are in the file. For example:

```
"c2.txt" [Modified] line 81 of 153 --52%-- col 1
```

This indicates that you are editing a file called *c2.txt*, and that it has been modified since the editing started. The cursor is positioned on line 81 out of a total of 153, or about 52% of the way through the file. The cursor is currently sitting in column 1.

Sometimes you will see a split column number (for example, **col 2-9**). This indicates that the cursor is positioned on character 2. But because character one is a tab, the screen column is 9. Figure 2-8 shows the results of a typical **CTRL-G** command.

The Vim Tutorial and Reference

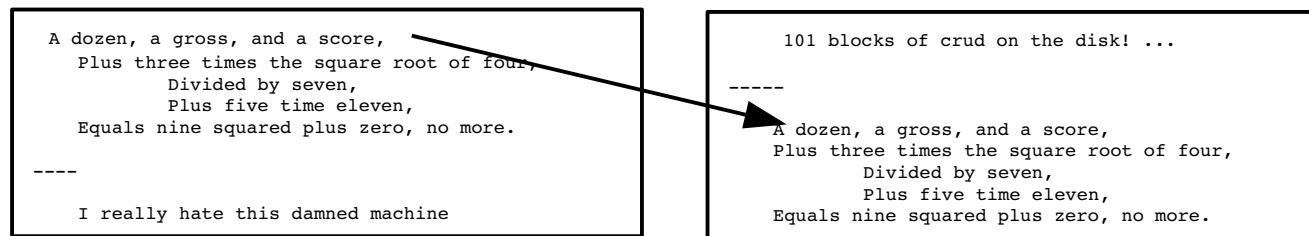
```
to open up the packing crate and find  
the manual. (What did they think we were  
reading anyway?)  
  
(Head)Dumb programmer stories  
  
    Ode to a maintenance programmer  
  
Once more I travel that lone dark road  
into someone else's impossible code  
Through "if" and "switch" and "do" and "while"  
that twist and turn for mile and mile  
  
"j.txt" [Modified] line 186 of 119--16%--col 2-9
```

Figure 2-8 The **CTRL-G** command.

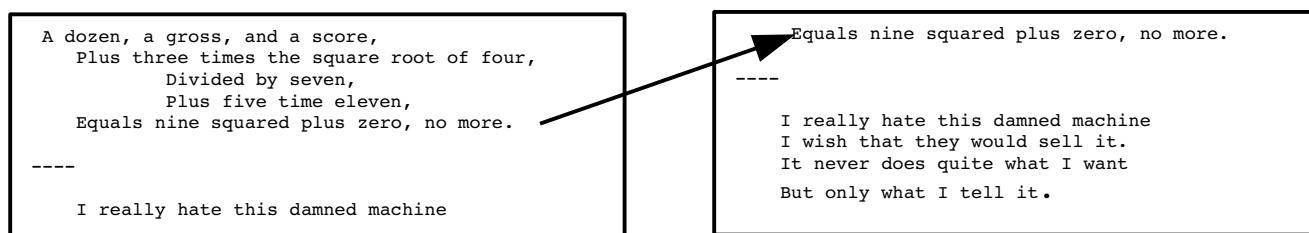
Scrolling Up and Down

The **CTRL-U** command scrolls up half a screen of text. (Up in this case is backward in the file; the text moves down on the screen. Don't worry if you have a little trouble remembering which end is up. Most programmers have the same problem.) The **CTRL-D** command scrolls you down half a screen. Figure 2-9 shows how these two commands work.

CTRL-U



3



CTRL-D

Figure 2-9: Results of the **CTRL-U** and **CTRL-D** commands.

Deleting Text

As you learned in *Chapter 1: Basic Editing*, the **dd** command deletes a line. But suppose you want to delete part of a line, say a word. The way to do that is to start visual mode with the **v** command. You can now highlight the text you want to delete with the cursor movement commands. We want to delete a word, so we pass over it with the **w** command. Unfortunately this leaves us with the cursor positioned on the first character of the next word. We don't want to delete that so we backup with the left (**h**) command.

Finally we need to tell *Vim* what to do with the text, so we enter the **d** command to delete it. Figure 2-10 shows the steps we used:

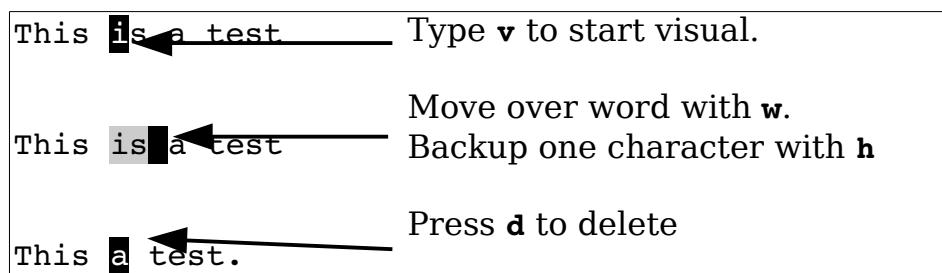


Figure 2-10: Visual mode and delete

Note: To get help about what the delete (**d**) command does in visual mode use the command:

```
:help v_d
```

As we will see in future sections the visual mode gives you a very powerful way of dealing with large blocks of text.

Deleting Text Without Visual Mode

It is possible to delete sections of text without using the visual mode. The advantage of doing things non-visually is that you save a single keystroke. The disadvantage is that you really have to know what you are doing as there is no visual feedback to show you what's going on.

To delete a word in normal mode you use the command **dw**. You may recognize the **w** command as the move word command. In fact, the **d** command may be followed by any motion command, and it deletes from the current location to the place where the cursor winds up. (Therefore, we say the syntax of the **d** command is **dmotion**.)

The Vim Tutorial and Reference

The **3w** command, for example, moves the cursor over three words. The **d3w** command deletes three words, as seen in Figure 2-11. (You can write it as **d3w** or **3dw**; both versions work the same.)

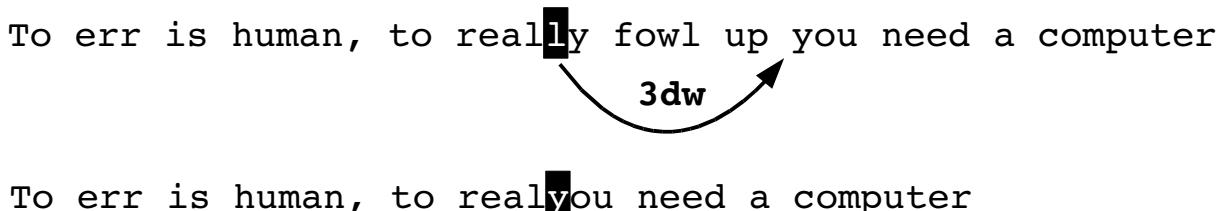


Figure 2-11: The **d3w** command.

The **\$** command moves to the end of a line. The **d\$** command deletes from the cursor to the end of the line, as seen in Figure 2-12. A shortcut for this is the **D** command.

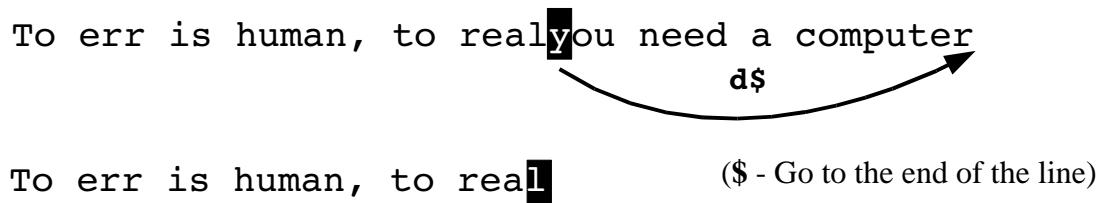


Figure 2-12: The **d\$** command.

Where to Put the Count (**3dw** or **d3w**)

The commands **3dw** and **d3w** delete three words. If you want to get really picky about things, the first command, **3dw**, deletes one word three times; the command **d3w** deletes three words once. This is a difference without a distinction. You can actually put in two counts, however (for example, **3d2w**). This command deletes two words, repeated three times, for a total of six words.

Visual vs. Normal Mode Delete

The visual method of deleting things lets you see exactly what you are going to delete before you delete it. You can also make adjustments *before* you delete the text. This makes things much easier for large complex deletes.

On the other hand, the normal mode delete provides no feedback. You type **d** and a motion command and hope you did the right thing. (If you didn't there's always the undo command.) The advantage of the normal mode method of doing things is that it's quicker for small amounts of text, especially when you know where the motion command is going to send the cursor. In other words if you have to delete three words, **d3w** is probably what you want to use. But if you have to delete 57 words (maybe 57 you're not sure) the **v<move>d** is better.

Changing Text

The **c** command changes text. It acts just like the **d** command, except it leaves you in insert mode. For example, if you go into visual mode (**v**) highlight a word (**w**) and then press **c**, the word will disappear and you'll be left in insert mode.

In the example in Figure 2-13 we change boss to idiot:

```
The boss said ...  
  
The boss said ...  
  
The said ....  
  
The idiot said ...
```

Figure 2-13: The visual c command.

The normal mode version of the **c** command acts like the **d** command except that it leaves you in insert mode. For example, **cw** changes a word. Or more specifically, it deletes a word and then puts you in insert mode. Figure 2-14 illustrates how this command works.

The Vim Tutorial and Reference

There is a saying that for every problem there is an answer that's simple, clear, and wrong. That is the case with the example used here for the **cw** command. The **cmotion** command works just like the **dmotion** command, with one exception: the **cw** and **dw** commands. Whereas **cw** deletes the text up to the space following the word (and then enters insert mode), the **dw** command deletes the word and the space following it.

To err is human, To really foul up you need a computer.



a word (**w**)

Change Word: **screw**



To err is human, To really **screw** up you need a computer.

Cwscrew <Esc>

c - Change

w - Word (the motion)

screw<blank> (text)

<**Esc**> - End insert

Figure 2-14: How cw works.

The **cc** command works on the entire line. That is, it deletes the line and then goes into insert mode. In other words, **cc** works on the current line just like **dd**. Likewise, **c\$** or **c** change from the cursor to the end of the line.

The . Command

The **.** command is one of the most simple yet powerful commands in *Vim*. It repeats the last delete or change command. For instance, suppose you are editing an HTML file and want to delete all the `` tags. You position the cursor on the first `<` and delete the `` with the command `df>`. You then go to the `<` of the next `` and kill it using the **.** command. The **.** command executes the last change command (in this case, `df>`). To delete another tag, position the cursor on the `<` and press the **.** command. Figure 2-15 illustrates how this can work.

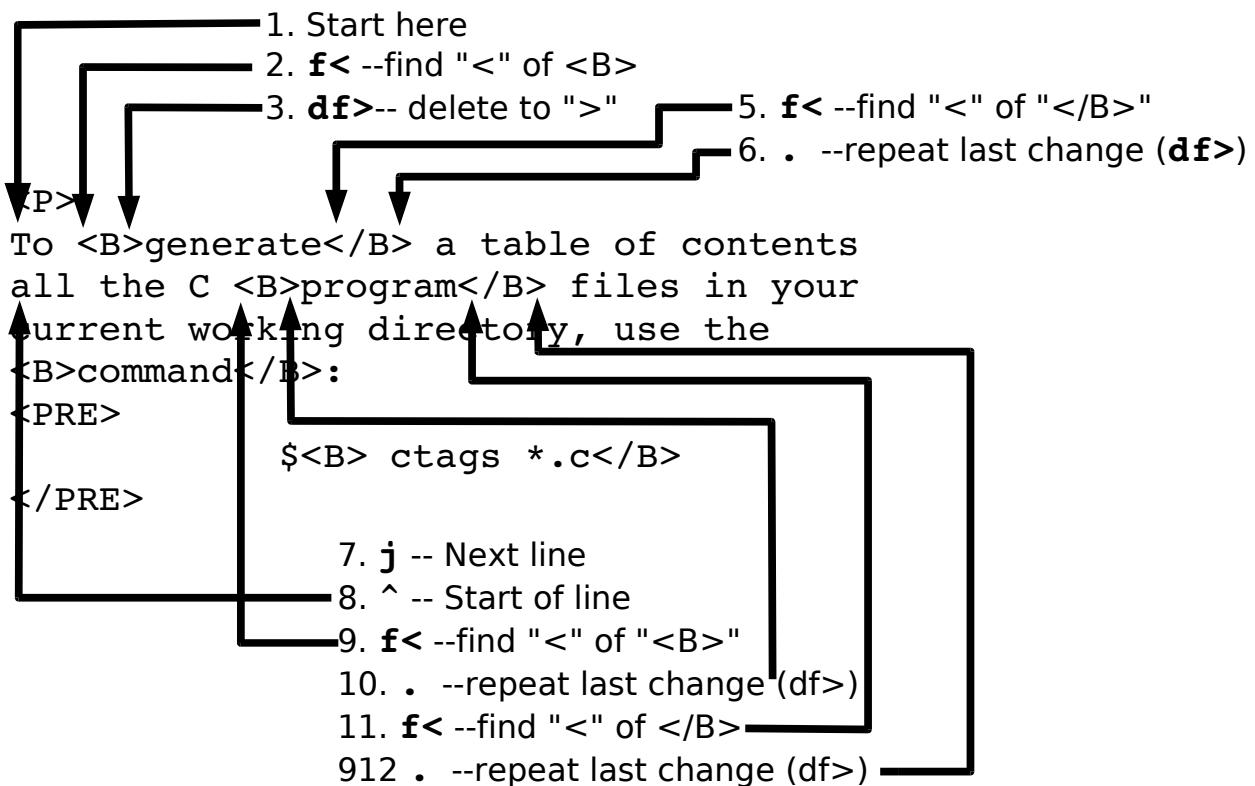


Figure 2-15 Using the **.** command.

Joining Lines

To join a set of lines in visual mode, start visual mode (**v**) highlight the lines you wish to join, and press **J**. All the highlighted lines will be put together in one big line. A space is placed between the pieces that are joined as seen in Figure 2-16. (If there is no space it is added. If there is one space, it is preserved. If there are multiple spaces, they are turned into one.)

The Vim Tutorial and Reference

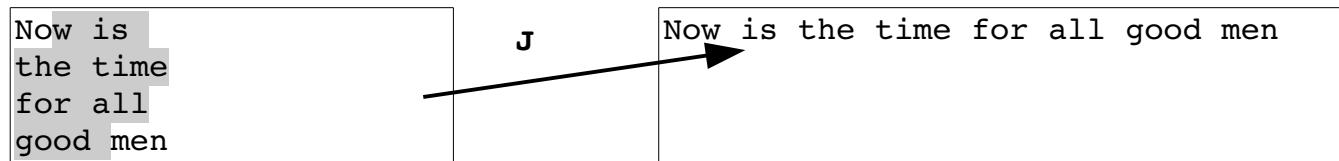


Figure 2-16 Visual **J** command.

In normal mode, the **J** command joins the current line with the next one., as illustrated by Figure 2-17. If a count is specified, then count lines are joined (minimum of two).

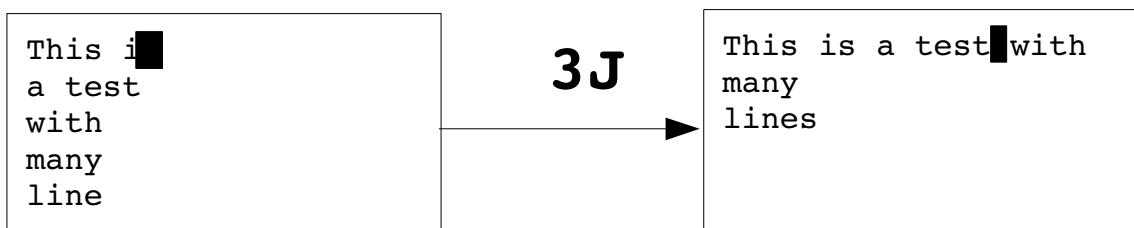
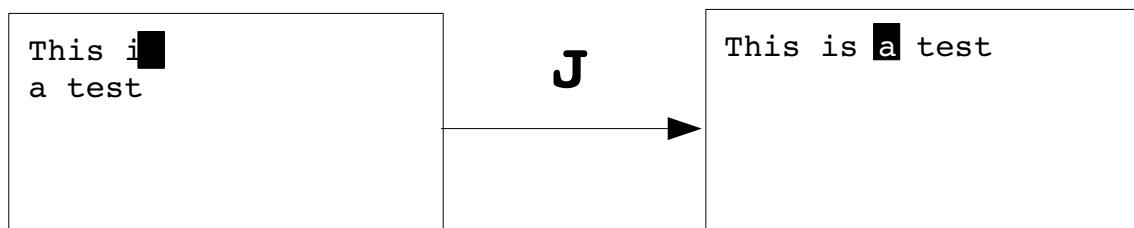


Figure 2-17: The **J** command.

Replacing Characters

The **r{char}** command replaces the character under the cursor with **{char}**. Figure 2-18 shows how you can use the **r** command to replace a **z** with an **s**. The **r** command can be preceded with a count, indicating the number of characters to be replaced. In Figure 2-19, we go to the beginning of line (the **^** command) and execute **5ra** to replace the first five characters with **a**.

This iz a test.
rs ↓

The Vim Tutorial and Reference

This is a test.

Figure 2-18: The replace (**r**) command.

5ra
↓
This is a test.
aaaaais a test.

Figure 2-19: Replace (**r**) command with count.

Note: The **r** command treats <Enter> in a special way. No matter how big the count is, only one <Enter> is inserted. Therefore, **5ra** inserts five a characters, whereas **5r<Enter>** replaces five characters with one <Enter>.

Be careful where you place the count. The **5rx** command replaces five characters with the character x, whereas **r5x** replaces the character under the cursor with 5 (r5) and then deletes a character (x).

Changing Case

The ~ command changes a character's case. It changes uppercase to lowercase and vice versa. If a count is specified, the count characters are changed. Figure 2-20 contains examples.

now is the time. . . .
~
Now is the time. . . .

now is the time . . .
6~
NOW IS the time . . .

Figure 2-20: Use of the ~ command.

Keyboard Macros

The . command repeats the preceding change. But what if you want to do something more complex than a single change? That's where the keyboard macros come in. The **q{character}** command records keystrokes into the register named character. (The character must be between a and z.) To finish recording, just type a **q** command. You can now execute the macro by typing the **@{character}** command. (This can be preceded by a count, which will cause the macro to be executed that number of times.)

The Vim Tutorial and Reference

Take a look at how to use these commands in practice. You have a list of filenames that look like this:

```
stdio.h  
fcntl.h  
unistd.h  
stdlib.h
```

And what you want is the following:

```
#include "stdio.h"  
#include "fcntl.h "  
#include "unistd.h"  
#include "stdlib.h"
```

You start by moving to the first character of the first line. Next you execute the following commands:

Command	Description
qa	Start recording a macro in register a
^	Go to beginning of the line
i#include "<Esc>"	Insert the text #include "
\$	Go to the end of line
a"<Esc>"	Append the character " after the cursor.
j	Go to the next line
q	Quit recording

Now that you have done the work once, you can repeat the change by typing the command **@a**. Alternatively, because you have three lines to go, you can change them using the command **3@a**. Figure 2-21 shows how to define and then execute a macro.

```
stdio.h  
fcntl.h  
unistd.h  
stdlib.h
```

Start

```
#include "stdio.h"  
fcntl.h  
unistd.h  
stdlib.h
```

qa - Record into register a
^ - Go to the beginning of a line
i#include "<Esc>" - Insert text
\$ Go to the end of the line

The Vim Tutorial and Reference



a "<Esc> - Insert more text
j - Go to the next line
q - Stop macro

```
#include "stdio.h"
#include "fcntl.h"
unistd.h
stdlib.h
```

@a - Execute macro "a"

```
#include "stdio.h"
#include "fcntl.h"
#include "unistd.h"
#include "stdlib.h"
```

2@a - Execute macro "a" twice

Figure 2-21: Keyboard Macros

Digraphs

Some characters are not on the keyboard--for example, the copyright character (©). To type these letters in *Vim*, you use digraphs, where two characters represent one. To enter a ©, for example, you type **CTRL-KcO** (Capital letter "O"). To find out what digraphs are available, use the following command:

```
:digraphs
```

Note: :dig is the same as :digraphs

The *Vim* editor will display the digraph-mapping table, as seen in Figure 2-22. This shows, for example, that the digraph you get by typing **CTRL-Kct** is the character (¢).This is character number 162.

Warning: The digraphs are set up assuming that you have a standard ISO-646 character set. Although this is an international standard, your particular display or printing system might not use it.

The Vim Tutorial and Reference

:digraphs															
NU ^@	10	SH ^A	1	SX ^B	2	EX ^C	3	ET ^D	4	EQ ^E	5	AK ^F	6		
BL ^G	7	BS ^H	8	HT ^I	9	LF ^@	10	VT ^K	11	FF ^L	12	CR ^M	13		
SO ^N	14	SI ^O	15	DL ^P	16	D1 ^Q	17	D2 ^R	18	D3 ^S	19	D4 ^T	20		
NK ^U	21	SY ^V	22	EB ^W	23	CN ^X	24	EM ^Y	25	SB ^Z	26	EC ^[27		
FS ^\	28	GS ^]	29	RS ^^	30	US ^_	31	SP	32	Nb #	35	DO \$	36		
At @	64	<([91	// \	92)>]	93	'> ^	94	'! `	96	(! {	123		
!!	124	!) }	125	'? ~	126	DT ^?	127	PA ~@	128	HO ~A	129	BH ~B	130		
NH ~C	131	IN ~D	132	NL ~E	133	SA ~F	134	ES ~G	135	HS ~H	136	HJ ~I	137		
VS ~J	138	PD ~K	139	PU ~L	140	RI ~M	141	S2 ~N	142	S3 ~O	143	DC ~P	144		
P1 ~Q	145	P2 ~R	146	TS ~S	147	CC ~T	148	MW ~U	149	SG ~V	150	EG ~W	151		
SS ~X	152	GC ~Y	153	SC ~Z	154	CI ~[155	ST ~\	156	OC ~]	157	PM ~^	158		
AC ~_	159	NS	160	! I i	161	Ct ¢	162	Pd £	163	Cu ¤	164	Ye ¥	165		
BB	166	SE §	167	' : ..	168	Co ©	169	-a ª	170	<< «	171	NO ¬	172		
-- -	173	Rg ®	174	'm -	175	DG °	176	+ - ±	177	2S ²	178	3S ³	179		
-- '	180	My µ	181	PI ¶	182	.M ·	183	', ,	184	1S ¹	185	-o ¸	186		
>> »	187	14 ¼	188	12 ½	189	34 ¾	190	?I ï	191	A! À	192	A' Á	193		
A> Â	194	A? Ä	195	A: Ä	196	AA Å	197	AE Æ	198	C, Ç	199	E! È	200		
E' É	201	E> Ê	202	E: Ë	203	I! Í	204	I' Í	205	I> Î	206	I: Ï	207		
D- Đ	208	N? Ñ	209	O! Ò	210	O' Ó	211	O> Ô	212	O? Õ	213	O: Ö	214		
*X ×	215	O/ Ø	216	U! Ù	217	U' Ú	218	U> Û	219	U: Ü	220	Y' Ý	221		
TH þ	222	ss ß	223	a! à	224	a' á	225	a> â	226	a? ã	227	a: ä	228		
aa å	229	ae æ	230	c, ç	231	e! è	232	e' é	233	e> ê	234	e: ë	235		
-- More --															

Figure 2-22: Digraph-mapping table.

Chapter 3: Searching

This chapter introduces you to the various *Vim* search commands. The basic search commands in *Vim* are rather simple, which means that you can get started with searching fairly easily. In this chapter, you learn about the following:

- Simple forward searches
- Search options
- Incremental searches
- Changing directions
- Basic regular expressions

Simple Searches

To search for a string, use the **/string** command. To find the word `include`, for example, use the command **/include**. An **<Enter>** is implied at the end of this command. (Any time the cursor jumps to the bottom of the screen and you type something, you must end it with **<Enter>**.)

Note: The characters `.*[]^%/\?~$` have special meaning. If you want to use them in a search you must put a `\` in front of them. Example: to find `$10` use the search command **/\\$10**

In this example, we searched for `include` (**/include**). The cursor now moves to the `i` of `include`, as seen in Figure 3-1.

The Vim Tutorial and Reference

```
*****  
* cd-speed  
* Report the speed of a cd-rom  
*      (Also works on hard drives and other  
*      devices)  
*  
* Usage:  
* cd-speed <device>  
*  
*****  
#include <iostream.h>  
#include <iomanip.h>  
/include
```

Figure 3-1: Searching for `include`.

To find the next `include`, use the command `/<Enter>`. The cursor now moves to the next occurrence of the string, as shown by Figure 3-2.

```
*****  
* cd-speed  
* Report the speed of a cd-rom  
*      (Also works on hard drives and other  
*      devices)  
*  
* Usage:  
* cd-speed <device>  
*  
*****  
#include <iostream.h>  
#include <iomanip.h>  
/include
```

Figure 3-2: Search again, forward (`/<Enter>`).

Another way to find the next match is with the `n` command. This command does the same thing as `/<Enter>`, but does it with one less keystroke.

Both the `/<Enter>` and `n` commands can have a count specified. If there is a count, the command searches for the count number of matches from the current location.

Search History

The search command has a history feature. Suppose, for example, that you do three searches:

```
/one  
/two  
/three
```

The Vim Tutorial and Reference

Now let's start searching by typing a simple `/` without pressing `<Enter>`. If you press `<Up>`, *Vim* puts `/three` on the prompt line. Pressing `<Enter>` at this point searches for `three`.

If you do not press `<Enter>`, but press `<Up>` instead, *Vim* changes the prompt to `/two`. Another `<Up>` command moves you to `/one`. In other words, after you do a number of searches, you can use the `<Up>` and `<Down>` keys to select one of your recent searches.

History Window

If you execute the command `q/`, *Vim* will open up a search history window. See Figure 3-3.

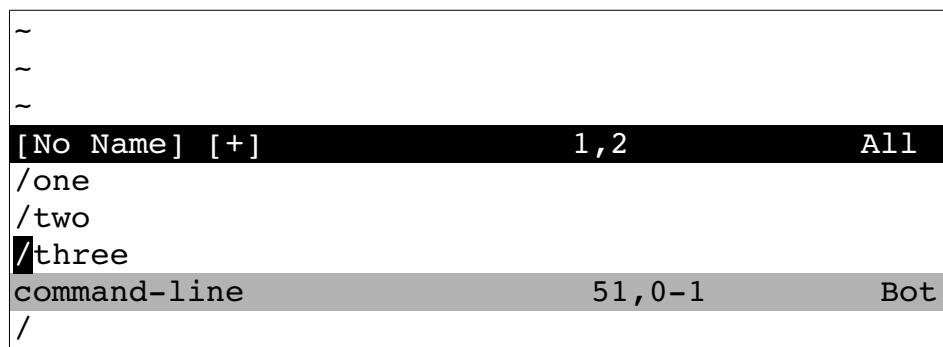


Figure 3-3: Search History Window

Most of the normal editing commands work within this window. When you find the search you want (or create it in the editor), press `<Enter>` to execute the search. If you wish to abort the search and return to command mode, use the `zz` or `:q` commands.

If you wish to abort everything `:qall` will close both the search history window and everything else.

Searching Options

Many different options control the way you perform a search. This section discusses a few of them.

Highlighting

The following command causes *Vim* to highlight any strings found matching the search pattern:

```
:set hlsearch
```

The Vim Tutorial and Reference

If you turn on this option and then search for `include`, for example, all the `include` strings are highlighted, as seen in Figure 3-4. To turn off search highlighting, use this command:

```
:set nohlsearch
```

To clear the current highlighting, use the following command:

```
:nohlsearch
```

Search highlighting is now turned off; matched text will not be highlighted. However, the highlighting will return when you use a search command.

```
*          devices)          *
*
* Usage:
*   cd-speed <device>
*
***** /*****
#include <iostream.h>
#include <iomanip.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
```

Figure 3-4 The '`hlsearch`' option.

Incremental Searches

By default, *Vim* uses the traditional search method: You specify the string, and then *Vim* performs the search. When you use the following command, the editor performs incremental searches:

```
:set incsearch
```

The editor starts searching as soon as you type the first character of the string. Each additional character further refines the search.

Suppose, for example, that you want to search for `ioctl.h`, but this time you want to use an incremental search. First, you turn on incremental searching. Next, you start the search by typing the `/i` command. Figure 3-5 shows how the editor searches for the first `i` and positions the cursor on it.

The Vim Tutorial and Reference

```
*          devices)          *
*
* Usage:
*   cd-speed <device>
*
***** */
#include <iostream.h>
#include <iomanip.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
```

Figure 3-5: Results after /i.

You continue the search by typing an **o**. Your search now is **/io**, so the editor finds the first io, as seen in Figure 3-6. This is still not the place you want, so you add a **c** to the search, resulting in the **/ioc** command. The *Vim* editor advances, as illustrated in Figure 3-7, to the first match of ioc. This is what you want to find, so you press **<Enter>**, and you're there.

```
*          devices)          *
*
* Usage:
*   cd-speed <device>
*
***** */
#include <iostream.h>
#include <iomanip.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
```

Figure 3-6: Incremental search after /io.

The Vim Tutorial and Reference

```
*           devices)
*
* Usage:
*     cd-speed <device>
*
*****/
```

```
#include <iostream.h>
#include <iomanip.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
```

Figure 3-7: Incremental search after /ioc.

To turn off incremental searches, use the following command:

```
:set noincsearch
```

Searching Backward

The reverse search command (?) searches backward. The n command repeats the last search. If a reverse search was the last one used, the n command searches in the reverse direction. If the last search was a forward search, the n command searches forward. Figure 3-8 shows how the ? and n commands can work together.

```
(wrap)  
Start _____  
?unsigned _____  
n _____  
n _____  
  
#include <sys/fcntl.h>  
#include <sys/time.h>  
#include <errno.h>  
// Read at most 10MB  
const unsigned int MAX_READ = (10 * 1024 * 1  
                                // Size of a buffer  
const unsigned int BUF_SIZE = (62 * 1024);  
// Buffer to be written  
state unsigned char buffer [BUF_SIZE];
```

Figure 3-8: ? and n commands.

Reverse Search History

Like forward search (?) you can use the **<Up>** and **<Down>** keys to go through the search history. You can also open a search history for reverse searches with the command **q?**.

Changing Direction

Suppose you start a forward search for unsigned using the `/unsigned` command. You can turn around and search in the reverse direction by using the `? Command`. The `n` command repeats the search in the same direction. The `N` command reverses the direction on the search and repeats it. To make things a little clearer, line numbering has been turned on using the following command:

```
:set number
```

In this example, we use the following search commands:

Command	Meaning	Result
<code>/unsigned</code>	Meaning Forward search for unsigned.	Line 24
<code>n</code>	Repeat search in the same (forward) direction.	Line 26
<code>n</code>	Search again.	Line 29
<code>?</code>	Reverse search	Line 26
<code>N</code>	Change search direction (reverse -> forward) and repeat search.	Line 29

Figure 3-9 shows the `/unsigned` command used to perform a search. The `n` command was used twice to go the next occurrences of the string. Then we reversed course with a `?` command (which always goes backward.) Finally, we reverse course again with the `N` command. Figure 3-9 shows this tortured path.

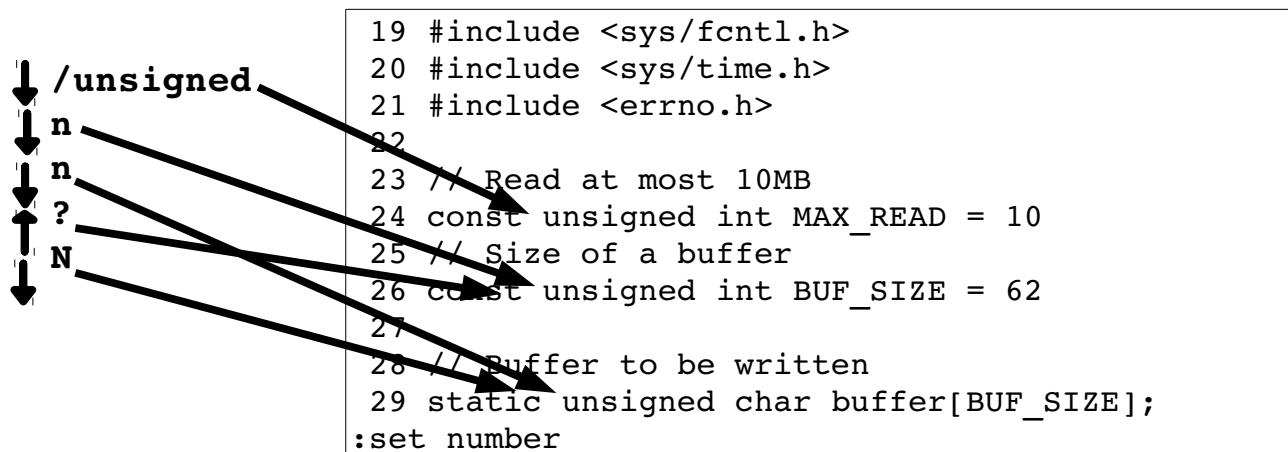


Figure 3-9: Different kinds of search commands.

Basic Regular Expressions

The *Vim* editor uses regular expressions to specify what to search for. Regular expressions are an extremely powerful and compact way to specify a search pattern. Unfortunately, this power comes at a price because regular expressions are a bit tricky to specify. Let's start with the simple stuff. In a regular expression, the normal letters match themselves. So the regular expression `Steve` will match `Steve`.

The Beginning (^) and End (\$) of a Line

The `^` character matches the beginning of a line. (It is no coincidence that this is also the command to move to the beginning of the line.) The expression `include` matches the word `include` anywhere on the line. But the expression `^include` matches the word `include` only if it is at the beginning of a line.

The `$` character matches the end of a line. Therefore, `was$` finds the word `was` only if it is at the end of a line. Figure 3-10, for example, shows a search for the pattern `the` with highlighting enabled.

```
<H1> Dumb user tricks
At one university the computer center was experience
trouble with a new type of computer terminal. Seems
that the professors loved to put papers on top of
the equipment, covering the ventilation holes. Many
terminals broke down because they became so hot that
the solder holding one of the chips melted and the
chip fell out.
The student technicians were used to this problem. One
day a technician took the back off a terminal
/the
```

Figure 3-10: Searching for `the`.

Next you see what happens when searching for the regular expression `^the`. The results, as seen in Figure 3-11, show that only two occurrences, both of which begin lines, are highlighted. Finally a search for `the$`. As you can see from Figure 3-12, only one `the` ends a line. If you want to search for a line consisting of just the word `the`, use the regular expression `^the$`. To search for empty lines, use the regular expression `^$`.

The Vim Tutorial and Reference

```
<H1> Dumb user tricks
At one university the computer center was experiencing
trouble with a new type of computer terminal. Seems
that the professors loved to put papers on top of
the equipment, covering the ventilation holes. Many
terminals broke down because they became so hot that
the solder holding one of the chips melted and the
chip fell out.
The student technicians were used to this problem. One
day a technician took the back off a terminal
/^the
```

Figure 3-11: Searching for ^the.

```
<H1> Dumb user tricks
At one university the computer center was experiencing
trouble with a new type of computer terminal. Seems
that the professors loved to put papers on top of
the equipment, covering the ventilation holes. Many
terminals broke down because they became so hot that
the solder holding one of the chips melted and the
chip fell out.
The student technicians were used to this problem. One
day a technician took the back off a terminal
/the$
```

Figure 3-12: Searching for the\$.

Match Any Single Character (.)

The character . matches any single character. For example, the expression c.m matches a string whose first character is a c, whose second character is anything, and whose third character is m. Figure 3-13 shows that the pattern matched the com of computer and the cam of became.

The Vim Tutorial and Reference

```
<H1> Dumb user tricks
At one university the computer center was experiencing
trouble with a new type of computer terminal. Seems
that the professors loved to put papers on top of
the equipment, covering the ventilation holes. Many
terminals broke down because they became so hot that
the solder holding one of the chips melted and the
chip fell out.
The student technicians were used to this problem. One
day a technician took the back off a terminal
/c.m
```

Figure 3-13: Special character ..

Matching Special Characters

Most symbols have a special meaning inside a regular expression. To match these special symbols, you need to precede them with a backslash (\). To find `the]`, for example, use the string `the\]`.

Regular Expression Summary

Note: The following list assumes that the '`magic`' option is on (the default). (See *The 'magic' Option* on page 304 for information on this option.)

Character	Meaning
<code>x</code>	The literal character <code>x</code>
<code>^</code>	Start of line
<code>\$</code>	End of line
<code>,</code>	A single character
<code>\x</code>	Turns off the special meaning of many characters, gives special meaning to a few others

Chapter 4: Text Blocks and Multiple Files

This chapter shows you how to deal with larger text blocks. This includes the commands that enable you to define a large text block as well as perform cut, paste, and copy operations.

With most editors, you can just cut and paste. However, the *Vim* editor has the concept of a register. This enables you to hold data for multiple cut, copy, or paste operations. Most other editors are limited to a single cut/paste clipboard. With the *Vim* registers you get more than 26 clipboards.

One of the strengths of UNIX is the number of text manipulation commands it provides. This chapter shows you how to use the filter command to take advantage of this power to use UNIX filters to edit text from within *Vim*.

Up until now, you have worked with single files in this book. You will now start using multiple files. This will enable you to perform the same edits on a series of files, and to cut and paste between files.

This chapter discusses the following topics:

- Simple cut-and-paste operations (in *Vim* terms, delete and put)
- Marking locations within the text
- Copying text into a register using the yank commands
- Filtering text Editing multiple files

Cut, Paste, and Copy

When you delete something with the **d**, **x**, or another command, the text is saved. You can paste it back by using the **p** command. (The *Vim* name for this is a put, but everyone else calls it “paste”).

Take a look at how this works. First you will delete an entire line by putting the cursor on the line you want to delete and pressing **dd**. Now you move the cursor to where you want to place the line and use the **p** (put) command. The line is inserted on the line following the cursor. Figure 4-1 shows the operation of these commands.

The Vim Tutorial and Reference

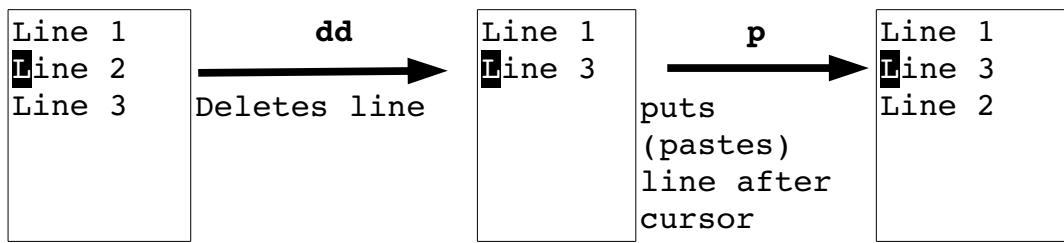


Figure 4-1: Deleting (cutting) and putting (pasting).

Because you deleted an entire line, the **p** command placed the text on the line after the cursor. If you delete part of a line (a word with the **dw** command, for instance), the **p** command puts it just after the character under the cursor (see Figure 4-2).

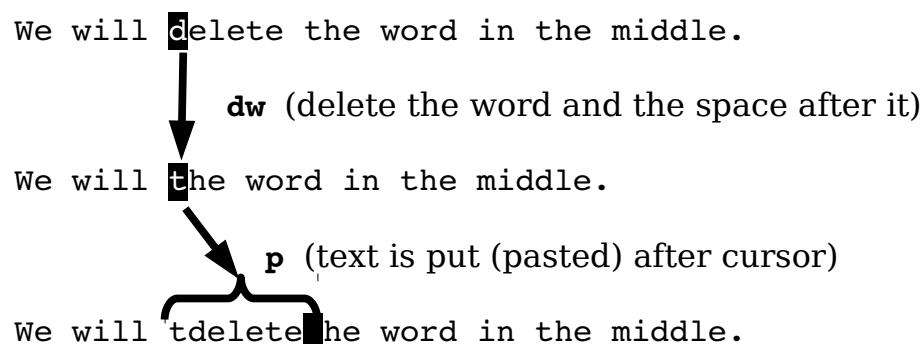


Figure 4-2: Deleting a word and putting back again.

Character Twiddling

Frequently when you are typing, your fingers get ahead of your brain. The result is a typo such as *teh* for *the*. The *Vim* editor makes it easy to correct such problems. Just put the cursor on the *e* of *teh* and execute the command **xp**. Figure 4-3 illustrates this command. This works as follows:

- x** Deletes the character *e* and places it in a register.
- p** Puts the text after the cursor, which is on the *h*.

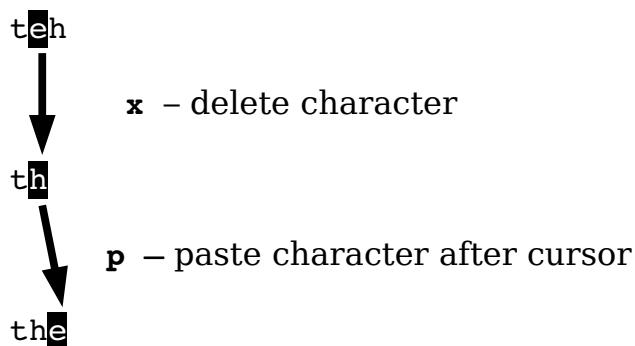


Figure 4-3: Character twiddling with **xp**.

More on "Putting"

You can execute the **p** command multiple times. Each time, it inserts another copy of the text into the file. The **p** command places the text after the cursor. The **P** command places the text before the cursor. A **{count}** can be used with both commands and, if specified, the text will be inserted **{count}** times.

Moving Large Blocks of Text

Let's say you wish to move 57 line block of text from one place to another. One solution is to perform the following commands:

1. Put the cursor on the first line of the block
2. Type **57dd** to delete the block.
3. Move the line just before where you want to insert the block.
4. Execute **p** to "put" the block on the line after the one the cursor is on.

This works great if you know the exact number of lines you wish to move. However for me, any text block larger than three lines confuses me. This method of moving text is impractical for such large text blocks.

Fortunately we have visual mode. We can start *line* visual mode with the **v** (upper case) command. This is like the simple visual mode (**v**) we've used before, only it works only on entire lines.

So to move a block using line visual mode, we execute the following commands:

The Vim Tutorial and Reference

1. Put the cursor on the first line of the block.
2. Start line visual mode with the **v** command.
3. Put the cursor on the last line of the block.
4. Delete the text with the **d** command (visual delete).
5. Move the line just before where you want to insert the block.
6. Execute **p** to “put” the block on the line after the one the cursor is on.

Figure 4-4 shows how this works. The major advantage of moving text using this method is that you don't have to count and you can tell exactly what text is going to be moved before you move it.

4

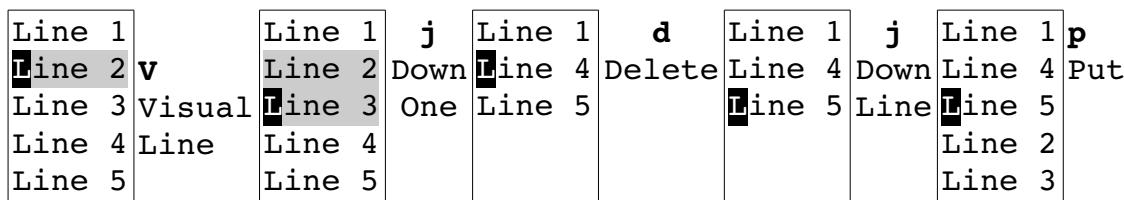


Figure 4-4: Moving a block of text visually.

Marks

The *Vim* editor enables you to place marks in your text. The command **ma** marks the place under the cursor as mark **a**. You can place 26 marks (**a** through **z**) in your text. (You can use a number of other special marks as well.)

To go to a mark, use the command ``{mark}`, where `{mark}` is the mark letter (and ``` is the backtick or open single-quote character).

The command `'{mark}` (single quotation mark, or apostrophe) moves you to the beginning of the line containing the mark. This differs from the ``{mark}` command, which moves you to the marked line and column.

The ``{mark}` command can be very useful when deleting a long series of lines. To delete a long series of lines, follow these steps:

1. Move the cursor to the beginning of the text you want to delete.
2. Mark it using the command **ma**. (This marks it with mark **a**.)

The Vim Tutorial and Reference

3. Go to the end of the text to be removed. Delete to mark a using the command **d'a**. (Entire lines will be deleted since '**a**' is a line type move.)

Note: There is nothing special about using the **a** mark. Any mark from **a** to **z** may be used.

There is nothing special about doing the beginning first followed by the end. You could just as easily have marked the end, moved the cursor to the beginning, and deleted to the mark.

One nice thing about marks is that they stay with the text even if the text moves (because you inserted or deleted text above the mark. Of course, if you delete the text containing the mark, the mark disappears.

Where Are the Marks?

To list all the marks, use the following command:

:marks

Figure 4-5 shows the typical results of such a command.

Figure 4-5: :marks.

The display shows the location of the marks **a** through **d** as well as the special marks: ` , " , [, and] . Marks **a** through **d** are located at lines 1, 8, 14, and 25 in the file.

The Vim Tutorial and Reference

The special marks are as follows:

Mark	Description	Location in this example
'	The last place the cursor was at.	Line 67 of the current file
"	The position of the cursor when we last left this buffer. Since we have only been editing this file, it defaults to the first character.	Line 1 (the default)
[The start of the last insert	Line 128
]	The end of the insert	Line 129

To view specific marks, use this command:

```
:marks args
```

Replace *args* with the characters representing the marks you want to view.

Yanking

For years, I used a simple method for copying a block of text from one place to another. I deleted it using the **d** command, restored the deleted text with the **p** command, and then went to where I wanted the copy and used the **p** to put it into the text.

There is a better way. The **y** command "yanks" text into a register (without removing it from the file). (Every other editor calls this a "copy".)

To yank visually you start visual mode with **v**, highlight the text you wish to yank and yank it with **y**. Figure 4-6 shows how this works.

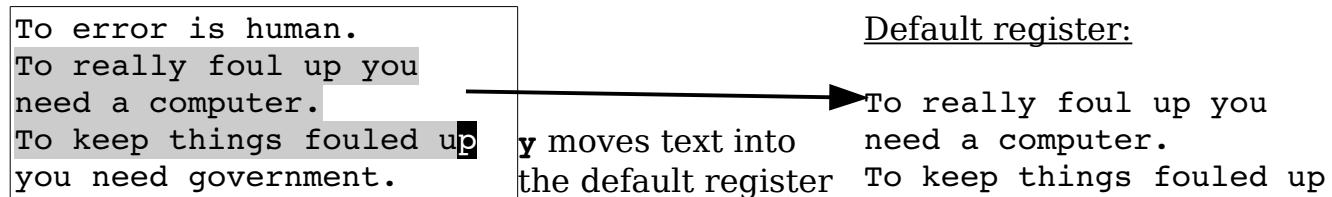


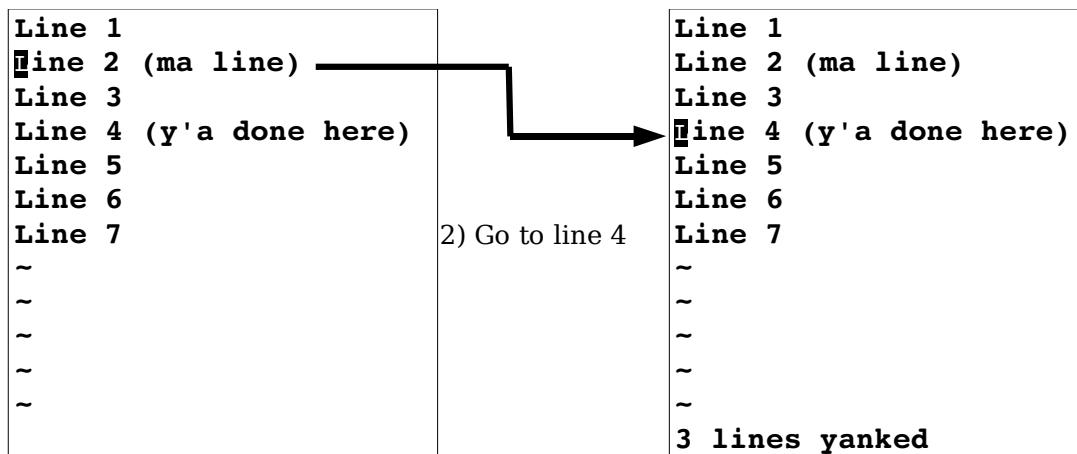
Figure 4-6: Yanking text.

Normal Mode Yanking

The general form of the normal mode **y** command is **y{motion}**. It works just like the delete (**d**) command except the text is not deleted. And the shorthand **yy** yanks the current line into the buffer.

The Vim Tutorial and Reference

Take a look at how you can use the yank (**y**) command along with the mark command (**m**) to duplicate a block of text. First go to the top of the text to be copied and mark it with **ma**. Then go to the bottom and do a **y'a** (yank to mark **a**). Now go to where the copied text is to be inserted and put it there using the **p** command. Figure 4-7 shows these commands in action.



1) Place mark a (**ma**)

```
Line 1
Line 2 (ma line)
Line 3
Line 4 (y'a done here)
Line 5
Line 6
Line 7
~  
~  
~  
~  
~
```

4) Move to line 6

3) Yank to mark a (**y'a**)

```
Line 1
Line 2 (ma line)
Line 3
Line 4 (y'a done here)
Line 5
Line 6
Line 7
~  
~  
~  
~  
~
```

The text "3 more lines" is displayed at the bottom of the right-hand buffer.

5) Put text in using
the **p** command

Figure 4-7: Yank (copy) and put (paste).

Yanking Lines

The **y** command yanks a single line. If preceded by a count, it yanks that number of lines into the register. You might have expected **y** to yank until the end of the line, like **d** and **c**, but it really yanks the whole line.

Filtering

The visual ! command takes a block of text and filters it through another program. In other words, it runs the system command represented by command, giving it the block of text represented by motion as input. The output of this command then replaces the selected block.

Because this summarizes badly if you are unfamiliar with UNIX filters, take a look at an example. The *sort* command sorts a file. If you execute the following command, the unsorted file *input.txt* will be sorted and written to *output.txt*. (This works on both UNIX and Microsoft Windows.)

```
$ sort <input.txt >output.txt
```

Now do the same thing in *Vim*. You want to sort lines 1 through 10 of a file. First start visual mode (**v**) and highlight the first 10 lines. Then press !.

In anticipation of the filtering, the cursor drops to the bottom of the screen and a ! prompt displays. You can now type in the name of the filter program, in this case *sort*. Therefore, your full command is as follows:

```
!sort<Enter>
```

The result is that the *sort* program is run on the first 10 lines. The output of the program replaces these lines.

Normal Mode Filtering

Like normal mode delete (**d**) and normal mode yank (**y**), the filter command has a normal mode version as well. The command !{**motion**} processes the block of text starting at the current line and going to whatever line **{motion}** takes you through a filter.

The !! command runs the current line through a filter. (I have found this a good way to get the output of system commands into a file.) I'm editing a *readme.txt* file, for example, and want to include in it a list of the files in the current directory. I position the cursor on a blank line and type the following:

```
!!ls
```

The Vim Tutorial and Reference

This puts the output of the *ls* command into my file. (Microsoft Windows users would use *dir*.) Another trick is to time stamp a change. To get the current date time (on UNIX), I use the following command:

```
! !date
```

This proves extremely useful for change histories and such.

Note: Using **!!** like this is technically not filtering because commands like *ls* and *date* don't read standard input.

Editing Another File

Suppose that you have finished editing one file and want to edit another file. The simple way to switch to the other file is to exit *Vim* and start it up again on the other file. Another way to do so is to execute the following command:

```
:e file
```

Note: **:e** is an abbreviation for **:edit**.

This command automatically closes the current file and opens the new one. If the current file has unsaved changes, however, *Vim* displays a warning message and aborts the command:

```
No write since last change (use ! to override)
```

At this point, you have a number of options. You can write the file using this command:

```
:write
```

Note: **:w** may be used instead of **:write**.

Or you can force *Vim* to discard your changes and edit the new file using the force (!) option, as follows:

```
:e! file
```

To edit a new,unnamed buffer use the **:enew** (**:ene**) command. It works just like **:edit** only the new buffer is unnamed.

The :view Command

The following command works just like the **:vi** command, except the new file is opened in read-only mode:

```
:view file
```

Note: **:vie** may be used instead of **:view**.

If you attempt to change a read-only file, you receive a warning. You can still make the changes; you just can't save them. When you attempt to save a changed read-only file, *Vim* issues an error message and refuses to save the file. (You can force the write with the **:write!** command, as described later in this chapter.)

Dealing with Multiple Files

So far the examples in this book have dealt with commands that edit a single file. This section introduces you to some commands that can edit multiple files. Consider the initial *Vim* command, for example. You can specify multiple files on the command line, as follows:

```
$ gvim one.c two.c three.c
```

This command starts *Vim* and tells it that you will be editing three files. By default, *Vim* displays just the first file (see Figure 4-8).

```
/* File one.c */
~
~
~
~
```

Figure 4-8: Editing the first of multiple files.

To edit the next file, you need to change files using the **:next** command (**:n**). Figure 4-10 shows the results.

```
/* File two.c */
~
~
~
~
"two.c" 1L, 17C
```

Figure 4-9: **:next**.

Note that if you have unsaved changes in the current file and you try to do a **:next**, you will get a warning message and the **:next** will not work. You can solve this problem in many different ways. The first is to save the file using the following command:

```
:write
```

The Vim Tutorial and Reference

In other words, you can perform a `:write` followed by a `:next`. The *Vim* editor has a shorthand command for this. The following command performs both operations:

```
:wnext
```

Note: `:wn` is the same as `:wnext`.

Or, you can force *Vim* to go the next file using the force (!) option. If you use the following command and your current file has changes, you will lose those changes:

```
:next!
```

Finally, there is the '`autowrite`' ('aw') option. If this option is set, *Vim* will not issue any `No write...`. Instead, it just writes the file for you and goes on. To turn this option on, use the following command:

```
:set autowrite
```

To turn it off, use this command:

```
:set noautowrite
```

- 5 You can continue to go through the file list using the following command until
- 6 you reach the last file:

```
:next
```

Also, the `:next` command can take a repeat count. For example, if you execute the command

```
:2 next
```

(or `:2next`), *Vim* acts like you issued a `:next` twice.

Note: Some commands like `:quit` will fail if any open file has been modified. The '`autowrite`' option only works for the current file. If you want something that works on all files you need the '`autowriteall`' ('awa') option.

Which File Am I On?

Suppose you are editing a number of files and want to see which one you are on. The following command displays the list of the files currently being edited:

```
:args
```

The one that you are working on now is enclosed in square brackets. Figure 4-10 shows the output of the command.

The Vim Tutorial and Reference

```
/* File two.c */
-
-
-
-
one.c [two.c] three.c
```

Figure 4-10: Output of :args

This figure shows three files being edited: *one.c*, *two.c*, and *three.c*. The file currently being editing is *two.c*.

Going Back a File

To go back a file, you can execute any of the following commands (all are equivalent):

```
:previous
:prev
:Next
:N
```

These commands act just like the :next command, except that they go backward rather than forward. If you want to write the current file and go to the previous one, use any of the following commands:

```
:wprevious
:wp
:wNext
:wN
```

Editing the First or Last File

To start editing from the first file, no matter which file you are on, execute the following command:

```
:first
```

(You can also use the commands :rewind, :rew, and :fir to do the same thing.)

To edit the last file, use this command:

```
:last
```

Note: :la is the same as :last.

Editing Two Files

Suppose that you edit two files by starting *Vim* with the following:

```
$ gvim one.c two.c
```

You edit a little on the first file, and then go to the next file with the following:

```
:wnext
```

At this point, the previous file, *one.c*, is considered the *alternate file*. This has special significance in *Vim*. For example, a special text register (#) contains the name of this file.

By pressing **CTRL-^**, you can switch editing from the current file to the alternate file. Therefore, if you are editing *two.c* and press **CTRL-^**, you will switch to *one.c* (*two.c* becoming the alternate file). Pressing **CTRL-^** one more time switches you back.

Suppose that you are editing a bunch of files, as follows:

```
$ gvim one.c two.c three.c
```

The command [*count*]**CTRL-^** goes to the count file on the command line. The following list shows the results of several **TRL-^** commands:

1 CTRL-^	<i>one.c</i>
2 CTRL-^	<i>two.c</i>
3 CTRL-^	<i>three.c</i>
CTRL-^	<i>two.c</i> (previous file)

Note When you first start editing (file *one.c*) and press **CTRL-^**, you will get an error message: No alternate file. Remember the alternate file is the last file you just edited before this one (in this editing session). Because *one.c* is the only file edited, there is no previous file and therefore the error message.

Matching

Though not exactly a search command, the **:match** (**:maxx**) command can be very useful in finding test. It causes all text that matches a given pattern to be highlighted on the screen. For example, to highlight all the word "TODO" with the Error syntax highlighting, use the command:

```
:match Error /TODO/
```

The Vim Tutorial and Reference

To find out what highlighting names are available use the **:highlight (:hi)** command:

```
:highlight
```

To clear the match from the screen, use the command:

```
:match none
```

There can be three matches active at one time. These are set by the **:match**, **:2match**, and **:3match** commands. If some text is matched by more than one command, the lowest one wins.

Chapter 5: Windows and Tabs

So far you have been using a single window. In this chapter, you split the screen into multiple windows and edit more than one file simultaneously. This chapter also discusses the use of editing buffers. A buffer is a copy of a file that you edit along with the setting and marks that go with it. The topics discussed in this chapter include the following:

- How to open a new window
- Window selection
- Editing two files at once
- Controlling the size of a window
- Basic buffer usage
- Basic Tabs

Opening a New Window

The easiest way to open a new window is to use the following command:

:split

(**:sp** may be used instead of **:split**)

This command splits the screen into two windows (and leaves the cursor in the top one), as seen in Figure 5-1.

```
/* File one.c */
~
~
one.c
/* File one.c */
~
one.c
```

Figure 5-1: Splitting a window.

Both are editing the same file, so you can view two different parts of a file simultaneously.

The Vim Tutorial and Reference

If you are at the bottom window, the **CTRL-Ww** command moves the cursor to the top window (alternate command: **CTRL-W CTRL-W**). If you are at the top window, the editor jumps to the bottom one on the screen.

To change windows, use **CTRL-Wj** (**CTRL-W CTRL-J**, **CTRL-W<Down>**) to go down a window and **CTRL-Wk** (**CTRL-W CTRL-K**, **CTRL-W<Up>**) to go up a window (see Figure 5-2). Or if you are using *Vim* with a mouse, you can just click in the window you want. (**CTRL-W CTRL-J** is an alternate for **CTRL-Wj** and **CTRL-W CTRL-K** is an alternate for **CTRL-Wk**)

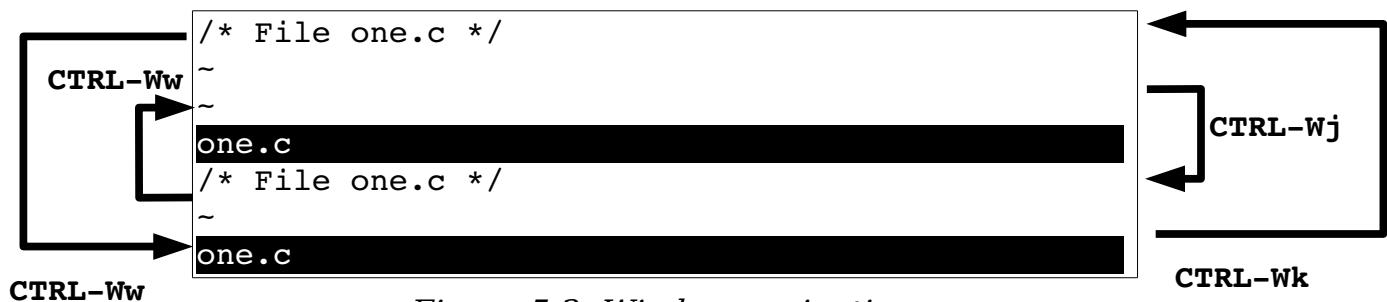


Figure 5-2: Window navigation.

To close a window, use **zz** or the following command:

```
:q
```

CTRL-Wc, **:close**, **:clo** and **:quit** do the same thing.

Usually you would expect **CTRL-W CTRL-C** also to close a window. It would if all the **CTRL-W** commands were consistent. Because **CTRL-C** cancels any pending operation, however, **CTRL-W CTRL-C** does nothing.

Vertical Windows

The **:split** (**:sp**) command divides the screen horizontally. To divide the screen vertically use the **:vsplit** (**:vs**) command. Figure 5-3 shows the result of this operation:

The Vim Tutorial and Reference

```
/* one.c */  
~  
~  
~  
~  
~  
~  
~  
one.c  
/* one.c */  
~  
~  
~  
~  
~  
~  
~  
one.c
```

Figure 5-3: Result of the `:vsplit` command.

It should be noted anything that the `:split` command can do, the `:vsplit` command can do vertically.

To move from the left window to the right use the **CTRL-W l** (**CTRL-W CTRL-L**, **CTRL-W<Right>**) command. To go from the right to left use the **CTRL-W h** (**CTRL-W CTRL-H**, **CTRL-W<Left>**) command. The **CTRL-W w** (**CTRL-W CTRL-W**) command moves you to the next window to the right, wrapping to the leftmost window if there is no window to the right. Figure 5-4 shows how these commands work:

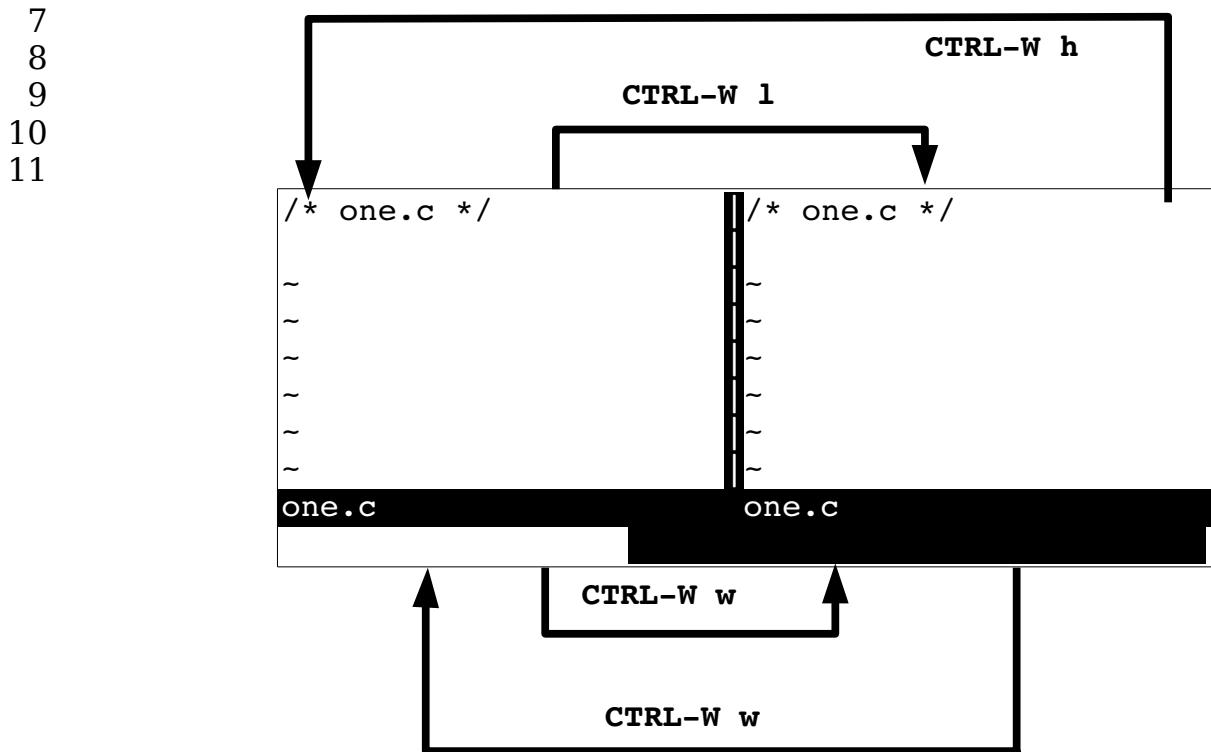


Figure 5-4: Result of the `:vsplit` command.

Opening Another Window with Another File

The following command opens a second window and starts editing the given file:

```
:split file
```

Figure 5-5 shows what happens when you start editing *one.c* and then execute the following command

```
:split two.c
```

```
/* File two.c */
~
~
two.c
/* File one.c */
~
one.c
```

Figure 5-5: Results of :split two.c.

The :split (:sp) command can also execute an initial command using the +**command** convention. Figure 5-6 shows what happens when you are editing *one.c* and execute the following command:

```
:split +/printf three.c
```

```
{
    printf("%2d squared is %3d\n", i, i*i);
}
three.c
/* File one.c */
~
one.c
```

Figure 5-6: Result of :split with a + command

Quick Split

The command **CTRL-Wn** (**CTRL-W** **CTRL-N**, :new) splits a window and starts editing a new file.

Controlling Window Size

The **:split (:sp)** command can take a number argument. If specified, this will be the number of lines in the new window. For example, the following opens a new window three lines high and starts editing the file *alpha.c*:

:3 split alpha.c

A space appears here for clarity. You could have just as easily write the following:

:3split alpha.c

Figure 5-7 shows the results.

Figure 5-7: :3split.

Split Summary

The general form of the `:split (:sp)` command is as follows:

:count split +command file

count The size of the new window in lines. (Default is to split the current window into two equal sizes.)

+command An initial command.

file The name of the file to edit. (Default is the current file.)

The :new Command

The **:new** command works just like the **:split** command except that the **:split** command splits the current window and displays the current file in both windows while **:new** opens a window containing a new, empty file.

The following command splits the current window and starts a new file in the other window:

```
:new
```

The **:vnew** (**:vne**) does the same thing vertically.

Split and View

The **:sview** (**:sv**) command acts like a combination of **:split** and **:view**. This command proves useful if you want to look at, but not edit, a file in another window.

There is no **:vsview** command. However if you wish to split the window vertically and edit a file you can use the **:vertical :sview** (**:vert :sview**) command. The **:vertical** command tells *Vim* to perform the command that follows splitting the window vertically instead of horizontally.

Changing Window Size

Changing window size when you are using *gvim* is easy. To change the size of a window, use the mouse to drag the separator up or down (see Figure 5-8). If you are using the terminal version of *Vim*, you need to type in some commands.

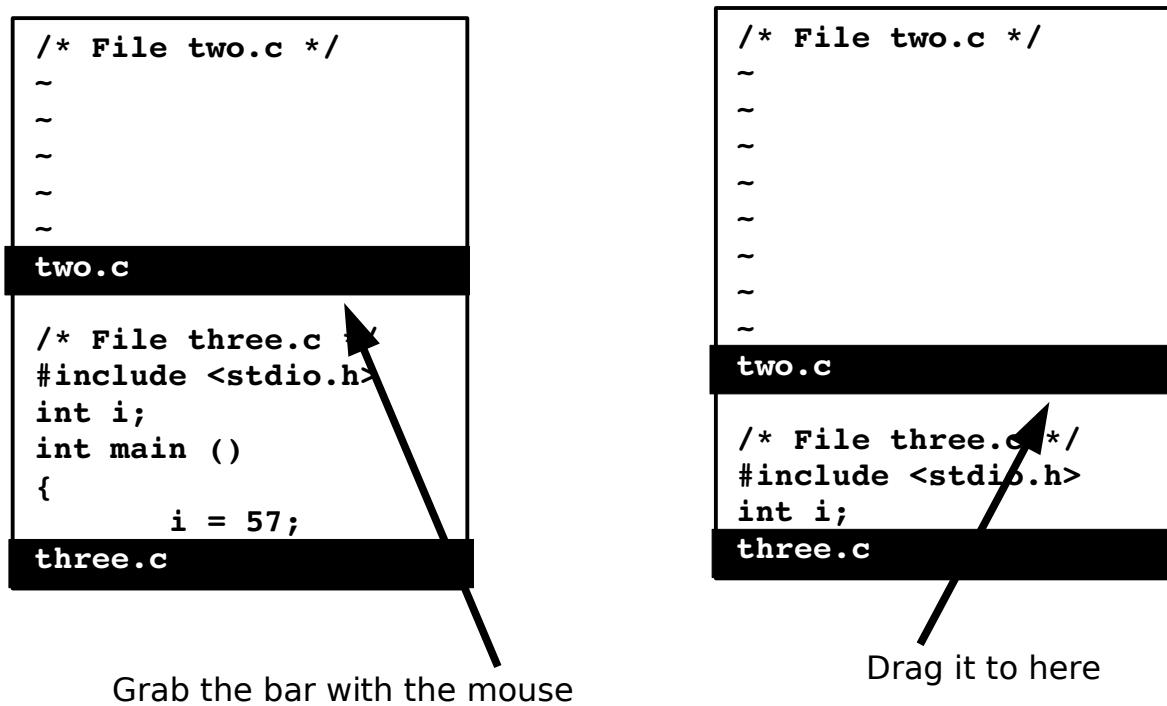


Figure 5-8: Adjusting the window size.

The command **count CTRL-W+** increases the window size by count (default = 1). Similarly **count CTRL-W-** decreases the window's size by count (default = 1). The command **CTRL-W=** makes all the windows the same size (or as close as possible).

The command **count CTRL-W_ ($\text{CTRL-W } \text{CTRL-}_$)** makes the current window count lines high. If no count is specified, the window is increased to its maximum size.

The **:resize (:res)** command can be used to change the height of the window from the command line. The argument to this command can be a simple number (i.e. **24**) in which case the window is set to that size. If the argument begins with a plus or minus, (i.e. **+5, -7**), then the window is increased or decreased in size.

If no argument is given to the **:resize** command, the window is enlarged to its maximum height.

Buffers

The *Vim* editor uses the term buffer to describe a file being edited. Actually, a buffer is a copy of the file that you edit. When you finish changing the buffer and exit, the contents of the buffer are written to the file. Buffers not only contain file contents, but also all the marks, settings, and other stuff that go with it.

Normally it is pretty easy to tell what buffers you have: If it has a window on the screen, it is a buffer; if it is not on the screen, it is not a buffer.

Now for a new concept thrown into the mix, that of the hidden buffer. Suppose you are editing a file named *one.c* and you need to do some work on *two.c*. You split the screen and create two windows, one for each file. But you do not like split-screen mode; you want to see one file at a time.

One solution is to make the window for *two.c* as big as possible. This works, but there still is an annoying little bit of *one.c* showing. Another solution is to close the *one.c* window, but then you lose all the changes for that file.

The *Vim* editor has another solution for you: the **:hide** (**:hid**) command. This causes the current buffer to become "hidden." This causes it to disappear from the screen. But *Vim* still knows that you are editing this buffer, so it keeps all the settings, marks, and other stuff around. Actually, a buffer can have three states:

Active	Appears onscreen.
Hidden	A file is being edited, but does not appear onscreen.
Inactive	The file is not being edited, but keep the information about it anyway.

The inactive state takes a little explaining. When you edit another file, the content of the current file is no longer needed, so *Vim* discards it. But information about marks in the file and some other things are still useful and are remembered along with the name of the file. Also, a file that was included in the command with which you started *Vim*, but was not edited, will also be an inactive buffer.

To find a list of buffers, use the following command:

:buffers

(**:buffers** can also be written as **:ls**, **:files**.)

The Vim Tutorial and Reference

Figure 5-9 shows the results of this command. The first column is the buffer number. The second is a series of flags indicating the state of the buffer. The third is the name of the file associated with the buffer. The state flags are:

- Inactive buffer.
- h Buffer is hidden.
- % Current buffer.
- # Alternate buffer.
- + File has been modified.

```
~  
~  
~  
~  
~  
~  
two.c  
:buffers  
 1 #h  "one.c"          line 1  
 2%   "two.c"           line 1  
 3 -   "three.c"         line 1  
 4 -   "four.c"          line 0  
 5 -   "help.txt"        line 1  
 6 -   "editing.txt"     line 234  
Press RETURN or enter command to continue █
```

Figure 5-9: **:buffers**.

In this case, you see six buffers:

1. The next to last file you were working on, also known as the alternate file (# flag). This buffer has been hidden (h flag). You were editing file *one.c* and left the cursor on line 1.
2. The active buffer (% flag). This is the file you are editing.
3. An inactive buffer. You want to edit *three.c*, but you have made no changes to it.
4. Another file on the argument list that has not been edited.
5. When you executed a **:help** command, the *Vim* editor opened two files. The first one of these is called *help.txt*.

6. This is another help file called *editing.txt*.

Selecting a Buffer

You can select which buffer to use by executing the following command:

:buffer number

(**:b** is the abbreviation for **:buffer**.)

The **number** parameter is the buffer number. If you do not know the number of the buffer, but you do know the filename, you can use this command:

:buffer file

Figure 5-10 shows the results of a typical **:buffer** command.

```
/* File three.c */
#include <stdio.h>
int i;
int main()
{
    for (i = 1; i <= 10; ++i)
    {
        printf("%2d squared is %3d\n", i,
i*i);
    }
    return (0);
}
~
~
three.c
```

Figure 5-10: **:3buffer** or **:buffer three.c**.

The following command splits the window and starts editing the buffer:

:sbuffer number

(**:sb** is the abbreviation for **:sbuffer**.)

If a number is specified, the new window will contain that buffer number. If no number is present, the current buffer is used.

To split the window vertically use the command:

:vertical sbuffer number

The Vim Tutorial and Reference

This also takes a filename as an argument. If you are editing the file *three.c* and execute the following command

```
:sbuffer one.c
```

You get the results seen in Figure 5-11.

```
/* File one.c */
~
~
~
~
~
one.c
/* File three.c */
#include <stdio.h>
int i;
int main()
{
    for (i = 1; i <= 10; ++i)
"three.c" line 1 of 1 100% col 1 ((2) of 3)
```

Figure 5-11: Result of :sbuffer.

93 Other buffer-related commands include the following:

:bnext	Go to the next buffer. (Also known as :bn.)
:count bnnext	Go to the next buffer count times.
:count sbnext	Shorthand for :split followed by :count bnnext. (Also known as :sbn.)
:count bprevious	Go to previous buffer. If a count is specified, go to the count previous buffer. (Also known as :bp, :bNext, :bn.)
:count sbprevious	Shorthand for :split and :bprevious. (Also known as :sbp, :sbNext, :sbn.)
:blast	Go to the last buffer in the list. (Also known as :bl.)
:sblast	Shorthand for :split and :blast. (Same as :sbl.)
:brewind	Go to the first buffer in the list. (Same as :bf, :bfFirst, :br.)
:sbrewind	Shorthand for :split and :brewind. (Same as :sbF, :sbFirst, :sbr.)
:bmodified count.	Go to count modified buffer on the list. (Same as

	:bm.)
:sbmodified count	Shorthand for :split and :bmodified. (Same as :sbm.)

Buffer Types

There are many different special buffers you'll encounter in the *Vim* editor. To keep track of them *Vim* uses the '**buftype**' ('**bt**') option. It can contain the following values:

<empty>	Normal buffer
acwrite	A buffer which will always be written with an :autocmd.
help	Help window
nofile	A buffer that is not associated with a file and will not be written.
nowrite	This buffer will not be written.
quickfix	A quickfix list.

Buffer Options

Usually when the last window of a file is closed, the buffer associated with the file becomes inactive. If the option '**hidden**' ('**hid**') is set, files that leave the screen do not become inactive; instead they automatically become hidden. Therefore if you want to keep the contents of all your old buffers around while editing, use the following command

```
:set hidden
```

Note: The :hide command always hides the current file no matter what the '**hidden**' option is set to.

The '**buflisted**' ('**bl**') command tell Vim if a buffer is to be listed in the :buffers or :ls command.

The '**bufhidden**' ('**bh**') can be used to fine tune when a buffer appears and hides. (Do not use this option unless you know what you are doing.) The values for this option are:

<empty>	The value of the ' hidden ' option is used to decide whether or not to hide the buffer.
delete	Delete the buffer as if a :bdelete were performed on it. (Use caution)

The Vim Tutorial and Reference

	unsaved changes can easily be discarded.)
hide	Hide, but do not unload the buffer. (As if ' hidden ' where set.)
unload	Unload the buffer. (Warning: Unsaved changes can easily be discarded.)
wipe	Wipeout the buffer as if the :bwipeout command were executed. (Again, unsaved changes can easily be discarded.)

Normally, the split/buffer related commands split the current window. If the '**switchbuf**' ('**swb**') option is set to **useopen** and there is a window displaying the buffer you want to display already on the screen, the *Vim* will just make that window the current one instead of performing the split (see Figure 5-12).

```
$ gvim t.c Makefile
```

```
:snext
```

```

int main()
{
    i = 5;
    printf("Hello World/n");
    return (0);
t.c
t: t.c
~
~
~
```

```

:set switchbuf=""
:sbrewind
{
    i = 5;
    printf("Hello World/n");
t.c
t: t.c
~
~
```

```

Makefile
```



```

:set switchbuf=useopen
:sbrewind
```

```

#include <stdio.h>
int main()
{
    i = 5;
    printf("Hello World/n");
t.c
t: t.c
~
~
```

```

gcc t.c
Makefile
```

Figure 5-12: The '**switchbuf**' option.

The Vim Tutorial and Reference

Note the '**switchbuf**' option is a list of the values: (nothing), **split**, **useopen** and **usetab**. For a description of the **split** argument see *Chapter 23: Advanced Commands for Programmers*.

Basic Tabbed Editing

The **:tabnew** (**:tabe**, **:tabedit**) command opens up a new tab. Figure 5-13 shows the result:

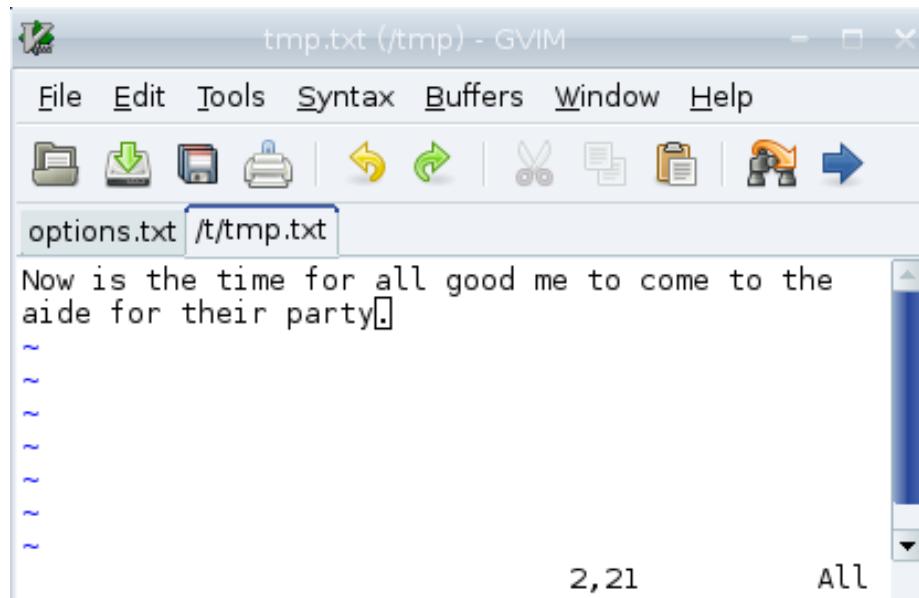


Figure 5-13: The **:tabnew /tmp/tmp.txt** command.

You can click on each tab to make that tab the current tab. If you wish to edit a specific file use the command:

```
:tabedit {file-name}
```

In general the **:tab** command will cause any command that would open a new window to open a new tab instead. For example:

```
:tab :split test.txt
```

Tabs may be closed by the **:quit** (**:q**) command. Or you can write the file and close the tab with **zz**. The **:tabclose** (**:tabc**) command acts just like **:quit**. However, if you give **:tabclose** a numeric argument, that tab is close. For example to close the second tab, use the command:

```
:tabclose 2
```

The Vim Tutorial and Reference

The `:tabonly (:tabo)` command closes all the *other* tabs. This will fail if closing a tab would result in data loss, unless the override (!) is specified.

Selecting a tab

To make a particular tab the current one, click on it. Or you can use the `vim` command `:tabnext (:tabn)` to go to the next tab. This command wraps so if you're on the last tab, you'll wind up on the first one.

If you specify a `{count}` as an argument to `:tabnext`, *Vim* will go to that tab. For example, to go to the third tab:

```
:tabnext 3
```

The `gt (<C-PageDown>)` command goes to the next tab. When a `[count]` is specified such as `3gt`, *Vim* goes to the specified tab.

The `:tabprevious (:tabp, :tabNext, :tabN)` command works like `:tabnext` only in the other direction. Similarly `gT (<C-PageUp>)` is like `gt` on to the left.

Finally `:tabfirst (:tabf, :tabrewind, :tabr)` goes the first tab and `:tablast (:tabl)` goes to the last one.

Figure 5-14 shows the tab commands in action.

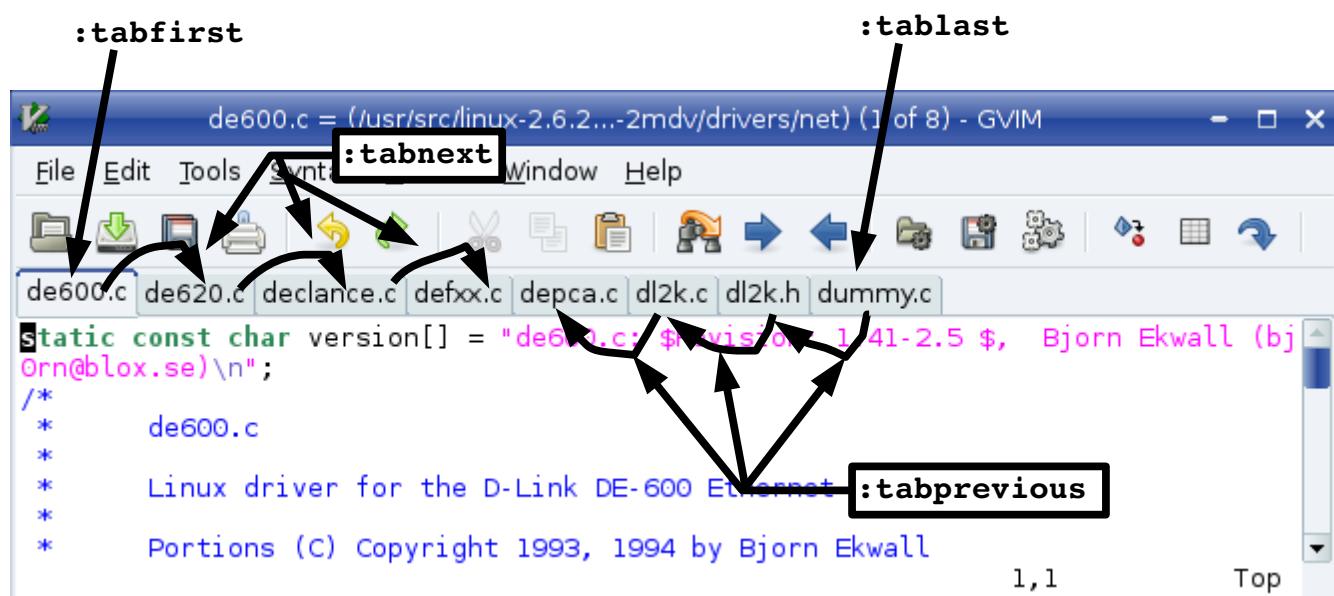


Figure 5-14: Tab navigation

Finding Files with Tabs

The **CTRL-Wgf** command opens a new tab and starts to edit the file whose name is under the cursor. *Vim* will search for the file using the same algorithm as it uses for the **:find** command. The **CTRL-WgF** command does the same thing, only it goes a step farther. It not only starts editing the file in a new tab, but jumps to the line number following the file name.

The **:tabfind** (**:tabf**) command performs a similar function. It's actually a shorthand for **:tabnew**, **:find**.

Editing Multiple Files From the Command Line

There are lots of different ways of editing multiple files. One way is to edit them one at a time as discussed in *Chapter 4: Text Blocks and Multiple Files*. Another is to edit them all at the same time.

To edit each file in its own window use the command:

```
$ gvim -o file1 file2 file3 ...
```

You can limit the number of windows opened by *Vim* by giving **-o** a numeric argument.

```
$ gvim -o3 file1 file2 file3 file4
```

Note: If you have more windows than files, some windows will be empty.

The **-p** command does the same thing except that it opens each file in its own tab. Like **-o** you can give it a numeric argument. Unlike **-o** there is an option ('**tabpagemax**' aka '**tpm**') which limits the number of tabs opened this way.

Chapter 6: Basic Visual Mode

One feature that sets *Vim* apart from its predecessor is something called visual mode. This mode gives you the ability to highlight a block of text and then execute a command on it. You can highlight a block of text, for example, and then delete it with a **d** command. The nice thing about visual mode is that, unlike other *Vim* commands where you operate blindly, with visual mode you can see what text is going to be affected before you make a change. In this chapter, you learn about the following:

- How to start visual mode
- Visual yanking
- Using visual mode to change text
- Visual commands for programmers
- Visual block mode

Entering Visual Mode

To enter visual mode, type the **v** command. Now when you move the cursor, the text from the start position to the current cursor location is highlighted (see Figure 6-1). After the text has been highlighted, you can do something with it. For example, the **d** command deletes it. Figure 6-2 shows the results.

A screenshot of a terminal window displaying a C program named 'three.c'. The code is as follows:

```
#include <stdio.h>
int i;
int main()
{
    for (i = 1; i <= 10; ++i)
        printf("%2d squared is %3d\n", i, i*i);
    return (0);
}
```

The word 'three.c' and the string '-- VISUAL --' are at the bottom of the screen. Two arrows point to specific parts of the code:

- An arrow labeled 'v - start visual mode' points to the opening brace '{' of the for loop.
- An arrow labeled 'Move cursor to here' points to the closing brace '}' of the for loop.

The text between the two arrows is highlighted in gray, indicating the visual selection.

Figure 6-1: Visual mode.

The Vim Tutorial and Reference

d – delete highlighted text

```
#include <stdio.h>
int i;
int main()
{
    for (i = 1; i <= 10; ++i)
        return (0);
~
~
~
three.c
3 fewer lines
```

Figure 6-2: Visual delete.

Note: Although the normal commands **x**, **d**, and **** do different things, in visual mode, they all delete the highlighted text.

The Three Visual Modes

There are actually three different visual modes. The **v** (lower case v) command starts a character-by character visual mode. All the characters from the start to the cursor are highlighted. Figure 6-3 shows this mode in action. The **V** (upper case V) command starts linewise visual mode. You can highlight only full lines in this mode (see Figure 6-4).

Visual start
(Cursor is here
when **v** is
pressed.)

```
Oh woe to Mertle the turtle
who found web surfing quite a hurtle.
    The system you see
        was slower than he.
And that's not saying much for the turtle.
~
~
[No File] [+]
-- VISUAL --
```

Visual end
(Cursor is moved
to here)

Figure 6-3: **v** (visual mode).

The Vim Tutorial and Reference

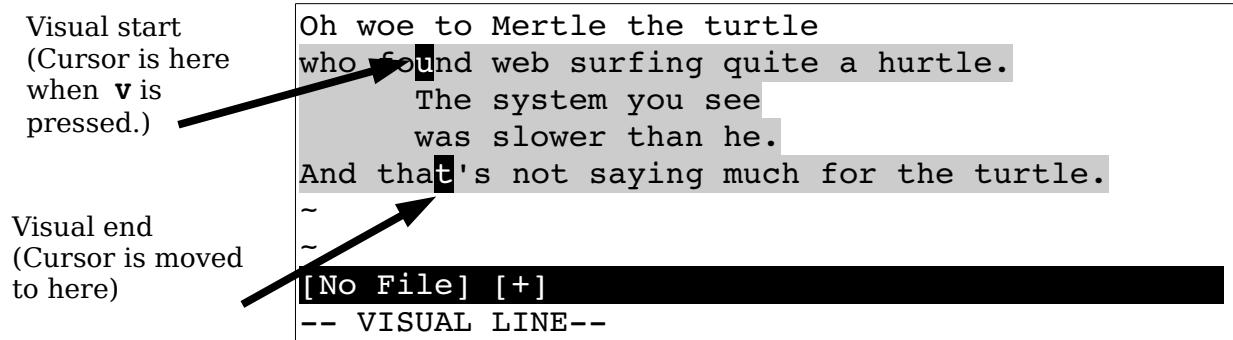


Figure 6-4: **v** (line visual mode).

Note: To get help on the commands that operate in visual mode, use the prefix **v_**. Therefore

```
:help v_d
```

describes what the **d** command does in visual mode.

To highlight a rectangle on the screen, use **CTRL-V** as shown in Figure 6-5. This mode is extremely useful if you want to work with tables. You can highlight a column and delete it using the **d** command.

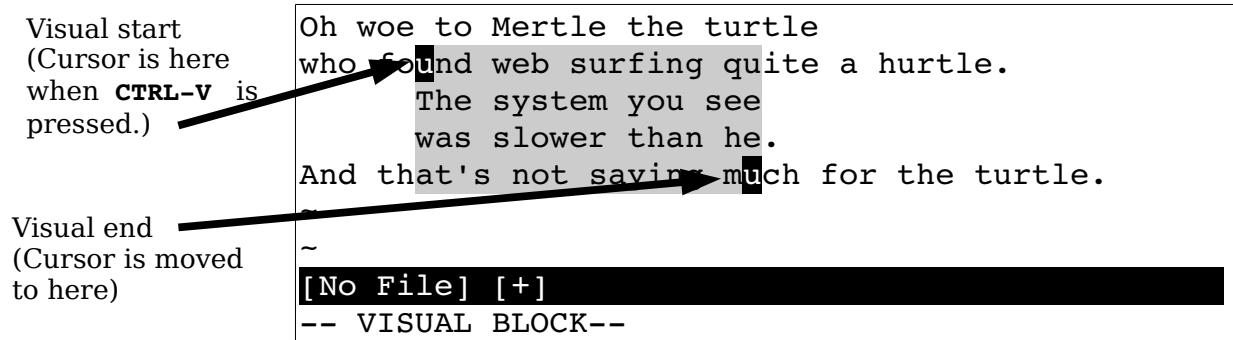


Figure 6-5: **CTRL-V** (block visual mode).

Leaving Visual Mode

Normally, you leave visual mode by typing a visual-mode command, such as **d** to delete the highlighted text. But you can also cancel visual mode by pressing the **<Esc>** key.

Remember, you can always type **<Esc>** to get back to normal mode so you know where you are. Some people find **<Esc>** a little annoying because it beeps if you type it twice. The first **<Esc>** goes from visual mode to normal mode. The second **<Esc>** in normal mode is an error and generates the beep. (The command **CTRL-C** will do the same thing as well.)

The Vim Tutorial and Reference

If you want to make sure that you are in normal mode and do not want to generate a beep, use the **CTRL-\CTRL-N** command. This acts just like **<Esc>** but without the noise.

Editing with Visual Mode

Editing with visual mode is simple. Select the text using the visual commands just discussed, and then type an editing command. This section shows you how to perform simple edits using a visual selection.

Deleting Text in Visual Mode

The **d** command deletes the highlighted text, as shown in Figure 6-6. The **D** command deletes the highlighted lines, even if only part of a line is highlighted (see Figure 6-7). (**x** does the same thing.)

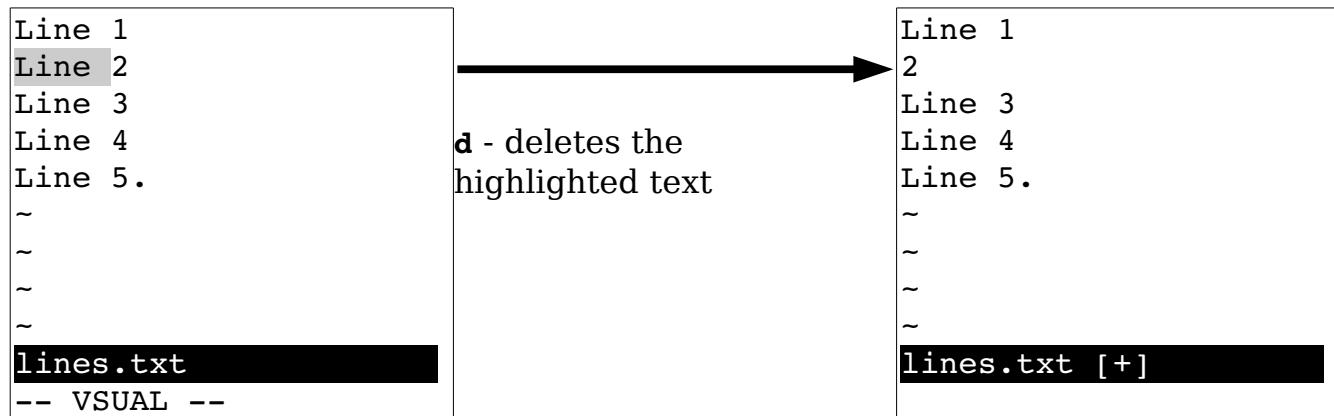


Figure 6-6: Deleting text in visual mode.

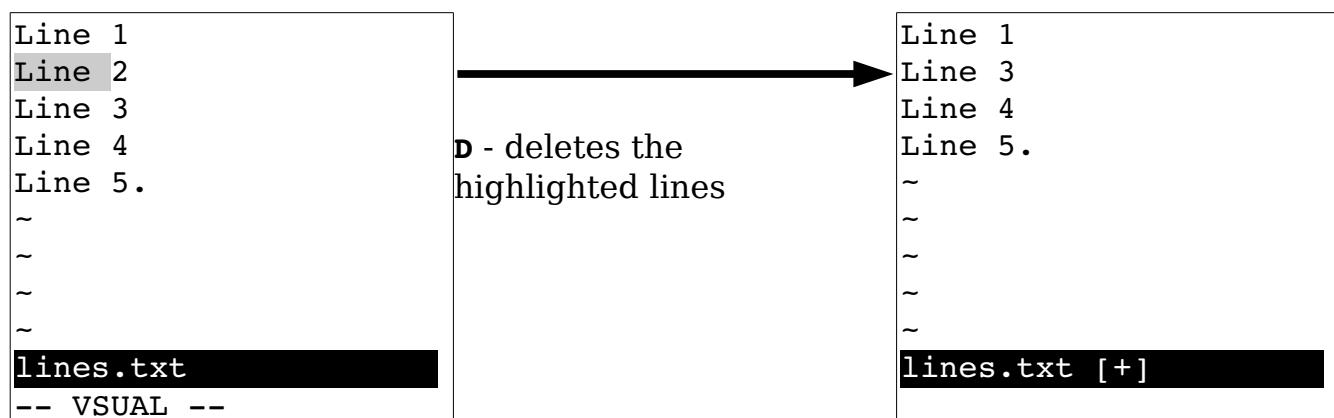


Figure 6-7: The visual D command.

Yanking Text

The **y** command places the highlighted text into a register. The linewise version of this command, **v**, places each line of the highlighted text into a register.

Switching Modes

Suppose you are in character mode (started by **v**) and you realize you want to be in block mode. You can switch to block mode by just pressing **CTRL-V**.

In fact, you can switch visual modes at any time by just selecting the new mode. To cancel visual mode, press the **<Esc>** key; or you can switch to the mode you are already in. (In other words, if you use **v** to start visual mode, you can use another **v** to exit it.)

Changing Text

The **c** command deletes the highlighted text and starts insert mode. The **C** does the same thing, but it works only on whole lines.

Note: **r** and **s** do the same thing as **c** in visual mode. The same thing goes for **R** and **S** and the **C** command.

Joining Lines

The **J** command joins all the highlighted lines into one long line. Spaces are used to separate the lines. If you want to join the lines without adding spaces, use the **gJ** command. Figure 6-8 shows how the **J** and the **gJ** commands work.

The Vim Tutorial and Reference

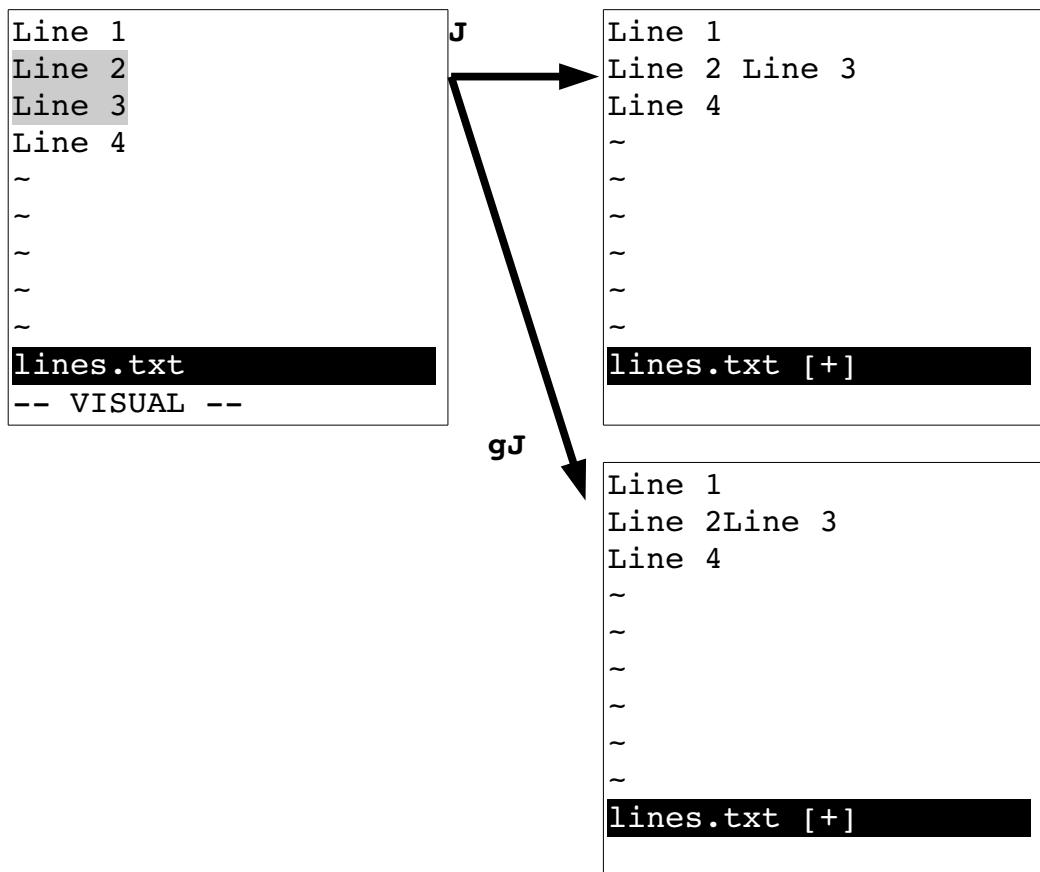


Figure 6-8: The visual `J` and `gJ` command.

Commands for Programmers

The `>` command indents the selected lines by one "shift width." (The amount of white space can be set with the '`shiftwidth`' option.)

The `<` does the process in reverse. (Note that these commands have a different meaning when using visual block mode.) The `=` command indents the text the proper amount according to what *Vim* thinks is the proper information for your program. The `CTRL-]` command will jump to definition of the function highlighted.

Keyword Lookup

The `k` command is designed to look up the selected text using the *man* command. (Linux and UNIX only.) It works just like the normal-mode `k` command except that it uses the highlighted text as the keyword.

Visual Block Mode

Some commands work a little differently in visual block mode. Visual block mode is started by pressing **CTRL-v** and is used to define a rectangle on the screen.

Inserting Text

The command **I***string***<Esc>** inserts the text on each line starting at the left side of the visual block, as seen in Figure 6-9.

You start by pressing **CTRL-v** to enter visual block mode. Now you define your block. Next you type **I** to enter insert mode followed by the text to insert. As you type, the text appears on the first line. After you press **<Esc>** to end the insert, the text will magically be inserted in the rest of the lines contained in the visual selection. Figure 6-9 shows how this process works.

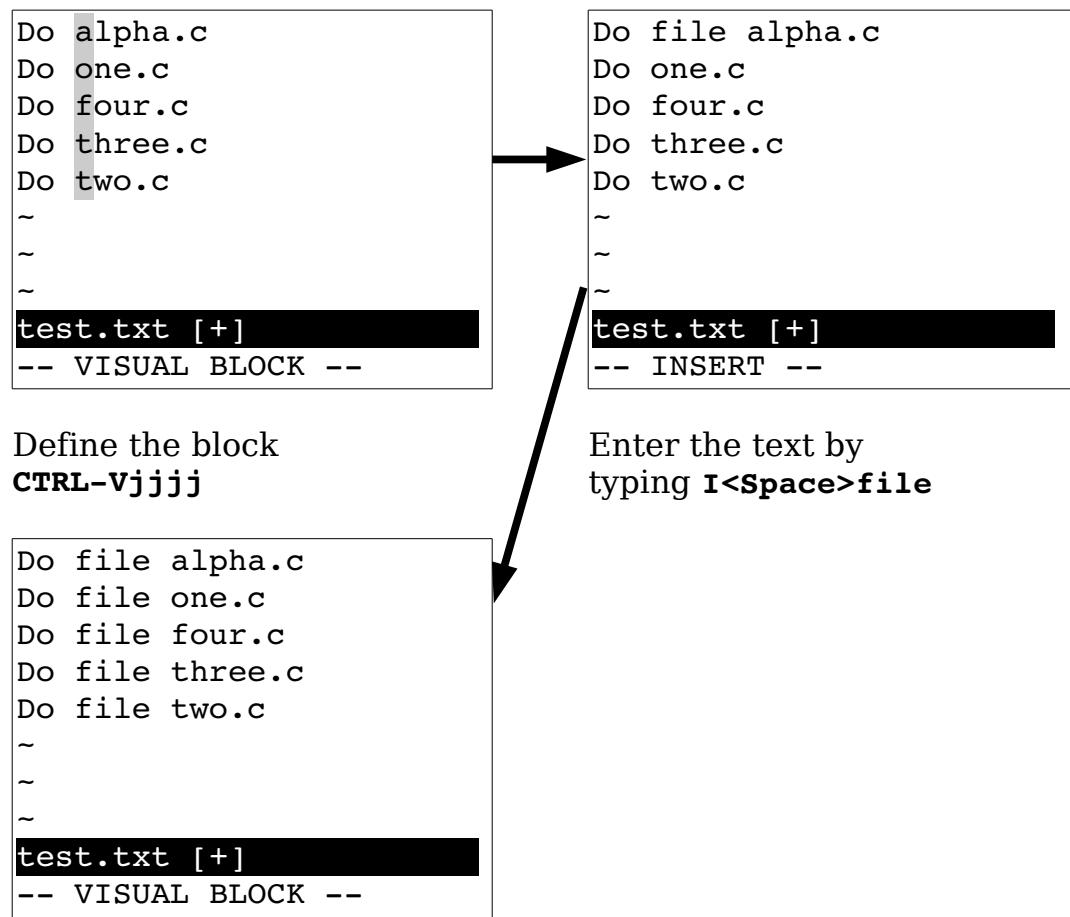
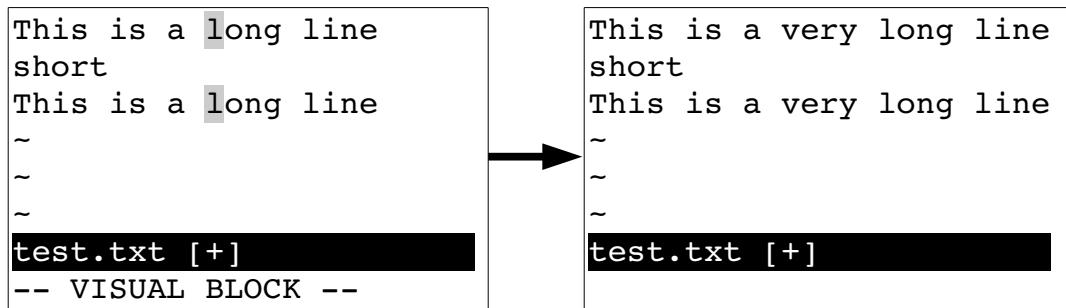


Figure 6-9: Inserting text in visual block mode.

If the block spans short lines that do not extend into the block, the text is not inserted in that line. Figure 6-10 illustrates what happens to short lines. If the string contains a newline, the **i** acts just like a normal-mode insert (**i**) command and affects only the first line of the block.



Select a block.
Notice that the short line is
not part of the selection.

Insert "very" in the line.
Notice that "Short" is
unchanged.

Figure 6-10: Inserting with short lines.

Changing Text

The visual block **c** command deletes the block and then throws you into insert mode to enable you to type in a string. The string will be inserted on each line in the block (see Figure 6-11). The **c** command works only if you enter less than one line of new text. If you enter something that contains a newline, only the first line is changed. (In other words, visual block **c** acts just normal-mode **c** if the text contains more than one line.)

The Vim Tutorial and Reference

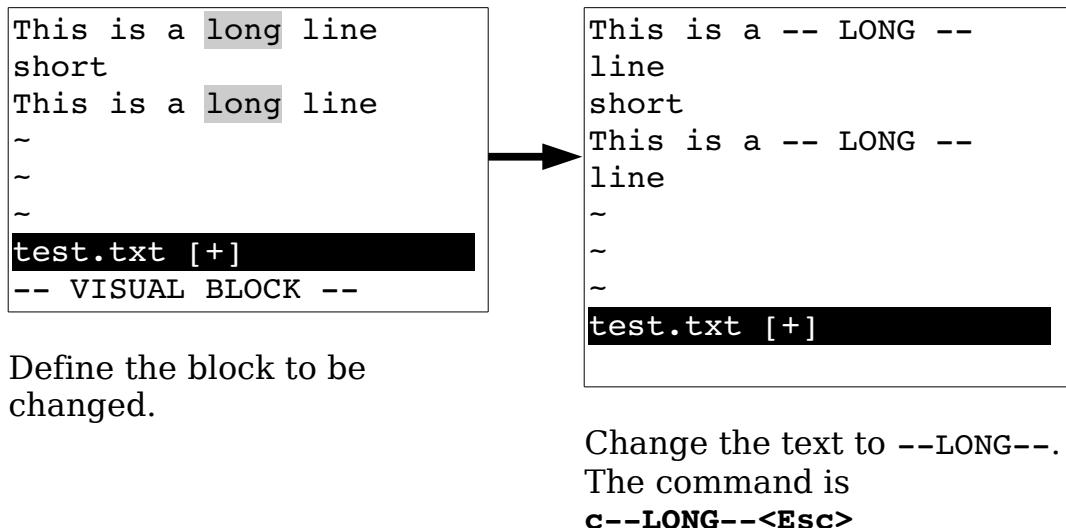


Figure 6-11: Block visual **c** command.

Note: The string will not be inserted on lines that do not extend into the block. Therefore if the block includes some short lines, the string will not be inserted in the short lines.

The **c** command deletes text from the left edge of the block to the end of line. It then puts you in insert mode so that you can type in a string, which is added to the end of each line (see Figure 6-12). Again, short lines that do not reach into the block are excluded. (**R** and **s** act just like the **c** command except they replace the text with what's insert instead of replacing multiple lines.)

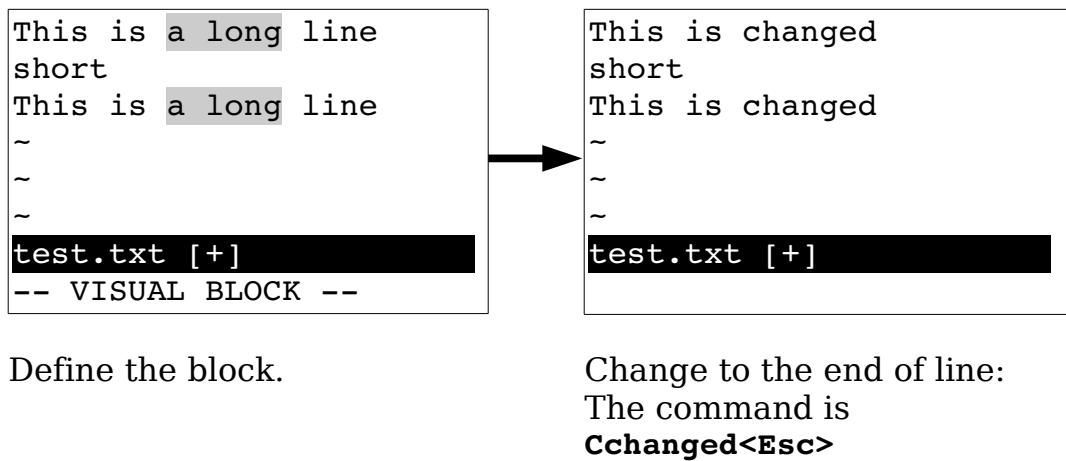


Figure 6-12: Block visual **C** with short lines.

The visual block **A** throws *Vim* into insert mode to enable you to input a string. The string is appended to the block (see Figure 6-13). If there are short lines in the block, spaces are added to pad the line and then string is appended.

The Vim Tutorial and Reference

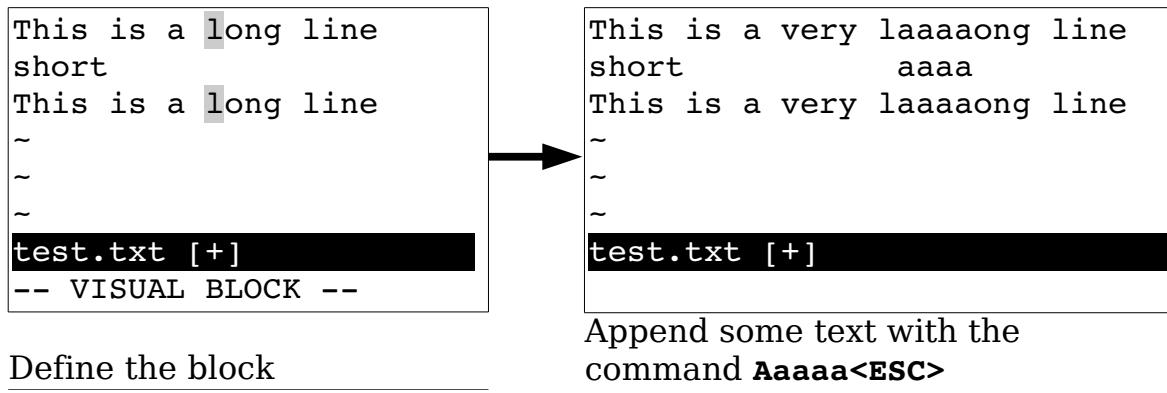


Figure 6-13: Block visual A command.

You can define a right edge of a visual block in two ways. If you use just motion keys, the right edge is the edge of the highlighted area. If you use the \$ key to extend the visual block to the end of line, the inserted text will add the text to the end of each line (see Figure 6-14).

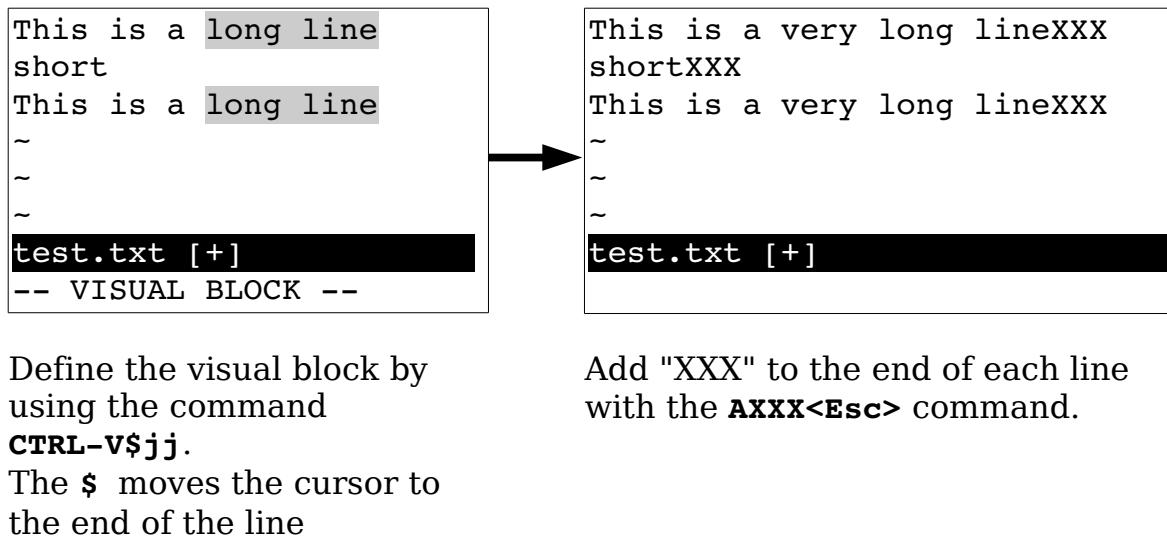


Figure 6-14: Block visual \$ and A commands.

Replacing

The **rchar** command applies all the selected characters with a single character (see Figure 6-15). Short lines that do not extend into the block are not affected.

The Vim Tutorial and Reference

```
This is a long line  
short  
This is a long line  
~  
~  
~  
test.txt [+] -- VISUAL BLOCK --
```

```
This is a very xxxx line  
short  
This is a very xxxx line  
~  
~  
~  
test.txt [+]
```

Define the block

The **rx** command replace all characters in the block with x

Figure 6-15: Block visual-mode r command.

Shifting

The command **>** shifts the text to the right one shift width, opening whitespace. The starting point for this shift is the left side of the visual block (see Figure 6-16). The **<** command removes one shift width of whitespace at the left side of the block (see Figure 6-17). This command is limited by the amount of text that is there; so if there is less than a shift width of whitespace available, it removes what it can.

```
Oh woe to Mertle the turtle  
who found web surfing quite a hurtle.  
The system you see  
was slower than he.  
And that's not saying much for the turtle.
```

```
Oh woe t o Mertle the turtle  
who foun d web surfing quite a hurtle.  
Th e system you see  
wa s slower than he.  
And that 's not saying much for the turtle.
```

Figure 6-16: Block visual-mode > command.

12

The diagram illustrates the effect of the < command in Block visual mode. It consists of two vertically aligned rectangular boxes. The top box contains the text:

```
aaa BBBB BBBB BBBB  
-- VISUAL BLOCK --
```

A black arrow points from the bottom of the first box down to the top of the second box. The second box contains the text:

```
aaa BBBB BBBB BBBB  
aaa BBBB BBBB BBBB
```

Figure 6-17: Block visual < command.

Visual Block Help

Getting help for the commands that use visual block mode differs a little from other commands. You need to prefix the command with **v_b_**. To get help on the visual block r command, for example, type the following:

```
:help v_b_r
```

Chapter 7: Commands for Programmers

The *Vim* editor contains a large number of commands to make life easier for programming. For example, *Vim* contains a number of commands to help you obtain correct indentation, match parentheses, and jump around in source files.

One of the best features of *Vim* as far as a programmer is concerned are the commands that enable you to build a program from within *Vim* and then go through the error list and edit the files that caused trouble. In this chapter, you learn about the following:

- Syntax coloring
- Automatic indentation
- Indentation commands
- Commands to navigate through the source code
- Getting information through the *man* command.
- The use of tags to go up and down a call stack
- Making programs with the **:make** command
- File searching with **:vimgrep**

Syntax Coloring

The following command turns on syntax coloring.

```
:syntax on
```

(**:syntax on** can be abbreviated **:sy on**.)

That means that things such as keywords, strings, comments, and other syntax elements will have different colors. (If you have a black-and-white terminal, they will have different attributes such as bold, underline, blink, and so on.) You can customize the colors used for syntax highlighting as well as the highlighting method itself.

To turn off syntax highlighting use the **:syntax off** (**:sy off**) command.

Syntax Coloring Problems

Most of the time syntax coloring works just fine. But sometimes it can be a little tricky to set up. The following sections take a look at some common problems and solutions.

Colors Look Bad When I Use Vim (UNIX only)

Ever try and read light yellow text on a white background? It is very hard to do. If you see this on your screen, you have a problem. The *Vim* editor has two sets of syntax colors. One is used when the background is light, and the other when the background is dark. When *Vim* starts, it tries to guess whether your terminal has a light or dark background and sets the option '**background**' to light or dark. It then decides which set of colors to use based on this option. Be aware, however, that the editor can guess wrong.

To find out the value of the 'background' option, use the following command:

```
:set background?
```

If *Vim*'s guess is not correct, you can change it using a command such as this:

```
:set background=light
```

You must use this command before executing the command:

```
:syntax on
```

I Turned on Syntax Colors, but All I Get Is Black and White (UNIX)

A common problem affects the *xterm* program used on many UNIX systems. The problem is that although the *xterm* program understands colors, the terminal description for *xterm* frequently omits colors. This cripples syntax coloring. To correct this problem, you need to set your terminal type to the color version. On many versions of Linux this is *xterm-color*, and on Solaris this is *xtermc*.

To fix this problem you need to know what shell (command process) you are using. If you use *csh*, put the following in your *\$HOME/.cshrc* file:

```
if ($term == xterm) set term = xterm-color
```

For *bash* put the following in your *\$HOME/.bashrc* file:

```
if [ $TERM = xterm]; then export TERM=xterm-color; fi
```

Other systems and other shells require different changes.

The Vim Tutorial and Reference

I'm Editing a C File with a Non-Standard Extension. How Do I Tell Vim About It?

The *Vim* editor uses a file's extension to determine the file type. For example, files that end in *.c* or *.h* are considered C files. But what if you are editing a C header file named *settings.inc*?

Because this is a non-standard extension, *Vim* will not know what to do with it. So how do you tell *Vim* that this is a C file? The answer is to use the option '**filetype**'. This tells *Vim* which type of syntax highlighting to use. With a C file, you use the following command:

```
:set filetype=c
```

Now *Vim* knows that this file is a C file and will highlight the text appropriately. If you want to make this setting automatically, look in the help files with this command:

```
:help new-filetype
```

Running the Color Test

If you are still having trouble with colors, run the *Vim* color test. This is a short *Vim* program that displays all the colors on the screen so that you can verify the correctness of the *Vim* colors. The color test can be started with these two commands:

```
:edit $VIMRUNTIME/syntax/colortest.vim  
:source %
```

Matching Pairs

The **%** command is designed to match pairs of **(**, **}**, or **[**. Place the cursor on one, type **%** and you will jump to the other. Figure 7-1 shows how the **%** command works.

The Vim Tutorial and Reference

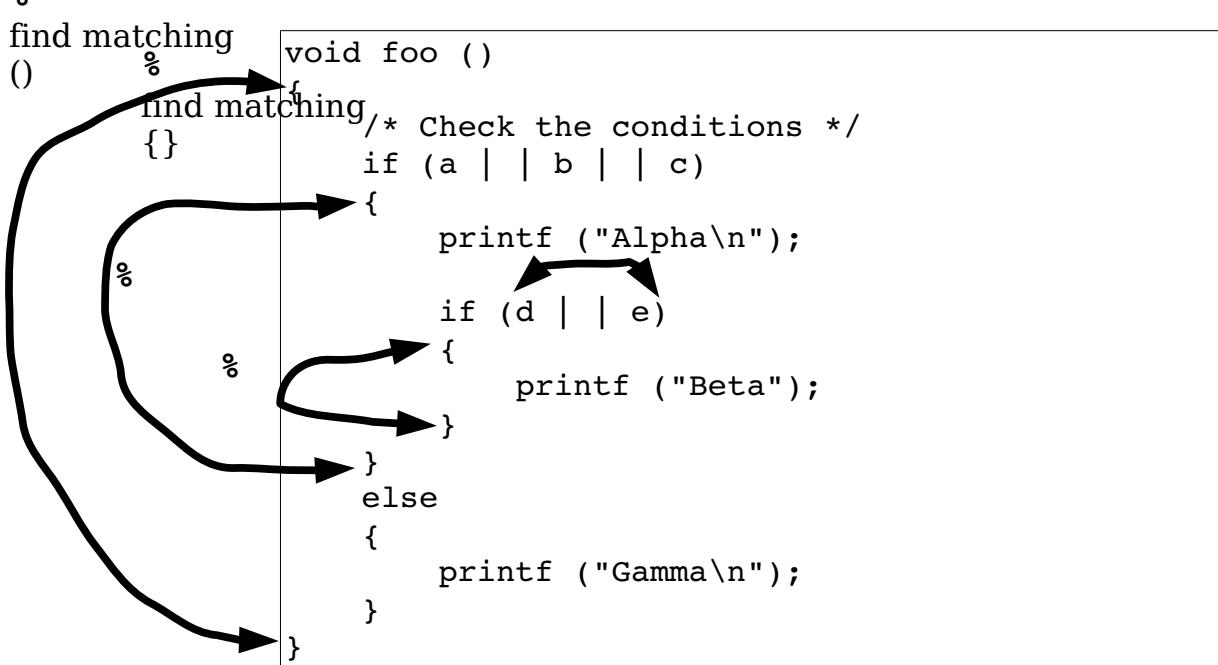


Figure 7-1: % command.

The 2% command will also match the ends of C comments (see Figure 7-2). (For you non-C programmers, these begin with /* and end with */.)

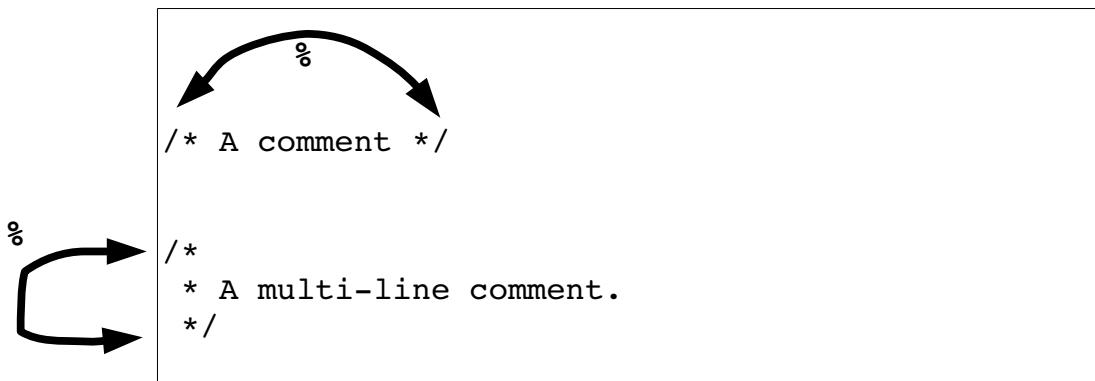


Figure 7-2: % and comments.

Also the % command will match #ifdef with the corresponding #endif. (Same goes for #ifndef and #if.) For #if, #else, and #endif sets, the % command will jump from the #if to the #else, and then to the #endif and back to the #if. Figure 7-3 shows how % works with preprocessor directives.

The Vim Tutorial and Reference

A code snippet illustrating the matching of percentage signs (%) and preprocessor directives (#if, #else, #endif). The code is:

```
#ifndef SIZE  
#define SIZE 100  
#endif /* SIZE */  
  
#ifdef UNIX  
%  
#define EOL "\n";  
#else /* UNIX */  
%  
#define EOL "\r\n";  
#endif /* UNIX */
```

Annotations with arrows show the matching pairs: one arrow from the first % to the first #endif, and three arrows from the three subsequent % signs to the three #else/* UNIX */ blocks.

Figure 7-3: % and the #if/#else/#endif.

Note: The *Vim* editor is smart about matching pairs of operators. It knows about strings, and {} or [] will be ignored inside a string.

Shift Commands

The *Vim* editor has lots of commands that help the programmer indent his program correctly. The first ones discussed here merely shift the text to the left (<<) or the right (>>).

The left shift command (<<) shifts the current line one shift width to the left. The right shift command (>>) does the same in the other direction.

But what is a shift width? By default, it is 8. However, studies have shown that an indentation of 4 spaces for each level of logic is the most readable. So a shift width of 4 would be much nicer. To change the size of the shift width, use the following command:

```
:set shiftwidth=4
```

Figure 7-4 shows how the shift width option affects the >> command.

A code snippet illustrating the effect of the shiftwidth option on the >> command. The code is:

```
printf ( "Hello world! \n") ; >>  
printf ( "Hello world! \n") ;  
  
printf ( "Hello world! \n") ; >>  
printf ( "Hello world! \n");
```

Annotations with arrows show the indentation of the second printf line and the third printf line relative to the first, demonstrating the effect of different shiftwidth values.

Figure 7-4: 'shiftwidth' and >>.

The Vim Tutorial and Reference

The << command shifts a single line. As usual you can prefix this command with a count; for example, 5<< shifts 5 lines. The command <**motion** shifts each line from the current cursor location to where motion carries you.

You can also highlight a set of lines in visual mode (**v** command) and then shift them with < or >.

Automatic Indentation

The *Vim* editor has a variety of automatic indentation options. The major indentation modes are the following:

- | | |
|--------------------|--|
| cindent | This works for C-style programs (C, C++, Java, and so on). When this style of indentation is enabled, the <i>Vim</i> editor automatically indents your program according to a "standard" C style. |
| smartindent | In this mode, <i>Vim</i> indents each line the same amount as the preceding one, adding an extra level of indentation if the line contains a left curly brace ({}) and removing a indentation level if the line contains a right curly brace (}). An extra indent is also added for any of the keywords specified by the ' 'cinwords' ' option. |
| autoindent | New lines are indented the same as the previous line. |

The next few sections explore these indentation modes in detail.

C Indentation

The *Vim* editor knows something about how C, C++, Java, and other structured language programs should be indented and can do a pretty good job of indenting things properly for you. To enable C-style indentation, just execute the following command:

```
:set cindent
```

With this option enabled, when you type something such as `if (x)`, the next line will automatically be indented an additional level. Figure 7-5 illustrates how '**'cindent'**' works.

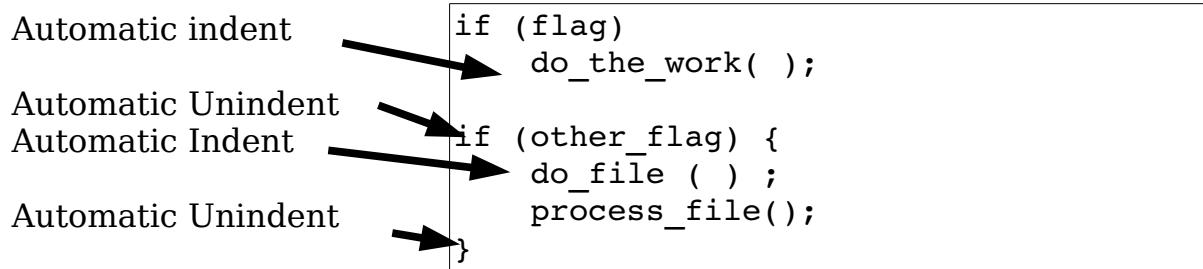


Figure 7-5: **cindent**.

When you type something in curly braces ({}), the text will be indented at the start and unindented at the end.

Note: One side effect of automatic indentation is that it helps you catch errors in your code early. I have frequently entered a } to finish a procedure, only to find that the automatic indentation put it in column 4.

This told me that I had accidentally left out a } somewhere in my text.

Different people have different styles of indentation. By default *Vim* does a pretty good job of indenting in a way that 90% of programmers do. There are still people out there with different styles, however; so if you want to, you can customize the indentation style through the use of several options.

You don't want to switch on the '**cindent**' option manually every time you edit a C file. This is how you make it work automatically: Put the following lines in your *.vimrc* (UNIX) or *_vimrc* (Windows) file:

```
:filetype on  
:autocmd FileType c, cpp :set cindent
```

(**:filetype** can be abbreviated **:filet**. **:autocmd** may be abbreviated **:au**.)

The first command (**:filetype on**) turns on *Vim*'s file type detection logic. The second, performs the command **:set cindent** if the file type detected is *c* or *cpp*. (This includes C, C++, and header files.)

You can use an autocommand to set the '**filetype**' as well. There is a special command **:set filetype** (**:setf**) which sets the '**filetype**' only if it has not already been set by another **:autocmd**. For example:

The Vim Tutorial and Reference

```
:autocmd BufRead *.cpp :set filetype=c
:autocmd BufRead *.html :set filetype=html
" Any other file types, set to text
:autocmd BufRead * :set filetype text
```

Smartindent

The '**cindent**' mode is not the only indent mode available to *Vim* users. There is also the '**smartindent**' mode. In this mode, an extra level of indentation is added for each { and removed for each }. An extra level of indentation will also be added for any of the words in the '**cinwords**' option.

Lines that begin with # are treated specially. If a line starts with #, all indentation is removed. This is done so that preprocessor directives will all start in column 1. The indentation is restored for the next line.

Note '**smartindent**' is not as smart as '**cindent**', but smarter than '**autoindent**'.

Autoindent

Structured languages such as a Pascal, Perl, and Python use indentation to help the programmer figure out what the program is doing. When writing these programs, most of the time you want the next line indented at the same level as the preceding one. To help you do this, the *Vim* editor has an '**autoindent**' option. When on, it causes lines to be automatically indented.

Suppose, for example, that you have '**autoindent**' off (**:set noautoindent**). To type the following text, you must type four spaces in front of each printf:

```
if (true) {
    printf("It is true\n");
    printf("It is really true\n");
}
```

The ↘ character indicates a typed space.

If you have set the '**autoindent**' option using the **:set autoindent** command, the *Vim* editor automatically indents the second printf by four spaces (to line up with the preceding line). Figure 7-6 illustrates the operation of the '**autoindent**' option. Type four spaces for indent; with '**autoindent**' set, the following lines are automatically indented.

```
if (true) {  
    printf("It is true\n");  
    printf("It is really true\n");  
}
```

Figure 7-6: '**autoindent**'

That is nice, but when you get ready to enter the } line, the *Vim* editor also indents four spaces. That is not good because you want the } to line up with the if statement. While in insert mode, the **CTRL-D** command will cause *Vim* to back up one shift width (see Figure 7-7). **CTRL-D** moves the } back one shift width.

13

```
if (true) {  
    printf("It is true\n");  
    printf("It is really true\n");  
}
```

Figure 7-7: **CTRL-D**

The = Command

The **=motion** command indents the selected text using *Vim*'s internal formatting program. If you want to indent a block of text, for example, you can use the = command to do it. The motion in this case is the % (go to matching {}) command. Figure 7-8 shows the results.

```
{  
if (strcmp (arg, option1) == 0)  
return (1) ;  
if (strcmp (arg, option2) == 0)  
return (1) ;  
return (0) ;  
}
```

- 1) Position cursor on the first "{"
- 2) Execute the command =%.

The Vim Tutorial and Reference

```
{  
    if (strcmp (arg, option1) == 0)  
        return (1) ;  
    if (strcmp (arg, option2) ==0)  
        return (1) ;  
    return (0) ;  
}
```

Figure 7-8: The = command.

Another way of doing this is to use the visual mode = command. For example, to indent all the code inside a set of {} (including the {}), execute the following commands:

1. Position the cursor on the starting {.
2. Start visual mode with **v**.
3. Go to the other } with the % command.

```
{  
if (strcmp (arg, option1) == 0)  
return (1) ;  
if (strcmp (arg, option2) == 0)  
return (1) ;  
return (0) ;  
}
```

4. Press = to indent the text.

```
{  
    if (strcmp (arg, option1) == 0)  
        return (1) ;  
    if (strcmp (arg, option2) ==0)  
        return (1) ;  
    return (0) ;  
}
```

Figure 7-9: The visual= command.

Diff Mode

Vim has a diff mode which displays the differences between two or more files side by side. For example, let's take a look at the difference between two different version of a program. To do this we execute the command:

```
$ gvimdiff dlt-status.cpp.bad dlt-status.cpp.works
```

The Vim Tutorial and Reference

Figure 7-10 shows the results:

The screenshot shows a Vim window titled "dlt-status.cpp.bad (/media/Data...diff/mtstatus) (1 of 2) - GVIMDIFF". It displays two files side-by-side for comparison. The left pane is labeled "dlt-status.cpp.bad" and the right pane is labeled "<dlt-status.cpp.works". Both panes show the same code content, which includes #include directives, variable declarations (const int EX_LOG_SENSE = 0x4D;), static variables (static int raw = 0;), and an enum page_code. A fold is visible at the top of both panes, spanning approximately 29 lines. In the right pane, the line "static int raw = 0;" is highlighted in pink, and the character 'i' in "iint" is highlighted in red. A blue dashed line separates the common code from the differences. The status bar at the bottom indicates "dlt-status.cpp.bad 1,1 Top <dlt-status.cpp.works 1,1 Top" and the file path "dlt-status.cpp.works" with "695L, 20161C".

Figure 7-10: Results of gvimdiff

The first 29 lines are the same in both files so *Vim* has put in a fold hiding them from you. The line:

```
static int raw;
```

is different between the two files. Not only is the line highlighted, but the actual different (and extra "i") is highlighted in a different color. You can also easily see where a line was added in the first file as well.

The nice thing about doing this in *Vim* is that you can edit both files. To move from one window to another use the window left (**CTRL-W h**) or window right (**CTRL-W l**) commands.

The Vim Tutorial and Reference

You can also easily move differences from one file to another. To move a change from the current file to the other one, use the *dp* (diff put) command. To move a change from the other file to the one you are editing now, use the *do* (diff obtain) command.²

Folding

Suppose you are looking at some badly written code such as that in Figure 7-11.

```
if (condition) {  
    // ... 1200 lines of code  
} else {  
    return (ERROR);  
}
```

Figure 7-11: Bad code

If you look at the code normally, this is all you're going to see:

```
if (condition) {  
    // Junky code line 1  
    // Junky code line 2  
    // Junky code line 3  
    // Junky code line 4
```

Now if you scroll down, and down, and down, and down, you'll finally get the to the **else** clause. As you see figuring out the structure of such code is difficult.

The *Vim* fold feature helps you see the structure of this style of code. Let's see how to use this feature. To do this we execute the following steps:

1. Put the cursor on the first line of junk code
2. Press **v** to start visual line mode.
3. Put the cursor on the "{" on the previous (**if**) line.
4. Press **%** to go to the matching "}".
5. Go up one line with the **j** command.
6. Create a fold with the **zf** command

Figure 7-12 displays the result:

² **dg** was already taken as it is the beginning of a **d{motion}** command.

The Vim Tutorial and Reference

```
#if (x) {  
+--1200 lines: Junk code line 1-----  
} else {  
    return (ERROR);  
}
```

Figure 7-12: Screen after folding

This makes it easy to see the structure of the program. If you want to see what's in the fold that you just created, position the cursor on the fold and execute the command **zo** (Fold Open).

Note: Power users can do the same thing by positioning the cursor on the "{" and entering **zfi{**. (This is the **zf{motion}** command with the inner { select command: **i{.**})

So far we've been creating folds manually. This assumes that the '**foldmethod**' ('**fdm**') is set to manual, the default.

We can also define folds based on indentation. Figure 7-13 shows some unfolded code.

```
if (a) {  
    if (b) {  
        if (d) {  
            if (e) {  
                if (f) {  
                    do_something();  
                }  
            }  
        }  
    }  
}
```

Figure 7-13: Unfolded text

Now let's set the '**foldmethod**' to **indent**.

```
:set foldmethod=indent
```

Suddenly all our code is folded. (See Figure 7-14.)

```
if (a) {  
+-- 9 lines: if (b) {-----  
}
```

Figure 7-14: Indent Level Folding

The Vim Tutorial and Reference

Now let's put the cursor on the fold and do a **zo**. One level of indentation is displayed. (See Figure 7-15.)

```
if (a) {  
    if (b) {  
+--- 7 lines: if (d) {-----  
    }  
}
```

Figure 7-15: One level of indent unfolded

If we do another **zo**, another level of indentation is opened up. The **zc** command closes one level of folding.

The **zo** and **zc** commands open and close folds manually. In other words this commands override what *Vim* would normally do. To reset the folding to the defaults (no overrides) with the command **zx**.

Sometimes a folding will cause the line the cursor is on to disappear. If this happens the **zv** command opens just enough folds to make the line the cursor is on visible.

So far we've been opening and closing folds manually. The '**foldlevel**' ('**fdl**') option controls how many level of indents cause *Vim* to fold. For example, if you set the '**foldlevel**' to 3 anything indented three or more '**shiftwidth**' ('**sw**') indentations is folded.

You can use the **:set** command to adjust the '**foldlevel**' but this is cumbersome. Instead you can use the **zm** (fold more) command to reduce the '**foldlevel**' increasing the amount of text folded. The **zr** (fold reduce) increases the '**foldlevel**', reducing the amount of text folded. Figure 7-16 shows how this works.

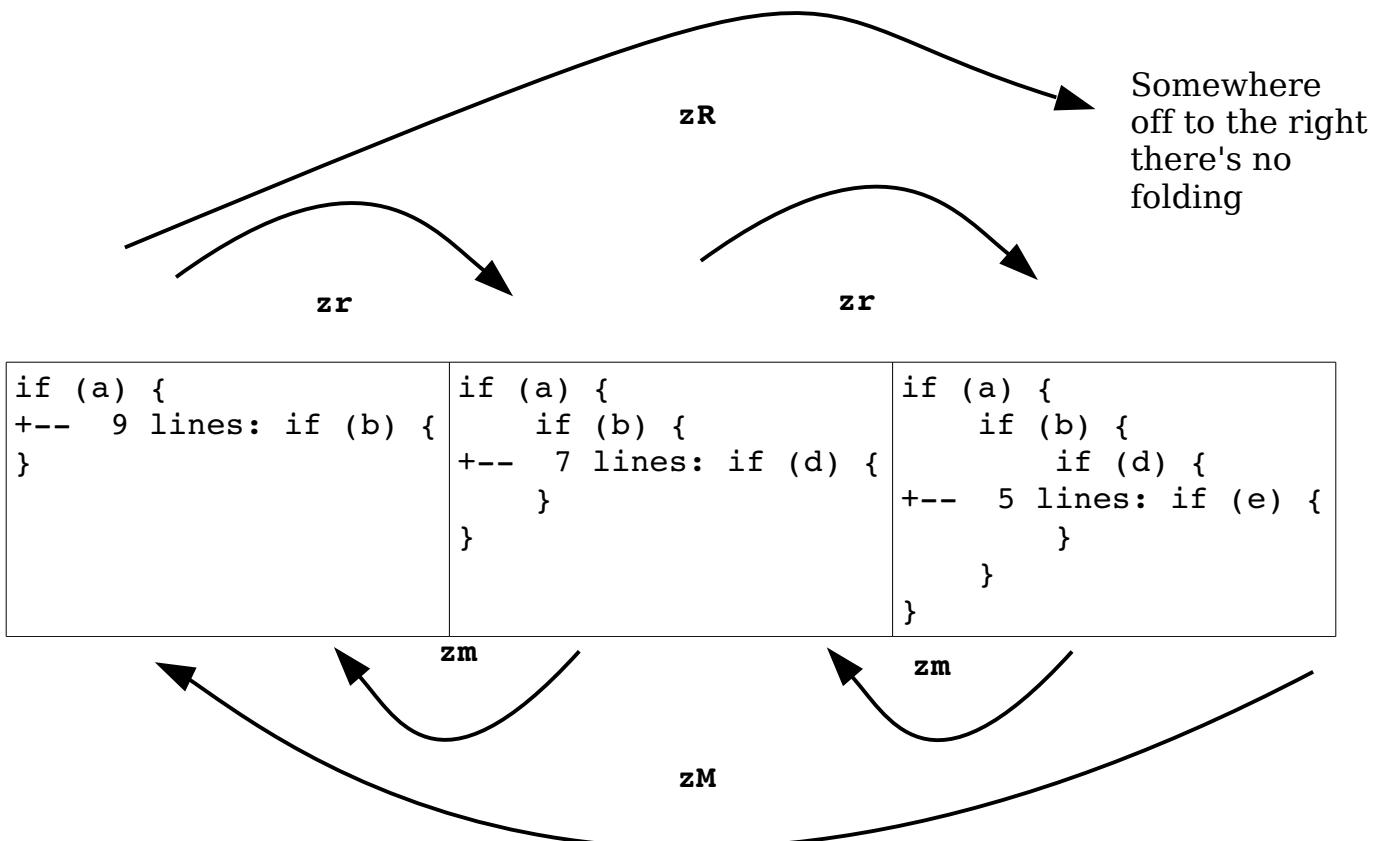


Figure 7-16: Increasing and decreasing the folding

Finally to totally get rid of folding use the **zR** command. To fold things to the max use the **zM** command.

C and C++ are problem languages in that they have these nasty pre-processor directives that don't follow the normal indenting rules. To handle these types of lines the '**foldignore**' ('**fdi**') is used. It tells *Vim* that any lines that begin with a certain character are *not* to be considered for computing the indentation of a line. Instead these lines inherit their indentation level from the lines above or below.

Initially when you start editing the '**foldlevel**' is set to 1. If you want to have it start at a different value, set the '**foldlevelstart**' ('**fdls**') to whatever you want the initial value to be.

Locating Items in a Program

Programmers made the *Vim* editor, so it has a lot of commands that can be used to navigate through a program. These include the following:

The Vim Tutorial and Reference

CTRL-I,]CTRL-I	Search for a word under the cursor in the current file and any brought in by #include directives.
gd, gD	Search for the definition of a variable.
]CTRL-D, [CTRL-D	Jump to a macro definition.
]d, [d,]D, [D	Display macro definitions.

Instant Word Searches Including #include Files ([CTRL-I,]CTRL-I)

The [CTRL-I command jumps to the word under the cursor. The search starts at the beginning of the file and also searches files brought in by #include directives. The]CTRL-I does the same thing, starting at the cursor location.

Jumping to a Variable Definition (gd, gD)

The **gd** command searches for the local declaration of the variable under the cursor (see Figure 7-17). This search is not perfect because *Vim* has a limited understanding of C and C++ syntax. In particular, it goes to the wrong place when you ask for the local declaration of a global variable. Most of the time, however, it does a pretty good job.

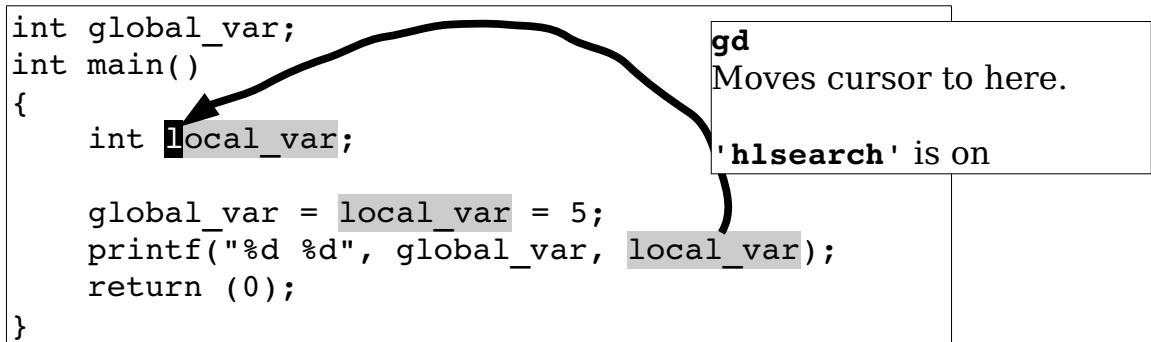


Figure 7-17: The **gd** command.

The **gD** command searches for the global definition of the variable under the cursor(see Figure 7-18). Again, this search is not perfect, but most of the time it does the right thing.

The Vim Tutorial and Reference

A code snippet in Vim:

```
int global_var;
int main()
{
    int local_var;

    global_var = local_var = 5;
    printf("%d %d", global_var, local_var);
    return (0);
}
```

The word `global_var` is highlighted. A curved arrow points from the cursor position to the `gd` command in a callout box.

gd
Moves cursor to here.
'hlsearch' is on

Figure 7-18: The **gd** command.

Jump to Macro Definition ([CTRL-D],]CTRL-D)

The **[CTRL-D** command searches for the first definition of the macro whose name is under the cursor. The **]CTRL-D** command searches for the next definition of the macro. These commands search not only the file you are editing, but also all files that are #included from this file. Figure 7-19 shows the **[CTRL-D** and **]CTRL-D** commands.

A code snippet in Vim:

```
#include <stdio.h>
#define SIZE 10
int i = EOF ;
int main ( )
{
    for (i = 1; i <= SIZE; ++i)
    {
        printf( "%2d squared is %3d\nd", i,
i*i)
    ;
        }
    return (0) ;
}
#undef SIZE
#define SIZE 20
```

The word `SIZE` is highlighted. Three curved arrows point from the cursor position to the **[CTRL-D]**, **]CTRL-D**, and **SIZE** in a callout box.

Figure 7-19: **[CTRL-D]** and **]CTRL-D**.

Displaying Macro Definitions ([d],]d, [D],]D)

The **[d** command displays the first definition of the macro whose name is under the cursor. The **]d** command does the same thing only it starts looking from the current cursor position and finds the next definition. Figure 7-20 shows the result of **[d**.

The Vim Tutorial and Reference

```
#include <stdio.h>
#define SIZE 10 ← [d]
int i = EOF;
int main()
{
    for (i = 1; i <= SIZE; ++i)
    {
        printf("%2d squared is %3d\n", i,
i*i);
    }
    return (0);
}
#undef SIZE
#define SIZE 20
```

Figure 7-20: [**d** command.

Again, #include files are searched as well as the current one.

The **[D** and **[d** commands list all the definitions of a macro. The difference between the two is that **[d** starts the list with the first definition, whereas **[D** starts the list with the first definition after the cursor. Figure 7-21 shows the results of a **[D** command.

```
int main()
{
    for (i = 1; i <= SIZE; ++i)
    {
        printf("%2d squared is %3d\n", i, i*i);
    }
    return (0);
}
#undef SIZE
#define SIZE 20
-
-
-
test.c
 1:      2 #define SIZE 10
 2:      13 #define SIZE 20
Press RETURN or enter command to continue
```

Figure 7-21: [**D** command.

Shifting a Block of Text Enclosed in {}

Suppose that you want to indent the text enclosed in {} one level. Position the cursor on the first (or last) {}. Execute the command **>%**. This shifts the text right to where the motion takes you. In this case, % takes you to the matching {} . Figure 7-22 shows how these commands work.

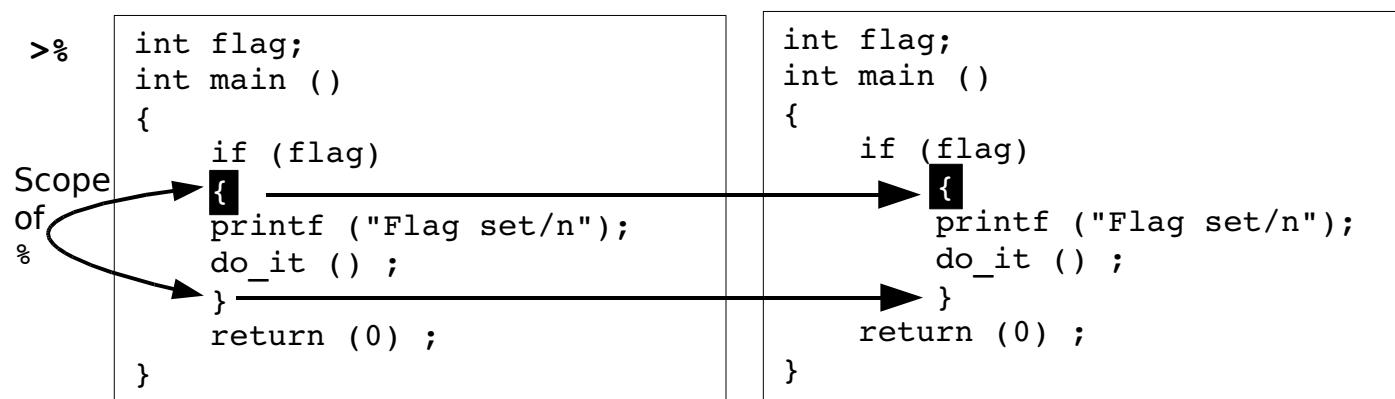


Figure 7-22: Shifting a block of text.

Unfortunately this shifts the {} in addition to the text. Suppose you just want to shift what is in the {}. Then you need to do the following:

1. Position the cursor on the first {}.
2. Execute the command **>i{**.

This shift right command (>) shifts the selected text to the right one shift width. In this case, the selection command that follows is **i{**, which is the "inner {} block" command. Figure 7-23 shows the execution of these commands.

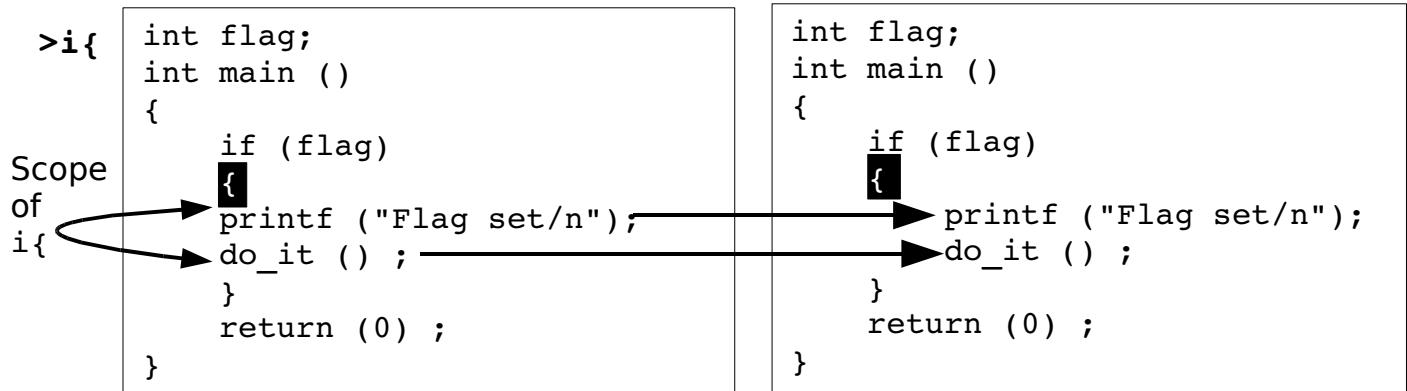


Figure 7-23: Shifting a block of text (better method).

Indenting a Block Using Visual Mode

To indent a block using visual mode, follow these steps:

1. Position the cursor on the left or right curly brace.
2. Start visual mode with the **v** command.
3. Select the inner {} block with the command **i}**.
4. Indent the text with **>**.

Finding the man Pages

The **k** command runs a UNIX *man* command using the word under the cursor as a subject. If you position the cursor on the word *open* and press **k**, for example, the man page for *open* will display.

On Microsoft Windows, the **k** command does the equivalent of performing a **:help (:h, <Help>, <F1>)** on the word under the cursor.

You can also use the visual **k** command to do the same thing. The format of the man command is as follows:

\$ man [section] subject

The Vim Tutorial and Reference

The **k** command gets the subject from the word under the cursor. But what if you need to select the section number? It is simple; the **k** command takes a count argument. If specified, this is used as the section number. Therefore, if you position the **k** over the word `mkdir` and execute the **2k**, you will get the *mkdir(2)* page.

You can customize the Error: Reference source not found**k** command. It runs the program specified by the '**keywordprg**' ('**kp**') option. By default, on UNIX this is *man*. Solaris has a non-standard *man* command. Sections must be specified with the **-s** switch. So the '**keywordprg**' option defaults to *man -s* on Solaris. The *Vim* editor is smart enough to know that if no section is specified, that it must drop the **-s**.

On Microsoft Windows, there is no *man* command, so '**keywordprg**' defaults to nothing (""). This tells *Vim* to use the internal :**help** command to handle the **k** command. Finally, the definition of what the **k** command considers a word is defined by the '**iskeyword**' ('**isk**') option.

Tags

The *Vim* editor can locate function definitions in C and C++ programs. This proves extremely useful when you are trying to understand a program. The location of function definitions (called tags in *Vim* terminology) is stored in a table of contents file generated by the program *ctags*.³ To generate the table of contents file, which is named *tags*, use the following command:

```
$ ctags *.c
```

Now when you are in *Vim* and you want to go to a function definition, you can jump to it by using the following command:

```
:tag function
```

This command will find the function even if it is another file. The **CTRL-]** command jumps to the tag of the word that is under the cursor. (**g<LeftMouse>** and **<C-LeftMouse>** are equivalent to **CTRL-]**.) This makes it easy to explore a tangle of C code.

³ You may need to install Exuberant ctags from <http://ctags.sourceforge.net/> if your system does not have it already.

The Vim Tutorial and Reference

Suppose, for example, that you are in the function `write_block`. You can see that it calls `write_line`. But what does `write_line` do? By putting the cursor on the call to `write_line` and typing **CTRL-J**, you jump to the definition of this function (see Figure 7-24). The `write_line` function calls `write_char`. You need to figure out what it does. So you position the cursor over the call to `write_char` and press **CTRL-J**. Now you are at the definition of `write_char` (see Figure 7-25).

```
void write_block(char line_set[])
{
    int i;
    for (i = 0; i < N_LINES; ++i)
        write_line(line_set[i]);
}
```

CTRL-J goes to the definition of `write_line` (switching files if needed).

The command `:tag write_line` does the same thing

```
void write_line(char line[])
{
    int i;
    for (i = 0; line[0] != '\0')
        write_char(line[i]);
}
~
"write_line.c" 6L,
```

Figure 7-24: Tag jumping with **CTRL-J**.

```
void write_char(char ch)
{
    write_raw(ch);
}
~
"write_char.c" 4L, 48C
```

CTRL-J with cursor on
`write_char` gets us here

Figure 7-25: Jumping to the `write_char` tag.

The `:tags` command shows the list of the tags that you have traversed through (see Figure 7-26).

The Vim Tutorial and Reference

```
~  
:tags  
# TO tag          FROM line      in file/text  
1   1 write_block           1 write_block.c  
2   1 write_line           5 write_block.c  
3   1 write_char           5 write_line.c  
>  
Press RETURN or enter command to continue
```

Figure 7-26: The **:tags** command.

Now to go back. The **CTRL-T** command goes the preceding tag. (The commands **g<RightMouse>** and **<C-RightMouse>** do the same thing.) This command takes a count argument that indicates how many tags to jump back.

So, you have gone forward, and now back. Let's go forward again. The following command goes to the next tag on the list:

```
:tag
```

You can prefix it with a count and jump forward that many tags. For example:

```
:3tag
```

Figure 7-27 illustrates the various types of tag navigation.

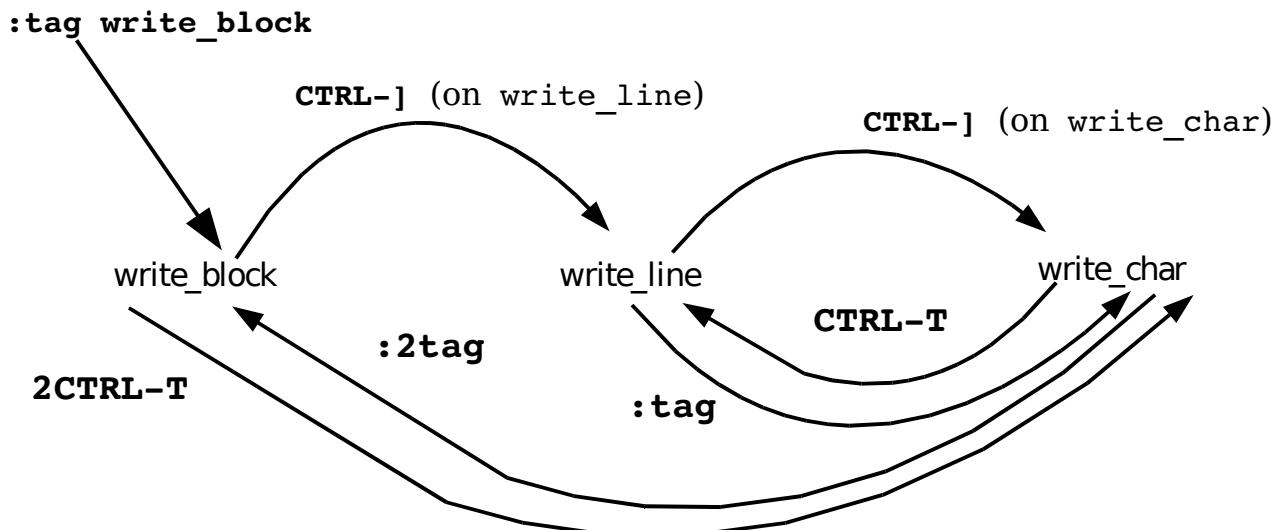


Figure 7-27: Tag navigation.

Help and Tags

The help system makes extensive use of tags. To execute a "hyperlink jump," you press **CTRL-]** (jump to tag). You can return to a preceding subject with **CTRL-T** (jump to preceding tag) and so on.

Windows and Tags

The **:tag** command replaces the current window with the one containing the new function. But suppose you want to see not only the old function but also the new one? You can split the window using the **:split** command followed by the **:tag** command. But *Vim* has a shorthand command that is shorthand for both commands:

```
:stag name
```

Figure 7-28 shows how this command works.

<pre>void write_block(char line_set[]) { int i; for (i = 0; i < N_LINES; write_line(line_set[i]); }</pre>	<pre>void write_char(char ch) { write_raw(ch); write_char.c for (i = 0; i < N_LINES; ++i) write_line(line_set[i]); write_block.c "write_char.c" 4L,</pre>
--	--

```
:stag write_char
```

Figure 7-28: The :stag command.

The **CTRL-W]** command splits the current window and jumps to the tag under the cursor in the upper window (see Figure 7-29). (**CTRL-W** **CTRL-]** works as well.) If a *count* is specified, the new window will be *count* lines high.

<pre>void write_block(char line_set[]) { int i; for (i = 0; i < N_LINES; write_line(line_set[i]); }</pre>	<pre>void write_line(char line[]) { int i; write_line.c for (i = 0; i < N_LINES; ++i) write_line(line_set[i]); write_block.c "write_char.c" 4L,</pre>
--	--

*Figure 7-29: **CTRL-WJ**.*

Finding a Procedure When You Only Know Part of the Name

Suppose you "sort of" know the name of the procedure you want to find? This is a common problem for Microsoft Windows programmers because of the extremely inconsistent naming convention of the procedures in the Windows API. UNIX programmers fare no better. The convention is consistent; the only problem is that UNIX likes to leave letters out of system call names (for example, `creat`).

You can use the `:tag` command to find a procedure by name, or it can search for a regular expression. If a procedure name begins with /, the `:tag` command assumes that the name is a regular expression. If you want to find a procedure named "something write something," for example, you can use the following command:

```
:tag /write
```

This finds all the procedures with the word `write` in their names and positions the cursor on the first one. If you want to find all procedures that begin with `read`, you need to use the following command:

```
:tag /^read
```

If you are not sure whether the procedure is `DoFile`, `do_file`, or `Do_File`, you can use this command:

```
:tag /DoFile\|do_file\|Do_File
```

or

```
:tag /[Dd]o_\=[Ff]ile
```

These commands can return multiple matches. You can get a list of the tags with the following command:

```
:tselect name
```

(`:tselect` may be abbreviated as `:ts`.) Figure 7-30 shows the results of a typical `:tselect` command.

The Vim Tutorial and Reference

```
~          # pri kind tag      file
> 1 F C f write_char      write_char.c
      void write_char(char ch)
2 F f write_block      write_block.c
      void write_block(char line_set[])
3 F f write_line      write_line.c
      void write_line(char line[])
4 F f write_raw       write_raw.c
      void write_raw(char ch)
Enter nr of choice (<CR> to abort): █
```

Figure 7-30: **:tselect** command.

The first column is the number of the tag. The second column is the Priority column. This contains a combination of three letters.

- F Full match (if missing, a case-ignored match)
- S Static tag (if missing, a global tag)
- F Tag in the current file

The last line of the **:tselect** command gives you a prompt that enables you to enter the number of the tag you want. Or you can just press Enter (**<CR>** in *Vim* terminology) to leave things alone.

The **g]** command does a **:tselect** on the identifier under the cursor. The **:tjump** command (a.k.a. **:tj**) works just like the **:tselect** command, except if the selection results in only one item, it is automatically selected. The **gCTRL-]** command does a **:tjump** on the word under the cursor. A number of other related commands relate to this tag selection set, including the following:

The Vim Tutorial and Reference

```
:[count] tnext      Go to the next tag
:[count] tn

:[count] tprevious   Go to the previous tag
:[count] tp

:[count] tNext
:[count] tN

:[count] trewind     Go to the first tag
:[count] tr

:[count] tfirst
:[count] tf

:[count] tlast       Go to the last tag
:[count] tl
```

Figure 7-31 shows how to use these commands to navigate between matching tags of a **:tag** or **:tselect** command.

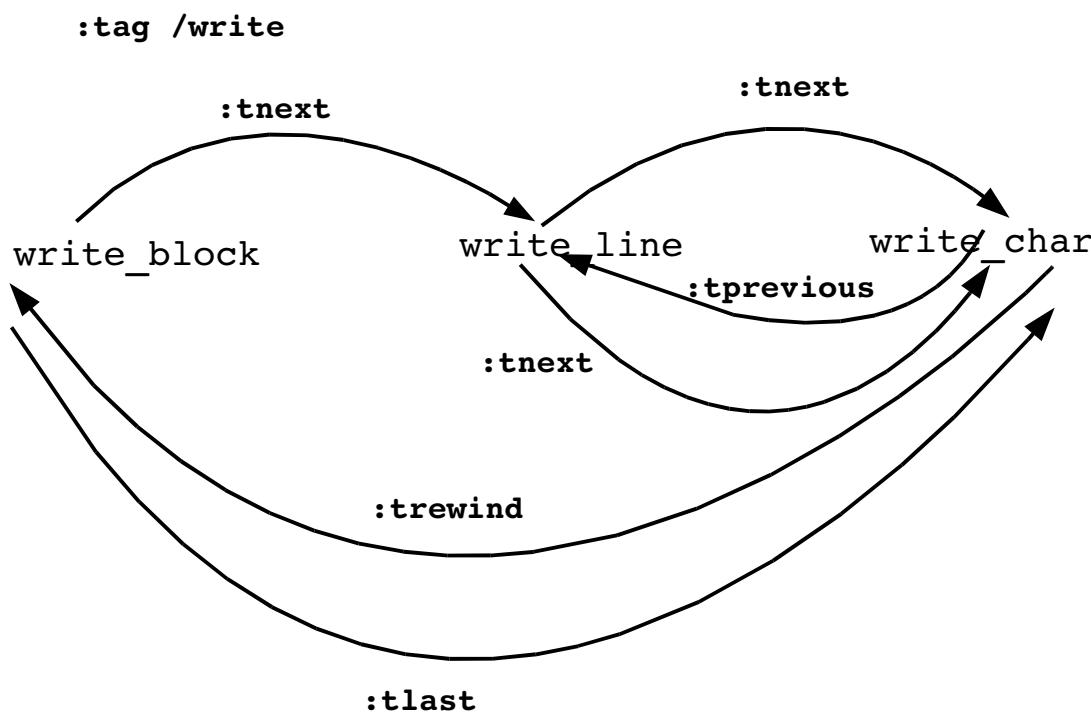


Figure 7-31: Tag navigation.

Shorthand Commands

The command `:stselect (:sts)` does the same thing as `:tselect`, except that it splits the window first. The `:stjump` does the same thing as a `:split` and a `:tjump`.

The Care and Feeding of Makefiles

The UNIX *make* command is designed to manage the compilation and building of programs. The commands to *make* are stored in a file called *Makefile*. The format of this file, although simple, is a little tricky to work with. In the following file, for example, the first command works, whereas the second contains an error:

```
alpha.o: alpha.c
        gcc -c alpha.c

beta.o: beta.c
        gcc -c beta.c
```

You may have a little difficulty seeing the problem from this listing. The problem is that the indent for the first command is a tab, whereas the indent on the second one uses eight spaces. This difference is impossible to see on screen; so how do you tell the difference between the two versions? The following command puts *Vim* in list mode:

```
:set list
```

In this mode, tabs show up as `^I`. Also the editor displays `$` at the end of each line (so you can check for trailing spaces). Therefore, if you use the following command

```
:set list
```

your example looks like this:

```
alpha.o: alpha.c$
^Igcc -c alpha.c$
$
beta.o: beta.c$
        gcc -c beta.c$
```

From this it is easy to see which line contains the spaces and which has a tab. (You can customize list mode by using the '`listchars`' ('`lcs`') option.)

The Vim Tutorial and Reference

If the '**expandtab**' option is set, when you type a tab, *Vim* inserts spaces. This is not good if you are editing a *Makefile*. To insert a real tab, no matter what the options are, type in **CTRL-V<Tab>** in insert mode. The **CTRL-V** tells *Vim* not to mess with the following character.

Note: If you have syntax coloring turned on, the *Vim* editor will highlight lines that begin with spaces in red, whereas lines that start with <Tab> display normally.

Sorting a List of Files

Frequently in a *Makefile*, you will see a list of files:

```
SOURCES = \
    time.cpp \
    set_ix.cpp \
    rio_io.cpp \
    arm.cpp \
    app.cpp \
    amem.cpp \
    als.cpp \
    aformat.cpp \
    adump.cpp \
    rio.cpp \
    progress.cpp \
    add.cpp \
    acp.cpp \
    rio_glob.cpp
```

To sort this list, execute the following:

1. Position the cursor on the start of the list.
2. Mark this location as a by using the command **ma**.
3. Go to the bottom of the list.
4. Run the block through the external program sort using the command
! 'a sort.

```
SOURCES = \
    acp.cpp \
    add.cpp \
    adump.cpp \
    aformat.cpp \
    als.cpp \
    amem.cpp \
    app.cpp \
    arm.cpp \
    progress.cpp \
    rio_glob.cpp \
    rio_io.cpp \
    rio.cpp \
    set_ix.cpp \
    time.cpp \

```

All the lines, except the last one, must end with a backslash (\). Sorting can disrupt this pattern. Make sure that the backslashes are in order after a sort. Figure 7-32 shows how you might need to fix the source list.

<pre>SOURCES = \ acp.cpp \ add.cpp \ adump.cpp \ aformat.cpp \ als.cpp \ amem.cpp \ app.cpp \ arm.cpp \ progress.cpp \ rio.cpp \ rio_glob.cpp \ rio_io.cpp \ set_ix.cpp \ time.cpp \ </pre>	<pre>SOURCES = \ acp.cpp \ add.cpp \ adump.cpp \ aformat.cpp \ als.cpp \ amem.cpp \ app.cpp \ arm.cpp \ progress.cpp \ rio.cpp \ Fix rio_glob.cpp \ Fix rio_io.cpp \ Fix set_ix.cpp \ Fix time.cpp \ </pre>
---	---

Figure 7-32: Fixing the source list.

Sorting a List in Visual Mode

To sort a list using visual mode, you need to execute the following commands:

1. Move to the top of the text to be sorted.
2. Start line visual mode with the command **v**.

3. Move to the bottom of the text to be sorted.
4. Execute the command **!sort**. The **!** tells *Vim* to pipe the selected text through a command. The command in this case is **sort**. (This command has an implied <Enter> at the end.)

Making the Program

The *Vim* editor has a set of commands it calls the quick-fix mode. These commands enable you to compile a program from within *Vim* and then go through the errors generated fixing them (hopefully). You can then recompile and fix any new errors that are found until finally your program compiles without error.

The :make command

The **:make** (**:mak**) command runs the program *make* (supplying it with any argument you give) and captures the results:

```
:make [arguments]
```

If errors were generated, they are captured and the editor positions you where the first error occurred.

Take a look at a typical **:make** session. (Typical **:make** sessions generate far more errors and fewer stupid ones.) Figure 7-33 shows the results. From this you can see that you have errors in two files, *main.c* and *sub.c*.

```
:!make | & tee /tmp/vim215953.err
gcc g Wall o prog main.c sub.c
main.c: In function `main':
main.c:6: too many arguments to function `do_sub'
main.c: At top level:
main.c:10: parse error before `}'
sub.c: In function `sub':
sub.c:3: `j' undeclared (first use in this function)
sub.c:3: (Each undeclared identifier is reported only once
sub.c:3: for each function it appears in.)
sub.c:4: parse error before `}'
sub.c:4: warning: control reaches end of non-void function
make: *** [prog] Error 1
2 returned
"main.c" 11L, 111C
(3 of 12): too many arguments to function `do_sub'
Press RETURN or enter command to continue
```

Figure 7-33: :make output.

The Vim Tutorial and Reference

When you press Enter (what *Vim* calls Return), you see the results shown in Figure 7-34.

```
int main()
{
    int i=3;
    do_sub("foo");
    ++i;
    return (0);
}
}
~
(3 of 12): too many arguments to function do_sub
```

Figure 7-34: The first error.

The editor has moved you to the first error. This is line 6 of *main.c*. You did not need to specify the file or the line number, *Vim* knew where to go automatically. The following command goes to where the next error occurs (see Figure 7-35):

```
:cnext
```

```
int main()
{
    int 1=3;
    do_sub("foo");
    ++i;
    return (0);
}
}
~
(5 of 12): parse error before `}'
```

Figure 7-35: :cnext.

Note: If you are a Visual-C++ user, the make program supplied by Microsoft is called *nmake*. You might need to customize *Vim* using the '**'makeprg'**' option so that it uses this program rather than the default make (as discussed on page 406).

The command **:cprevious** or **:cNext** goes to the previous error. Similarly, the command **:clast** goes to the last error and **:crewind** goes to the first. The **:cnfile** goes to first error message for the next file (see Figure 7-36).

The Vim Tutorial and Reference

```
int sub(int i)
{
    return (i * j)
}
~
~
~
~
~
~
(7 of 12): `j' undeclared (first use in this function)
```

Figure 7-36: **:cnfile** command

If you forget what the current error is, you can display it using the following command:

```
:cc
```

To see a list of errors, use the commands:

```
:copen
```

This command opens a new window showing all the errors as shown in Figure 7-37.

```
int sub(int i)
{
    return (i * j)
}
~
~
~
~
~
~
(7 of 12): `j' undeclared (first use in this function)
```

Figure 7-37: **:copen** command

To see a list of errors, execute the command:

```
:clist
```

Figure 7-38 shows the output of a **:clist** command.

The Vim Tutorial and Reference

```
~  
~  
:clist  
3 main.c:6: too many arguments to function `do_sub'  
5 main.c:10: parse error before `}'  
7 sub.c:3: `j' undeclared (first use in this function)  
8 sub.c:3: (Each undeclared identifier is reported only once  
9 sub.c:3: for each function it appears in.)  
10 sub.c:4: parse error before `}'  
11 sub.c:4: warning: control reaches end of non-void function  
Press RETURN or enter command to continue
```

Figure 7-38: **:clist** command

If you want to list only a subset of the errors, you can give **:clist** a range of errors to list. For example:

:clist 3,5 (List errors 3 through 5)

:clist ,5 (List errors 1-5)

:clist 5, (List errors 5 to the end)

The *Vim* editor suppresses all informational messages. If you want everything, use the following command:

:clist!

The override option (!) tells Vim to not suppress anything.

If you have already run make and generated your own error file, you can tell *Vim* about it by using the **:cfile error-file** command. Where **error-file** is the name of the output of the make command or compiler. If the **error-file** is not specified, the file specified by the '**errorfile**' option is used.

Finally the following command exits *Vim* like **:quit** but exits with an error status (exit code=1):

:cquit

This is useful if you are using *Vim* in an integrated development environment and a normal exit would cause a recompilation.

The 'errorfile' Option

The '**errorfile**' option (also '**ef**') defines the default filename used for the **:clist** command (**:c1**) as well as the **-q** command-line option. (This file is not used for the **:make** command's output.) If you want to define your default error file, use the following command:

```
:set errorfile=error.list
```

Searching for a Given String

The **:vimgrep** (**:vi**) command searches files for a given regular expression. To find all occurrences of the word **hack**, for example, you use this command:

```
:vimgrep /hack/ /usr/src/linux**/*.c
```

This command searches for the regular expression **hack** through all the files that match the wildcard ***.c** in the directory **/usr/src/linux** and all directories below it (**). Figure 7-39 shows the results.

```
/* A hack. Not moving message rate limiting to adev->xxx
 * (it's only a debug message after all) */
static int rate_limit = 0;

if (rate_limit++ < 3)
    log(L_IOCTL, "Please report in case toggling the power "
        "LED doesn't work for your card!\n");
if (enable)
    write_reg16(adev, IO_ACX_GPIO_OUT,
                read_reg16(adev, IO_ACX_GPIO_OUT) & ~gpio_pled);
(1 of 481): /* A hack. Not moving message rate limiting t 2677,7-14 63%
```

Figure 7-39: **:vimgrep** output.

Note: The **:vimgrep** command knows nothing about C syntax, so it will find **hack** even if it occurs inside a string or comment.

You can use the **:cnext** (**:cn**), **:cprevious** (**:cp**, **:cNext**, **:cN**), and **:cc** commands to page through the list of matches. Also **:crewind** (**:crew**, **:cf**), **:cfirst**, **:cf**) goes to the first match and **:clast** (**:cl**) to the last. Finally, the following command goes to the first match in the next file:

```
:cnfile
```

The **:copen** (**:cope**) command opens a new window containing all the items found as seen in Figure 7-40.

The Vim Tutorial and Reference

```
acxpci_l_power_led(acx_device_t *adev, int enable)
{
    u16 gpio_pled = IS_ACX111(adev) ? 0x0040 : 0x0800;

    /* A hack. Not moving message rate limiting to adev->xxx
     * (it's only a debug message after all) */
    static int rate_limit = 0;

    if (rate_limit++ < 3)

3rdparty/acx/pci.c [RO]                                2677,7-14      63%
3rdparty/acx/pci.c|2677 col 7| /* A hack. Not moving message rate limiting to
adev->xxx
3rdparty/at76c503a/at76_usb.c|5410 col 27| /* jal: this is a dirty hack needed by
Tim in ad-hoc mode */
3rdparty/at76c503a/at76_usb.c|5560 col 12| /* Magic hack for Novell IPX-in-802.3
packets */
3rdparty/ipw3945/ipw3945.c|11004 col 4| /* hack this function to show different
aspects of received frames,
3rdparty/ipw3945/ipw3945.c|11149 col 21| *      but you can hack it to show more, if
you'd like to. */
3rdparty/ndiswrapper/ntoskernel.c|1778 col 57| * related to ACPI: "_SM_" and
" DMI ". This may be the hack they do
[Quickfix List]                                         1,1           Top
:cc
```

Figure 7-40: Result of :copen

You can move up and down this list using the normal movement commands. When you a item you are interested in, you can press **<Enter>** and *Vim* will jump to that location.

***Vim* and outside edits**

When I'm programming Java, I like to use the *Eclipse* IDE. One nice feature of this program is that it does a good job of suggesting fixes for simple errors in your code.

Let's suppose you've written a Java program with *Vim* and wish to see if it compiles. You write it out using the **:w** command and load it in *Eclipse*. Being a typical programmer, the file needs some fixing, so you let *Eclipse* fix it. After saving the file, the one in the *Vim* editor is out of date.

Vim will detect this and ask if you wish to load the file. If you answer yes, it will read in the new file and let you continue editing. *Eclipse* will do the same thing, so the two editors (*Vim* and *Eclipse*'s internal editor) work well together as long as you remember to save the changes after each edit.

Vim also has a option: '**autoread**' ('**ar**') which will cause the editor to automatically read a file when a change is detected.

Other Interesting Commands

The *Vim* editor can use different options for different types of files through the use of the `:autocmd` command. See *Chapter 13: Autocommands* for more information. You can also customize your options on a per-file basis by putting something called a modeline in each file. The *Vim* editor scans your file looking for these lines and sets things up based on their content.

Chapter 8: Basic Abbreviations, Keyboard Mapping, and Initialization Files

The *Vim* editor has some features that enable you to automate repetitive tasks.

One of these is abbreviation, which enables you to type in part of a word and let *Vim* type the rest. Another is the ability to remap the keyboard. You can easily redefine a key to be a whole set of commands. After you design your customizations, you can save them to an initialization file that will automatically be read the next time you start *Vim*. This chapter discusses the most common and useful subset of these commands. For a more complete reference, see *Chapter 24: All About Abbreviations and Keyboard Mapping.*"

Abbreviations

An abbreviation is a short word that takes the place of a long one. For example, ad stands for advertisement. The *Vim* editor enables you to type in an abbreviation and then will automatically expand it for you. To tell *Vim* to expand the abbreviation ad into advertisement every time you type it, use the following command:

```
:abbreviate ad advertisement
```

(`:abbreviate` can be abbreviated as `:ab`.)

Now, when you type ad, the whole word advertisement will be inserted into the text.

What is Typed	Result
I saw the a	I saw the a
I saw the ad	I saw the ad
I saw the ad<space>	I saw the advertisement<space>

It is possible to define an abbreviation that results in multiple words. For example, to define JB as Jack Benny, use the following command:

```
:abbreviate JB Jack Benny
```

As a programmer, I use two rather unusual abbreviations:

```
:abbreviate #b /*****  
:abbreviate #e <space>***** /
```

The Vim Tutorial and Reference

These are used for creating boxed comments. The comment starts with `#b`, which draws the top line. I then put in the text and use `#e` to draw the bottom line. The number of stars (*) in the abbreviations is designed so that the right side is aligned to a tab stop. One other thing to notice is that the `#e` abbreviation begins with a space. In other words, the first two characters are space-star. Usually *Vim* ignores spaces between the abbreviation and the expansion. To avoid that problem, I spell space as seven characters: "<", "s", "p", "a", "c", "e", ">".

Listing Your Abbreviations

The command **:abbreviate (:ab)** lists all your current abbreviations. Figure 8-1 shows a typical execution of this command.

```
~  
~  
! #j          Jack Benny Show  
! #l          /*-----*/  
! #e          *****  
! #b          ****  
! #i          #include  
! #d          #define  
Press RETURN or enter command to continue █
```

Figure 8-1: **:abbreviate**.

Note: The abbreviation is not expanded until after you finish the word by typing a space, tab, or other whitespace. That is so that a word such as addition won't get expanded to advertisementdition.

Mapping

Mapping enables you to bind a set of *Vim* commands to a single key. Suppose, for example, that you need to surround certain words with curly braces. In other words, you need to change a word such as amount into {amount}. With the **:map** command, you can configure *Vim* so that the **F5** key does this job. The command is as follows:

```
:map <F5> i{<Esc>ea}<Esc>
```

Let's break this down:

<F5> The **F5** function key. This is the trigger key that causes the command to be executed as the key is pressed. (In this example, the trigger is a single key; it can be any string.)

i{<Esc>ea} Insert the { character. Note that we end with the **<Esc>** key.

The Vim Tutorial and Reference

E Move to the end of the word.

A}{Esc} Append the } to the word.

After you execute the **:map** command, all you have to do to put {} around a word is to put the cursor on the first character and press **F5**.

Note: When entering this command, you can enter **<F5>** by pressing the **F5** key or by entering the characters <, **F**, **5**, and >.

Either way works. However, you must enter **<Esc>** as characters. That is because the **<Esc>** key tells *Vim* to abort the command. Another way of entering an **<Esc>** key is to type **CTRL-V** followed by the **<Esc>** key. (The **CTRL-V** tells *Vim* to treat the **<Esc>** as a literal character instead of acting on it.)

Warning: The **:map** command can remap the *Vim* commands. If the trigger string is the same as a normal *Vim* command, the **:map** will supersede the command in *Vim*.

Listing Your Mappings

The **:map** command (with no arguments) lists out all your current mappings (see Figure 8-2).

```
~  
~  
~  
  
<F5>           i {<Esc>ea}<Esc>  
<xHome>        <Home>  
<xEnd>         <End>  
<S-xF4>        <S-F4>  
<S-xF3>        <S-F3>  
<S-xF2>        <S-F2>  
<S-xF1>        <S-F1>  
<xF4>          <F4>  
<xF3>          <F3>  
<xF2>          <F2>  
<xF1>          <F1>  
  
Press RETURN or enter command to continue █
```

Figure 8-2: :map command.

Fixing the Way Delete Works

On most terminals, the Backspace key acts like a backspace character and the Delete key sends a delete character. Some systems try to be helpful by remapping the keyboard and mapping the Backspace key to Delete. If you find that your keyboard has the Backspace and Delete keys backward, you can use the following command to swap them:

```
:fixdel
```

(`:fixdel` can be abbreviated as `:fix`)

It does this by modifying the internal *Vim* definitions for backspace (`t_kb`) and delete (`t_kD`). This command affects only the *Vim* keyboard mappings.

Your operating system may have its own keyboard mapping tables. For example, Linux users can change their keyboard mapping by using the *loadkeys* command. For further information, Linux users should check out the online documentation for *loadkeys*.

The X Window system also has a keyboard mapping table. If you want to change this table, you need to check out the *xmodmap* command. Check the X Window system documentation for details on how to use this command.

Controlling What the Backspace Key Does

The '`backspace`' option controls how the `<Backspace>` key works in insert mode. For example, the following command tells *Vim* to allow backspacing over autoindents:

```
:set backspace=indent
```

The following command enables you to backspace over the end of lines:

```
:set backspace=eol
```

In other words, with this option set, if you are positioned on the first column and press `<Backspace>`, the current line will be joined with the preceding one. The following command enables you to backspace over the start of an insert:

```
:set backspace=start
```

In other words, you can erase more text than you entered during a single insert command. You can combine these options, separated by commas. For example:

```
:set backspace=indent,eol,start
```

The Vim Tutorial and Reference

Earlier versions of *Vim* (5.4 and prior) use the following option values. These still work but are deprecated.

- 0 "" (No special backspace operations allowed)
- 1 "**indent, eol**"
- 2 "**indent, eol, start**"

Saving Your Setting

After performing all your **:map**, **:abbreviate**, and **:set** commands, it would be nice if you could save them and use them again. The command **:mkvimrc** (**:mkv**) writes all your settings to a file. The format of this command is as follows:

```
:mkvimrc file
```

Where *file* is the name of the file to which you want to write the settings. You can read this file by using the following command:

```
:source file
```

(**:source** can be abbreviated as **:so**.)

During startup, the *Vim* editor looks for an initialization file. If it is found, it is automatically executed. (Only the first file found is read.) The initialization files are as follows:

UNIX

```
$HOME/.vimrc  
$HOME/_vimrc  
$HOME/.exrc  
$HOME/_exrc
```

MS-DOS

```
$HOME/_vimrc  
$HOME/.vimrc  
$VIM/_vimrc  
$VIM/.vimrc
```

The Vim Tutorial and Reference

\$HOME/_exrc

\$HOME/.exrc

\$VIM/_exrc

\$VIM/.exrc

When you are running the GUI version, some other files are also read. The *gvimrc* file is found in the same location as the *vimrc* files mentioned in the list. The *\$VIMRUNTIME/menu.vim* is read too. One way you can find out which initialization files are read is to use the **:version (:ve)** command:

:version

In the middle of all the junk it lists out is a list of the initialization files (see Figure 8-3).

The Vim Tutorial and Reference

```
:version
:version
VIM - Vi IMproved 7.1 (2007 May 12, compiled Sep 7 2007 17:07:14)
Included patches: 1-87
Compiled by Anssi Hannula <anssi@mandriva.org>
Huge version with GTK2-GNOME GUI. Features included (+) or not (-):
+arabic +autocmd +balloon_eval +browse ++builtin_terms +byte_offset +cindent
+clientserver +clipboard +cmdline_compl +cmdline_hist +cmdline_info +comments
+cryptv +cscope
+cursorshape +dialog_con_gui +diff +digraphs +dnd -ebcdic +emacs_tags +eval
+tex_extra +extra_search +farsi +file_in_path +find_in_path +folding -footer
+fork() +gettext
-hangul_input +iconv +insert_expand +jumplist +keymap +langmap +libcall +linebreak
+lispindent +listcmds +localmap +menu +mksession +modify_fname +mouse +mouseshape
+mouse_dec
-mouse_gpm -mouse_jsbterm +mouse_netterm +mouse_xterm +multi_byte +multi_lang
-mzscheme +netbeans_intg -osfiletype +path_extra +perl +postscript +printer
+profile +python
+quickfix +reltime +rightleft -ruby +scrollbind +signs +smartindent -sniff
+statusline -sun_workshop +syntax +tag_binary +tag_old_static -tag_any_white +tcl
+terminfo
+termresponse +textobjects +title +toolbar +user_commands +vertsplit +virtualedit
+visual +visualextra +viminfo +vreplace +wildignore +wildmenu +windows
+writebackup +X11
-xfontset +xim +xsmp_interact +xterm_clipboard -xterm_save
    system vimrc file: "/etc/vim/vimrc"
        user vimrc file: "$HOME/.vimrc"
        user exrc file: "$HOME/.exrc"
    system gvimrc file: "/etc/vim/gvimrc"
        user gvimrc file: "$HOME/.gvimrc"
        system menu file: "$VIMRUNTIME/menu.vim"
    fall-back for $VIM: "/usr/share/vim"
Compilation: gcc -c -I. -Iproto -DHAVE_CONFIG_H -DFEAT_GUI_GTK
-I/usr/include/gtk-2.0 -I/usr/lib gtk-2.0/include -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/pan
go-1.0 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
-I/usr/include/freetype2 -I/usr/include/libpng12 -DORBIT2=1 -pthread
-I/usr/include/libgnomeui-2.0 -I/usr/include/l
ibart-2.0 -I/usr/include/gconf/2 -I/usr/include/gnome-keyring-1
-I/usr/include/libgnome-2.0 -I/usr/include/libbonoboui-2.0
-I/usr/include/libgnomecanvas-2.0 -I/usr/include/gtk-
2.0 -I/usr/include/gnome-vfs-2.0 -I/usr/lib/gnome-vfs-2.0/include
-I/usr/include/orbit-2.0 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
-I/usr/include/libbonobo-2.0 -I/u
sr/include/bonobo-activation-2.0 -I/usr/include/libxml2 -I/usr/include/pango-1.0
-I/usr/include/freetype2 -I/usr/include/gail-1.0 -I/usr/lib/gtk-2.0/include
-I/usr/include/atk-
1.0 -I/usr/include/cairo -I/usr/include/libpng12 -O2 -g -pipe
-Wp,-D_FORTIFY_SOURCE=2 -fstack-protector --param=ssp-buffer-size=4 -fexceptions
-fomit-frame-pointer -march=i
586 -mtune=generic -fasynchronous-unwind-tables -I/usr/local/include
-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 -I/usr/include/gdbm
-I/usr/lib/perl5/5.8.8/i386-linux/CORE
```

The Vim Tutorial and Reference

```
-I/usr/include/python2.5 -pthread -I/usr/include -D_REENTRANT=1 -D_THREAD_SAFE=1  
-D_LARGEFILE64_SOURCE=1  
Linking: gcc -Wl,-E -Wl,-rpath,/usr/lib/perl5/5.8.8/i386-linux/CORE  
-L/usr/local/lib -o vim -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0  
-lpangocairo-1.0 -lpang  
o-1.0 -lcairo -lgobject-2.0 -lgmodule-2.0 -lplib-2.0 -lgnomeui-2 -lbonoboui-2  
-lgnomevfs-2 -lgnomecanvas-2 -lgnome-2 -lpopt -lbonobo-2 -lbonobo-activation  
-lart_lgpl_2 -lgt  
k-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lpangocairo-1.0 -lpango-1.0  
-lcairo -lgconf-2 -lgmodule-2.0 -lORBit-2 -lgthread-2.0 -lrt -lgobject-2.0  
-lplib-2.0 -lxxt -lt  
ermcap -lacl -Wl,-E -Wl,-rpath,/usr/lib/perl5/5.8.8/i386-linux/CORE  
-L/usr/local/lib /usr/lib/perl5/5.8.8/i386-linux/auto/DynaLoader/DynaLoader.a  
-L/usr/lib/perl5/5.8.8/i386-1  
inux/CORE -lperl -lutil -lc -L/usr/lib/python2.5/config -lpython2.5 -lutil  
-Xlinker -export-dynamic -L/usr/lib -ltc18.5 -lieee -lm  
Press ENTER or type command to continue
```

Figure 8-3: Locating the initialization files with :version.

As part of the initialization process, *Vim* checks the value of the '**loadplugins**' ('**1pl**') option. If it is set, any available plug-ins are loaded. (A plug-in is a special script that's automatically loaded at initialization time. See the section *Plugins and other scripts* in *Chapter 27: Expressions and Functions* for more details.)

One other initialization file has not yet been discussed: *.exrc*. The old *Vi* editor used this file for initialization. This is only read if *Vim* cannot find any other initialization file. Because the old *Vi* program does not understand many of the *Vim* commands, you will probably want to put everything in the *.vimrc* file. The **:mkexrc** (**:mk**) command writes the mappings to the *.exrc* file. If you want to use all the power of *Vim*, however, you must use the **:mkvimrc** command instead.

My *.vimrc* File

My *.vimrc* file contains the following:

The Vim Tutorial and Reference

```
:syntax on
:autocmd FileType * set formatoptions=tcql
  \ nocindent comments&
:autocmd FileType c,cpp set formatoptions=croql
  \ cindent comments=sr:/*,mb:*,ex:*/,://
:set autoindent
:set autowrite
:ab #d #define
:ab #i #include
:ab #b /*****\
:ab #e <Space>*****
:ab #l /*----- */
:ab #j Jack Benny Show
:set shiftwidth=4
:set hlsearch
:set incsearch
:set textwidth=70
```

The file starts with a command to turn syntax coloring on:

```
:syntax on
```

The next thing is an autocmd executed every time a file type is determined (on file load). In this case, set the formatting options to **tcql**, which means autowrap text (**t**), autowrap comments (**c**), allow **qq** to format things (**q**), and do not break long lines in insert mode (**l**). I also turn off C-style indenting (**nocindent**) and set the '**comments**' option to the default (**comments&**):

```
:autocmd FileType * set formatoptions=tcql
  \ nocindent comments&
```

If a C or C++ file is loaded, the following autocmd is executed. It defines some additional format options, namely adding the comment header for new lines (**r**) and new lines opened with an O command (**o**). It also turns on C indentation and defines the '**comments**' option for C- and C++-style comments. Because this autocmd comes after the one for all files, it is executed second (but only for C and C++ files). Because it is executed second, its settings override any set by a previous autocmd:

```
:autocmd FileType c,cpp set formatoptions=croql
  \ cindent comments=sr:/*,mb:*,ex:*/,://
```

The next options turn on automatic indentation (indent each line the same as the preceding one) and autowriting (write files when needed). Note that because the autocmds execute when the file type is determined, any settings they have override these:

The Vim Tutorial and Reference

```
:set autoindent  
:set autowrite
```

What follows is a set of abbreviations useful to programmers and a collector of old Jack Benny radio shows:

```
:ab #d #define  
:ab #i #include  
:ab #b /*****  
:ab #e <Space>*****  
:ab #l /*-----*/  
:ab #j Jack Benny Show
```

The indentation size is set to 4, a value that studies have shown is best for programming:

```
:set shiftwidth=4
```

The next two options turn on fancy searching:

```
:set hlsearch  
:set incsearch
```

When working with text, I like a 70-column page:

```
:set textwidth=70
```

Script Files

To read a file containing Vim commands (and to execute the commands), use the **:source** (**:so**) command:

```
:source my-file.vim
```

A special version of this command is the **:runtime** (**:ru**) command. This searches the '**runtimepath**' ('**rtp**') for the file. If the file is not present it reports an error. (This is so you can find and load scripts that are part of the Vim runtime.)

To see what files have already been loaded, use the **:scriptnames** (**:scr**) command. Figure 8-4 shows the results.

The Vim Tutorial and Reference

```
1: /home/sdo/.vimrc
2: /usr/local/share/vim/vim71/syntax/syntax.vim
3: /usr/local/share/vim/vim71/syntax/synload.vim
4: /usr/local/share/vim/vim71/syntax/syncolor.vim
5: /usr/local/share/vim/vim71/filetype.vim
6: /home/sdo/.vim/plugin/getscriptPlugin.vim
7: /usr/local/share/vim/vim71/plugin/getscriptPlugin.vim
8: /usr/local/share/vim/vim71/plugin/gzip.vim
9: /usr/local/share/vim/vim71/plugin/matchparen.vim
10: /usr/local/share/vim/vim71/plugin/netrwPlugin.vim
11: /usr/local/share/vim/vim71/plugin/rrhelper.vim
12: /usr/local/share/vim/vim71/plugin/spellfile.vim
13: /usr/local/share/vim/vim71/plugin/tarPlugin.vim
14: /usr/local/share/vim/vim71/plugin/tohtml.vim
15: /usr/local/share/vim/vim71/plugin/vimballPlugin.vim
16: /usr/local/share/vim/vim71/plugin/zipPlugin.vim
Press ENTER or type command to continue
```

Figure 8-4: Output of :scriptnames

Scripts which deal with non-English languages are a little tricky to read. To help with the internationalization of *Vim*, the **:scriptencoding** (**:scripte**) command lets you tell *Vim* what encoding to use for a script.

For example, *Vim* comes with a script that provides Chinese menu translations, *\$VIMRUNTIME/langmenu_chinese_gb.936.vim*. The beginning of this script includes the line:

```
:scriptencoding cp936
```

To return the encoding to the default use the **:scriptencoding** command with no parameters:

```
:scriptencoding
```

Chapter 9: Basic Command-Mode Commands

The *Vim* editor is based on an older editor called *Vi*. The *Vi* editor was based on a command-line editor called *ex*. The *ex* editor was made before screen-oriented editors were popular. It was designed for the old printing terminals that were standard at that time.

Even though it was line oriented, the *ex* editor was an extremely versatile and efficient editor. It is still very useful today. Even with *Vim*'s tremendous command set, a few things are still better done with *ex*-style commands. So the people who created *Vim* give you access to all the *ex* commands through the use of command-line mode. Any command that begins with a colon(:) is considered an *ex*-style command. This chapter shows how *ex*-mode commands are structured and also discusses the most useful ones, including the following:

- Printing text lines
- Substitution
- Shell (command prompt) escapes

Entering Command-Line Mode

If you want to execute a single command-line-mode command, just type a colon(:) followed by the command. For example, the command `:set number` is actually a command-mode command. This command tells *Vim* to turn on the '`number`' option and display line numbers on the screen. (`:set` can be abbreviated as `:se` and '`number`' can be abbreviated as '`nu`'.)

A discussion of command-mode commands makes more sense with line numbering turned on. Therefore, the first command-mode command you enter for this chapter is as follows:

```
:set number
```

After this command has been executed, the editor returns to normal mode. Switch to command-line mode by executing the command `gq`. To switch back to normal mode (visual mode), use the `:visual (:vi)` command.

You will get a colon(:) prompt at the beginning of each line when you are in Ex mode. This can be turned off by turning off the '`prompt`' option. Also the `Q` command⁴ will also enter Ex mode, but will not give you command line editing capability.

⁴ Some distributions of Linux come with a system *vimrc* file which maps `Q` to `gq` (format text).

The Print Command

The **:print** command (short form **:p**) prints out the specified lines. Without arguments, it just prints the current line:

```
:print 1  
1 At one university the computer center was
```

Ranges

The **:print** command can be made to print a range of lines. A simple range can be something like **1,5**. This specifies lines **1** through **5**. To print these lines, use the following command:

```
:1,5 print  
1 At one university the computer center was  
2 experiencing trouble with a new type of  
3 terminal. Seems that the professors loved to  
4 put papers on top of the equipment, covering  
5 the ventilation holes. Many terminals broke
```

Strictly speaking, you do not have put a space between the **5** and the **print**, but it does make the example look nicer. If you want to print only line **5**, you can use this command:

```
:5 print  
5 the ventilation holes. Many terminals broke
```

You can use a number of special line numbers. For example, the line number **\$** is the last line in the file. So to print the whole file, use the following command:

```
:1,$ print  
1 At one university the computer center was  
...  
36 Notice:  
37  
38 If your computer catches fire, please turn it  
39 off and notify computing services.
```

The **%** range is shorthand for the entire file (**1,\$**). For example:

The Vim Tutorial and Reference

```
:1% print
1 At one university the computer center was
...
36 Notice:
37
38 If your computer catches fire, please turn it
39 off and notify computing services.
```

The line number dot (.) is the current line. For example:

```
.. print
39 off and notify computing services.
```

You can also specify lines by their content. The line number */pattern/* specifies the next line containing the pattern.

Let's move up to the top with :1 **print**, and then print the lines from the current line (.) to the first line containing the word trouble:

```
:1 print
1 At one university the computer center
was ..,/trouble/print
1 At one university the computer center was
2 experiencing trouble with a new type of
```

Similarly, **?pattern?** specifies the first previous line with pattern in it. In the following example, we first move to the end of the file with :39 **print** and then print the last line with the word Notice in it to the end of the file:

```
:39 print
39 off and notify computing services. :?
Notice:?,39 print
36 Notice:
37
38 If your computer catches fire, please turn it
39 off and notify computing services.
```

Marks

Marks can be placed with the normal-mode **m** command. For example, the **ma** command marks the current location with mark **a**.

You can use marks to specify a line for command-mode commands. The line number '**a**' specifies the line with mark **a** is to be used. Start in normal mode, for example, and move to the first line of the file. This is marked with a using the command **ma**. You then move to line 3 and use the command **mz** to mark line as **z**.

The Vim Tutorial and Reference

The command

```
:`a, `z print
```

is the same as the following command:

```
:1,3 print
```

Visual-Mode Range Specification

You can run a command-mode command on a visual selection. The first step is to enter visual mode and select the lines you want. Then enter the command-mode command to execute. Figure 9-1 shows that the first three lines of the text have been selected.

```
1 At one university the computer center was
2 experience trouble with a new type of computer
3 terminal. Seems that the professors loved to
4 put papers on top of the equipment, covering
5 the ventilation holes. Many terminals broke
6 down because they became so hot that the solder
-- VISUAL --
```

Figure 9-1: Visual-mode selection.

Next, enter the **:print** command to print these lines. The minute you press **:**, *Vim* goes to the bottom of the screen and displays the following:

```
:`<, `>
```

The special mark **<** is the top line of the visual selection and the mark **>** is the bottom. Thus, *Vim* is telling you that it will run the command on the visual selection. Because **<** is on line 1, and **>** is on line 3, a **:print** at this point prints lines 1 to 3. The full command, as it appears onscreen, looks like this:

```
:`<, `>print
```

Substitute Command

The **:substitute** command enables you to perform string replacements on a whole range of lines. The general form of this command is as follows:

```
:range substitute /from/to/ flags
```

(Spaces were added for readability.)

This command changes the from string to the to string. For example, you can change all occurrences of Professor to Teacher with the following command:

The Vim Tutorial and Reference

```
:% substitute /Professor/Teacher/
```

Note: The `:substitute` command is almost never spelled out completely. Most of the time, people use the abbreviated version `:s`. (The long version is used here for clarity.)

By default, the `:substitute` command changes only the first occurrence on each line. For example, the preceding command changes the line

```
Professor Smith criticized Professor Johnson today.
```

to

```
Teacher Smith criticized Professor Johnson today.
```

If you want to change every occurrence on the line, you need to add the `g` (global) flag. The command

```
:% substitute /Professor/Teacher/g
```

results in

```
Teacher Smith criticized Teacher Johnson today.
```

Other flags include `p` (print), which causes the `:substitute` command to print out each line it changes. The `c` (confirm) flag tells the `:substitute` to ask you for confirmation before it performs each substitution. When you enter the following

```
:1,$ substitute /Professor/Teacher/c
```

the *Vim* editor displays the text it is about to change and displays the following prompt:

```
Professor: You mean it's not supposed to do that?  
replace with Teacher (y/n/a/q/^E/^Y)?
```

At this point, you must enter one of the following answers:

- y** Make this replacement.
- n** Skip this replacement.
- a** Replace all remaining occurrences without confirmation.
- q** Quit. Don't make any more changes.
- CTRL-E** Scroll one line up.
- CTRL-Y** Scroll one line down.

How to Change Last, First to First, Last

Suppose you have a file containing a list of names in the form last, first, and you want to change it to first, last. How do you do it?

You can use the :substitute command to do it in one operation. The command you need is shown in Figure 9-2. The to string takes the first name (\2) and last name (\1) and puts them in order.

Note: This command uses regular expressions which are covered in detail in *Chapter 19: Advanced Searching Using Regular Expressions*.

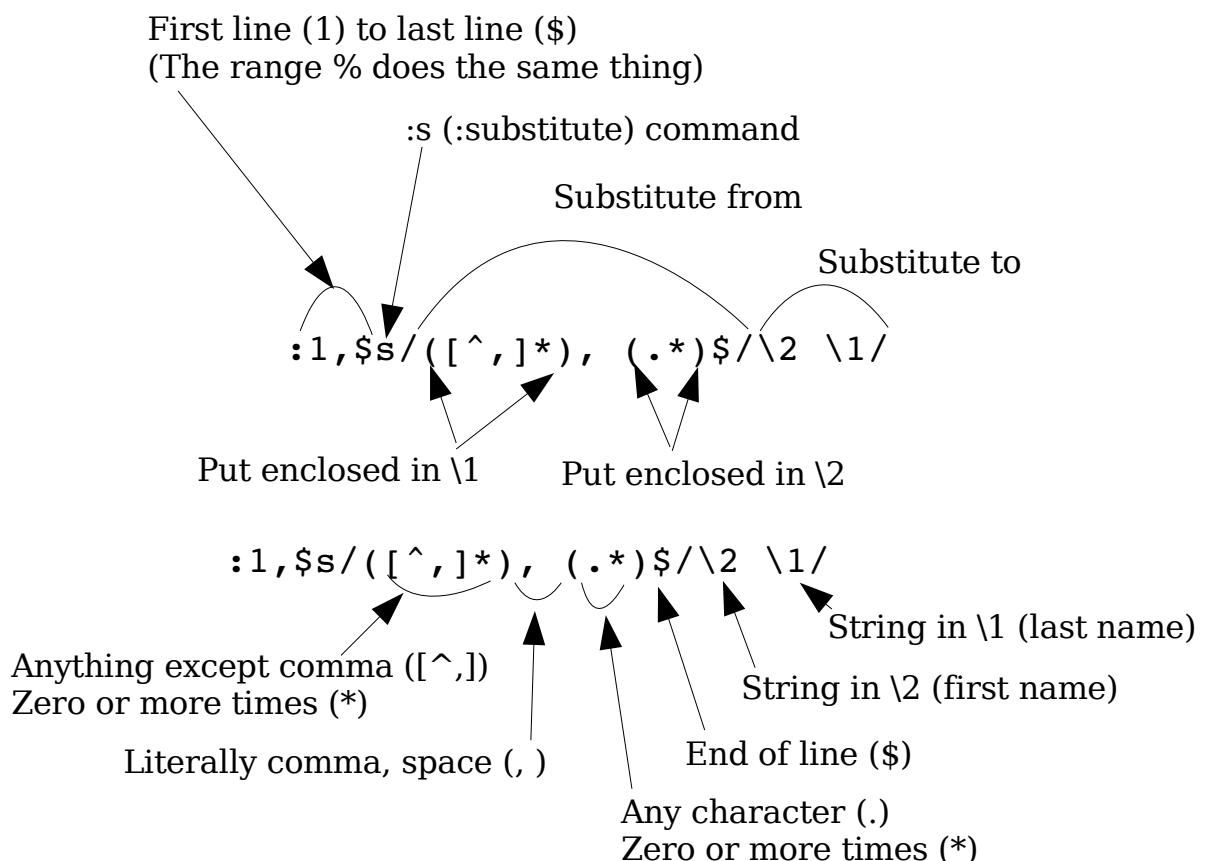


Figure 9-2: Changing last, first to first, last.

Reading and Writing Files

The `:read filename` command (short form `:r`) reads in a file and inserts it after the current line. The `:write` command (short form `:w`) writes out the file. This is a way of saving your work. You can write a different file (*prog.c.new*, for example) by giving `:write` a filename as an argument:

```
:write prog.c.new
```

Warning: If you exit using the emergency abort command `:q!`, the file reverts to the last written version.

The `:write` command usually does not overwrite an existing file. The force (!) option causes it to ignore this protection and to destroy any existing file. The `:write` command proves extremely useful when it comes to exporting portions of a large file to a smaller one--for example, if you have a collection of jokes and want to write one out to a file to send to a friend. To export a single joke, first highlight it in visual mode. Then use the following command to write it out to the file *joke.txt*:

```
: '<,'> write joke.txt
```

Saving the file under a new name

The `:saveas` (`:sav`) command saves the current file under a new name. Unlike the `:write` command, it also changes the name of the file being edited so future writes will send their output to the new file.

The `:shell` Command

The `:shell` command takes you to the command prompt. You can return to *Vim* by executing the exit command. For example:

```
:shell
$ date
Mon Jan 17 18:55:45 PST 2000
$ exit
-- vim window appears --
```

In this example, we are on UNIX, so we get the UNIX prompt (\$). If we were using the UNIX version of *gvim*, *Vim* would start a shell in the GUI window. On MS-DOS, the *Vim* command acts just like UNIX when a `:shell` command is executed. If you are using the GUI, however, the `:shell` command causes an MS-DOS prompt window to appear.

Printing the file

The `:[range]hardcopy (:ha)` command prints the file to a printer. On Linux and UNIX this sends the file to the default printer (unless you customize the command, see the section *Advanced Hardcopy* in *Chapter .*)

On Microsoft Windows, the command brings up a windows print dialog that lets you select the printer. If you wish to skip this dialog and print to the default printer, use the override (!) option:

```
:hardcopy!
```

The '`printoptions`' ('`popt`') option controls how the file is printed. The options available are:

<code>bottom: {size}</code>	bottom margin (default = <code>5pc</code>)
<code>collate: {y n}</code>	Turns on collation. (Default = <code>y</code>)
<code>duplex: {off long short}</code>	Sets duplex printing (default = <code>long</code>).
<code>formfeed: {y n}</code>	If set a formfeed character starts a new page. (Default = <code>n</code>).
<code>header: {count}</code>	Number of lines to reserve for the header. If this number is 0, no header is printed. If it is one or more, a line of text (defined by the ' <code>printhead</code> ' option) is printed, followed by any additional blank lines needed to make up the count.
<code>jobsplit: {y n}</code>	Do each copy as a separate print job. (Default = <code>n</code>)
<code>left: {size}</code>	Left margin (Default = <code>10pc</code>)
<code>number: {y n}</code>	Turn on or of line numbers. (Default = <code>n</code>)
<code>paper: {paper-size}</code>	Paper size. (Default = <code>A4</code>) The possible sizes are: <code>10x14, A3, A4, A5, B4, B5, executive, folio, ledger, legal, letter, quarto, statement, and tabloid</code> .
<code>portrait: {y n}</code>	If set to <code>y</code> to portrait printing. Otherwise do landscape. (Default = <code>y</code>)
<code>right: {size}</code>	Right margin. (Default = <code>5pc</code>)
<code>syntax: {y n}</code>	If <code>y</code> , turn on syntax highlighting. If <code>n</code> , turn it off. If <code>a</code> (default), turn it on if the printer appears to have the ability to do color or greyscale output.
<code>top: {size}</code>	Top margin. (Default = <code>5pc</code>)
<code>wrap: {y n}</code>	Wrap long lines. (Default= <code>y</code>)

The `{size}` parameter for the margins is of the form `{number}{unit}` where `{unit}` is :

`in` Inches

The Vim Tutorial and Reference

mm Millimeters

pc Percentage of the media

Chapter 10: Basic GUI Usage

The *Vim* editor works well inside a windowing environment. This graphical user interface (GUI) provides you with not only all of *Vim*'s keyboard commands, but also a number of menus and other options. This chapter shows you how to start *Vim* in GUI mode and how to make use of the special GUI features.

Starting Vim in GUI Mode

To start *Vim* in windowing mode, use the following command:

```
$ gvim file
```

This command starts up a *Vim* window and begins to edit file. The actual appearance of the screen depends on which operating system you are using. On UNIX it also depends on which X Window system toolkit (Motif, Athena, GTK) you have. Figure 10-1 and 10-2 show some of the various types of GUIs.

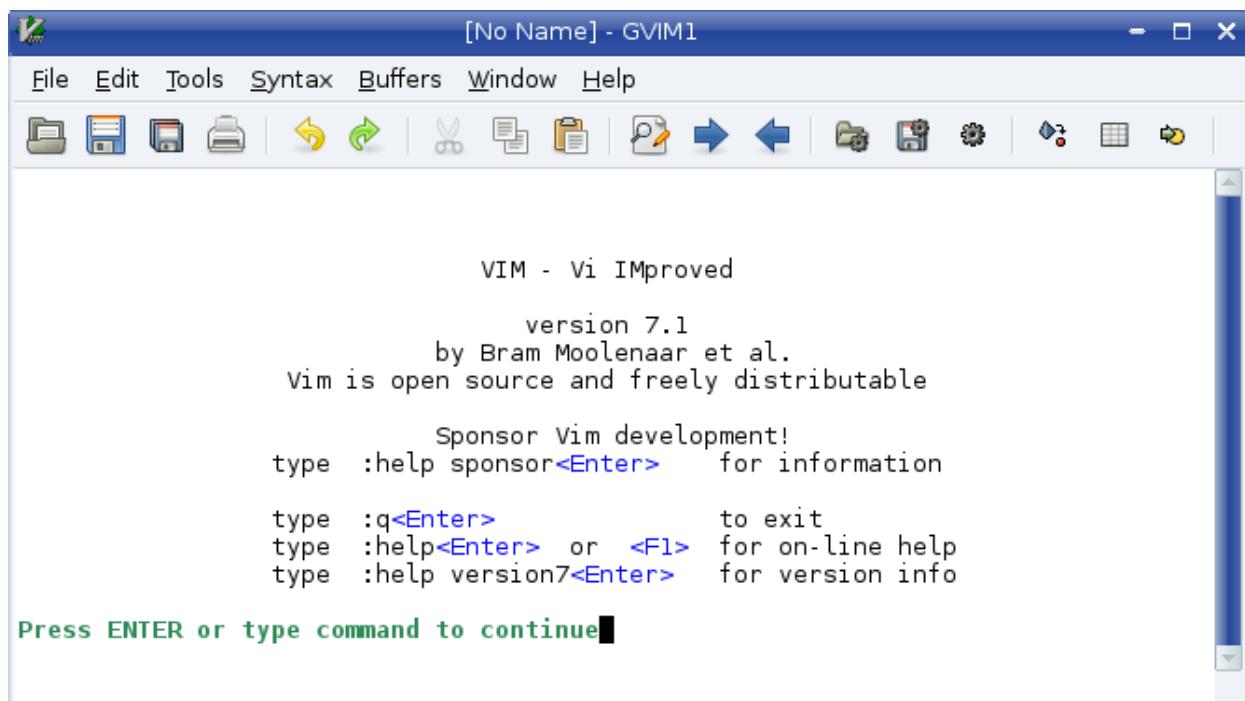


Figure 10-1: UNIX with the GTK toolkit.

The Vim Tutorial and Reference



Figure 10-2: Microsoft Windows.

If you have a choice of UNIX GUIs to choose from, it is recommended that you use the GTK version.

Mouse Usage

Standards are wonderful. In Microsoft Windows, you can use the mouse to select text in a standard manner. The X Window system also has a standard system for using the mouse. Unfortunately, these two standards are not the same. Fortunately, you can customize *Vim*. You can make the behavior of the mouse look like an X Window system mouse or a Microsoft Windows mouse. The following command makes the mouse behave like an X Window mouse:

```
:behave xterm
```

(**:behave** can be abbreviated as **:be**.)

The following command makes the mouse look like a Microsoft Windows mouse:

```
:behave mswin
```

The default behavior of the mouse on UNIX systems is *xterm*. The default behavior on a Microsoft Windows system is selected during the installation process. In addition to controlling the behavior of the mouse, the **:behave** command affects the following options:

The Vim Tutorial and Reference

Option	Setting For :behave xterm	Setting For :behave mswin
'selectmode'	mouse, key	(empty)
'mousemodel'	popup	extend
'keymodel'	startsel, stopsel	(empty)
'selection'	exclusive	inclusive

X Mouse Behavior

When **xterm** behavior is enabled, the mouse behavior is as follows:

- <LeftMouse>** Move the cursor.
- Drag with <Left Mouse>** Select text in visual mode.
- <RightMouse>** Extend select from cursor location to the location of the mouse.
- <MiddleMouse>** Paste selected text into the buffer at the mouse location.

Microsoft Windows Mouse Behavior

When **mswin** behavior is enabled, the mouse behavior is as follows:

- <LeftMouse>** Move the cursor.
- Drag with <LeftMouse>** Select text in select mode.
- <S-LeftMouse>** Extend selection to the cursor location.
- <RightMouse>** Display pop-up menu.
- <MiddleMouse>** Paste the text on the system Clipboard into file.

Special Mouse Usage

You can issue a number of other special commands for the mouse, including the following

- <S-LeftMouse>** Search forward for the next occurrence of the word under the cursor.
- <S-RightMouse>** Search backward for the preceding occurrence of the word under the cursor.

The Vim Tutorial and Reference

- | | |
|-----------------------------|---|
| <C-LeftMouse> | Jump to the tag whose name is under the cursor. |
| <C-RightMouse> | Jump to the preceding tag in the stack. |

Note: If you execute a command that requires a motion, such as **dmotion**, you can use the left mouse button for the motion.

Tear-Off Menus

The menus in *Vim* (all GUI versions except Athena) have an interesting feature: "tearoff" menus. If you select the first menu item (the dotted lines), you can drag the menu to another location on the screen. Figure 10-3 shows how to tear off a menu.

The Vim Tutorial and Reference

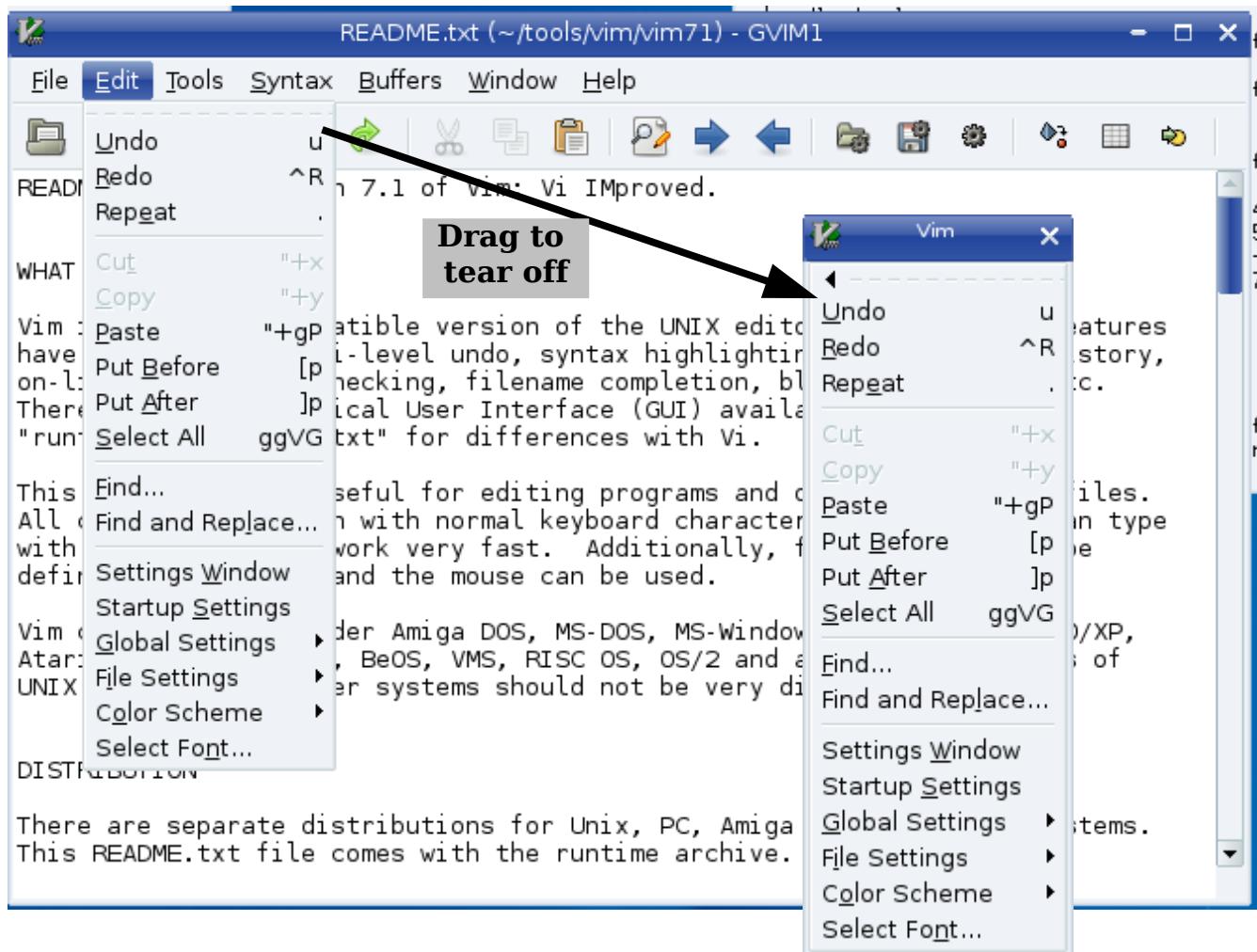


Figure 10-3: Tear-off menus.

When torn off, the menu remains as its own window until you close it using the normal window close command.

Toolbar

A toolbar appears in the GTK and MS-Windows versions of the GUI. It looks something like Figure 10-4. The icons perform the following functions:

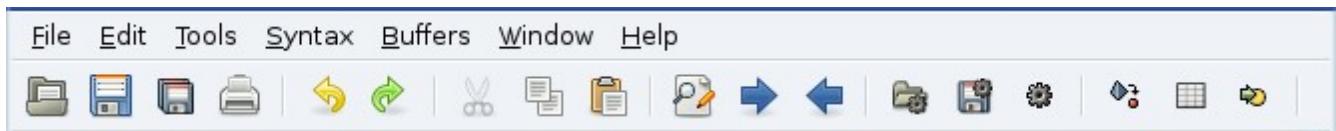


Figure 10-4: Toolbar.

-  Open. Brings up a File Open dialog box.
-  Save. Saves the current file.
-  Save All. Save all the open files in all windows and buffers.
-  Print. Print to system printer.
-  Undo.
-  Redo.
-  Cut. (Actually "delete.")
-  Copy. (Actually "yank.")
-  Paste.
-  Search. Brings up a dialog box so that you can enter a pattern.
-  Find Next.
-  Find Previous.
-  Replace. Brings up a Search-and-Replace dialog box.
-  Make Session. Brings up a dialog box so that you can enter the name of a session file to write to.
-  Load Session. Brings up a dialog box so that you can select the session file to load.
-  Script. Brings up a dialog box so that you can select a script to run.
-  Make. Performs a `:make`.
-  Shell. Does a `:shell`.
-  Make Tags. Does a `:!ctags -R .` command.
-  Tag. Jumps to the definition of the tag under the cursor.
-  Help. Brings up the general help screen.
-  Help Search. Brings up a dialog box so that you can enter a help topic to be displayed. This button is slightly misnamed because it does not do a general search of the help documents, but only looks for tags.

Showing the cursor

If you set the '`cursorcolumn`' ('`cuc`') option, Vim will highlight the column the cursor is in. Figure 10-5 shows the result:

The Vim Tutorial and Reference

```
Now is the time  
for all  
good men to come  
to the aide of their party.
```

*Figure 10-5: '**cursorcolumn**' option*

The '**cursorline**' ('**cul**') option highlights the line the cursor is on.
(Warning: This command uses a different highlighting than '**cursorcolumn**' and by default, does not highlight the background.)

Figure 10-6 shows an example:

```
Now is the time  
for all  
good men to come  
to the aide of their party.
```

*Figure 10-6: '**cursorcolumn**' option*

Chapter 11: Dealing with Text Files

Despite the proliferation of word processing tools such as Microsoft Word, OpenOffice, and such, people still use plain-text files for documentation because this type of file is the most easily read. In this chapter, you learn about the following:

- Automatic text wrapping
- Text formatting command
- Text formatting options
- Basic Spell Checking
- Dealing with different file formats
- Troff-related commands
- The rot13 algorithm

Automatic Text Wrapping

The *Vim* editor has a number of functions that make dealing with text easier. By default, the editor does not perform automatic line breaks. In other words, you have to press **<Enter>** yourself. This is extremely useful when you are writing programs where you want to decide where the line ends. It is not so good when you are creating documentation and do not want to have to worry about where to break the lines.

If you set the '**textwidth**' ('**tw**') option, *Vim* automatically inserts line breaks. Suppose, for example, that you want a very narrow column of only 30 characters. You need to execute the following command:

```
:set textwidth=30
```

Now you start typing (ruler added):

1	2	3
12345678901234567890123456789012345		
I taught programming for a while		

The word *while* makes the line longer than the 30-character limit. When *Vim* sees this, it inserts a line break and you get the following:

The Vim Tutorial and Reference

```
1           2           3  
12345678901234567890123456789012345  
I taught programming for a  
while
```

Continuing on, you can type in the rest of the paragraph:

```
1           2           3  
12345678901234567890123456789012345  
I taught programming for a  
while. One time, I was stopped  
by the Fort Worth police  
because my homework was too  
hard. True story.
```

You do not have to type newlines; *Vim* puts them in automatically. You can specify when to break the line in two different ways. The following option tells *Vim* to break the line 30 characters from the left side of the screen:

```
:set textwidth=30
```

If you use the following option, you tell *Vim* to break the lines so that you have margin characters from the right side of the screen.:.

```
:set wrapmargin=margin
```

('wm' is short for 'wrapmargin').

Therefore, if you have a screen that is 80 characters wide, the following commands do the same thing:

```
:set wrapmargin=10  
:set textwidth=70
```

Note: The '**textwidth**' option overrules '**wrapmargin**'.

The *Vim* editor is not a word processor. In a word processor, if you delete something at the beginning of the paragraph, the line breaks are reworked. In *Vim* they are not; so if you delete some words from the first line, all you get is a short line:

The Vim Tutorial and Reference

```
1          2          3  
12345678901234567890123456789012345  
I taught for a  
while. One time, I was stopped  
by the Fort Worth police  
because my homework was too  
hard. True story.
```

This does not look good; so how do you get the paragraph into shape? There are several ways. The first is to select the paragraph as part of a visual selection:

```
I taught for a  
while. One time, I was stopped  
by the Fort Worth police  
because my homework was too  
hard. True story.
```

Then you execute the **gq** command to format the paragraph.

```
I taught for a while. One  
time, I was stopped by the  
Fort Worth police because my  
homework was too hard. True  
story.
```

Another way to format a paragraph is to use the **gqmotion** command. Therefore to format 5 lines, you use the command **gq4j**. (The **4j** tells **gq** to format this line and the next 4 -- 5 lines total.)

The move forward paragraph command (**{**) also proves useful in such cases. To format a paragraph, for example, position the cursor on the first line of the paragraph and use the command **gq{}**. It is much easier to use this command than to count the lines.

The command **gqip** formats the current paragraph. (The **gq** formats the selected text and the **ip** selects the "inner paragraph.") This is easier than **gq{}** because you don't have to put the cursor on the beginning of a paragraph.

Finally, to format a line, use the **gqqq** command. You can shorten this to **gqq**.

Text Formatting Commands

To center a range of lines, use the following command:

The Vim Tutorial and Reference

```
:range center width
```

(**:ce** can be used for **:center**.)

If a width is not specified, it defaults to the value of '**textwidth**'. (If '**textwidth**' is 0, the default is 80.) For example:

```
:1,5 center 30
```

results in the following:

```
I taught for a while. One  
time, I was stopped by the  
Fort Worth police because my  
homework was too hard. True  
story.
```

Similarly, the command **:right** (**:ri**) right-justifies the text. So,

```
:1,5 right 30
```

results in the following:

```
I taught for a while. One  
time, I was stopped by the  
Fort Worth police because my  
homework was too hard. True  
story.
```

Finally there is the **:left** (**:le**) command:

```
:range left margin
```

Unlike **:center** and **:right**, however, the argument to **:left** is not the length of the line. Instead it is the left margin. If this is 0, the text will be put against the left side of the screen. If it is 5, the text will be indented 5 spaces. For example, these commands

```
:1 left 5  
:2,5 left 0
```

result in the following:

The Vim Tutorial and Reference

```
I taught for a while. One  
time, I was stopped by the  
Fort Worth police because my  
homework was too hard. True  
story.
```

Justifying Text

The *Vim* editor has no built-in way of justifying text. However, there is a neat macro package that does the job. To use this package, execute the following command:

```
:source $VIMRUNTIME/macros/justify.vim
```

This macro file defines a new visual command j. To justify a block of text, highlight the text in visual mode and then execute j.

Fine-Tuning the Formatting

A number of options enable you to fine-tune and customize your spaces.

The joinspaces Option

The **J** command joins two lines putting in one space to separate them. If the '**joinspaces**' option is set, when the first line ends with a punctuation mark (period, question mark, or exclamation point), two spaces are added. Input the following (= represents a space):

```
This=is=a=test.  
Second=line.
```

When the '**joinspaces**' option is turned off with the following command

```
:set nojoinspaces
```

the result of a **J** on the first line is as follows:

```
This=is=a=test.=Second=line .
```

If the option is set using this command

```
:set joinspaces
```

the result is as follows:

```
This=is=a=test.==Second=line .
```

The `formatoptions` Option

The option '**`formatoptions`**' ('**`fo`**') controls how *Vim* performs automatic wrapping. The *Vim* editor is smart about comments and does a proper job of formatting them. With '**`formatoptions`**' you can control both how text and comments are wrapped. The format of this option is as follows:

```
:set formatoptions=characters
```

where *characters* is a set of formatting flags. The following list identifies the formatting flags.

- t** Automatically wrap text.
- c** Automatically wrap comments.
- r** Insert the comment leader automatically.
- o** Insert comment leader in a comment when a new line is inserted.
- q** Insert comment leader in a comment when a new line is created using the **o** and **o** command.
- 2** Allow **gq** to format comments.
- v** Format based on the indent of the second line, not the first. Do old-style *Vi* text wrapping. Wrap only on blanks that you enter.
- b** Wrap only on blanks you type, but only if they occur before '**textwidth**'.
- 1** Do not break line in insert mode. Only let **gq** break the lines.

Take a look at how these flags affect the formatting.

The **t** flag must be on for normal text to be wrapped.

The **c** flag must be on for comments to be wrapped. Therefore, setting the '**`formatoptions`**' option using the following command is good for programming:

```
:set formatoptions=c
```

Long lines inside a comment are automatically wrapped. Long lines of code (*Vim* calls them text) are not wrapped. Actually you want to set this option:

```
:set formatoptions=cq
```

The Vim Tutorial and Reference

This tells *Vim* to not only wrap comments, but also to reformat comments as part of a **gq** command.

Vim is smart about comments. When it wraps a line in the middle of a C-style comment, it automatically adds the comment header in front of the line. Suppose, for example, that you enter the following command:

```
/* This is a test of a long line.
```

This line is longer than the '**textwidth**', so it wraps. Because it is in a comment, *Vim* automatically puts in an asterisk (*). Therefore, although you typed everything on one line, the result is as follows:

```
/* This is a test of a long  
 * line.
```

But suppose you actually type <Enter>? By default, *Vim* does not insert the asterisk. This means that if you type a two-line comment, you get the following:

```
/* Line 1  
Line 2
```

If you put an **r** flag in the '**formatoptions**', however, *Vim* automatically supplies the comment leader (*) when you press Return:

```
/* Line 1  
* Line 2
```

If you want to have this done for the **o** and **o** commands, you need to put in the **o** flag as well.

The **2** option tells *Vim* to format based on the second line of the text rather than the first. For example, the original example text is displayed in Figure 11-1. If you do not have the **2** flag in the '**formatoptions**' and you reformat the paragraph with **gq}**, you get the results shown in Figure 11-2.

```
The first Centronics Printer manual had a whole  
chapter devoted to how to open up the packing  
crate and find the manual. (What did they think we  
were reading anyway?)  
~  
~  
~  
~  
~
```

Figure 11-1: The original text.

The Vim Tutorial and Reference

```
The first Centronics Printer manual  
had a whole chapter devoted to how  
to open up the packing crate and  
find the manual. (What did they  
think we were reading anyway?)  
~  
~  
~  
~
```

*Figure 11-2: Formatted text (no **2** flag).*

If you go back to the original paragraph, however, set the **2** flag with the following

```
:set formatoptions += 2
```

and reformat using **gq}**, you will get the results shown in Figure 11-3.

```
The first Centronics Printer manual  
had a whole chapter devoted to how to  
open up the packing crate and find the  
manual. (What did they think we were  
reading anyway?)  
~  
~  
~  
~
```

*Figure 11-3: Formatted text (**2** set).*

The **v** flag character controls where a line will be split. Suppose that you have the following line:

```
This is a test of the very long line wrapping
```

Now add the word **logic** to the end of the sentence. Without the **v** flag, the result is as follows:

```
This is a test of the very  
long line wrapping logic.
```

With **v** in '**formatoptions**', you get the following:

The Vim Tutorial and Reference

```
This is a test of the very long line wrapping logic.
```

Even though the existing line is much longer than '**textwidth**', with **v** set, *Vim* will not break the line in the existing text. Instead it breaks only things in the text you add.

If the **l** character is present in the '**formatoptions**', *Vim* will break only the line if the space you type is less than the '**textwidth**'. If you add the word logic to the preceding example, you get the following:

```
This is a test of the very long line wrapping logic.
```

If you were to type this line from scratch, however, you would get the following:

```
This is a test of the very  
long line wrapping logic.
```

Formatting and numbered list

Vim will attempt to recognize numbered lists and properly format them. In order to tell what a number list looks like, it is matched against the string in the '**formatlistpat**' ('**f1p**') option. By default this is: `^\s*\d\+[\]\:\.)}\t]\s*` which translates to beginning of line (^), any number of spaces (`\s*`), one or more digits (`\d\+`), any one of the characters: `] :\.)}`, <tab> or <space>, followed by a bunch of spaces (`\s*`).

Using an External Formatting Program

By default, *Vim* uses its internal formatting logic to format the text. If you want, however, you can run an external program to do the job. On UNIX, the standard program *fmt* does a good job of doing the work. If you want to use this command for the **gq** work, set the following option:

```
:set formatprg=fmt
```

('fp' is short for '**formatprg**')

You can also use the '**formatexpr**' ('**fex**') option to define an expression which will tell *Vim* how to format a paragraph. If both '**formatexpr**' and '**formatprg**' are set, '**formatexpr**' will be used.

Even without this option, however, you can always use the filter (!) command to format text. To run a paragraph through the program *fmt*, for example, use the command !}fmt. The ! starts a filter command, the } tells *Vim* to filter a paragraph, and the rest (**fmt**) is the name of the command to use.

Basic Spelling

Vim can spell check your text on the fly. Words that are not spelled correctly will be highlighted. By default this feature is turned off, to turn it on, set the '**spell**' option:

```
:set spell
```

Figure 11-4 shows an editing session with lots of misspelled words and spell checking enabled. (In this example we use underline to indicate a misspelled word. The *Vim* GUI actually uses a red squiggly line which is impossible to reproduce in a black and white book.)

```
A church hhad just bought ttheir first  
computer and were learning how to uuse it.  
The church secretary decided to set uppp  
a form letter to be used in a funeral  
service. Where tthe person's name was to  
be she put in tthe word "<name>". When a  
funeral occurred she would change this  
word to tthe actual name of tthe departed.
```

Figure 11-4: Editing with spelling errors highlighted

Now that you can see which words are bad, it would be nice to get them corrected. To get a list of suggested fixes for the bad word, position the cursor on a misspelled word and enter the command **z=**. Figure 11-5 shows the result.

```
Change "hhad" to:  
1 "had"  
2 "head"  
3 "shad"  
4 "Had"  
5 "hand"  
6 "hard"  
7 "Head"  
8 "Chad"  
Type number (<Enter> cancels):
```

Figure 11-5: Result of the **z=** command

The Vim Tutorial and Reference

To change the word, just enter the number of the replacement (in this example 1). If you don't want to change anything, just press <Enter>.

Note: The change you make is stored in the . (dot) command so you can repeat it as needed.

If you are like me and you make the same spelling mistakes over and over again the same way, you'll want to fix all the spelling errors like one. (I must remember that "beleive" is spelled "believe".)

To do enter the :spellrepall (:spellr) command. It repeats the last z= change for all every similar word in your file.

Finding Spelling Errors

To find the next misspelled word use the js command. The js does the same thing, only it does not stop on rare words or words that come from another region.

Both the js and js commands start from the current cursor location for their searches. If you want to start from the beginning of the file use the [s and [s commands.

Spelling Language

Unfortunately everyone in the world does not speak the same language. If you need to change the language Vim uses for spell checking, set the 'spelllang' ('spl') option. This option takes a comma separated list of dictionaries to use for spell checking. If a spellfile for that language is not available, Vim will ask if you want to download it. For example:

```
:set spelllang=de
not found in 'runtimepath': "spell/de.latin1.spl"
not found in 'runtimepath': "spell/de.ascii.spl"
Cannot find spell file for "de" in
Do you want me to try downloading it?
(Y)es, [N]o:
```

Note: Automatic downloading is not enabled if you don't have the ability to write into the directory containing the spelling files (\$RUNTIME/spell).

The Limits of Automated Spell Checking

Can you spot the spelling error in the following word? Answer on page 191.

CAT

Word Lists

Vim lets you create your own word list to augment the built-in dictionary. To really use this feature you need to first set the '**spellfile**' ('**spf**') option. The name of this file must end in .add. For example:

```
:set spellfile=my_words.add
```

Now suppose you find that you have a perfectly good word in your document but it's not in the dictionary. To add it to your local word list, put the cursor on the word and enter the command **zg**. The word will be added to your local word list.

You may notice that two files are created: *my_words.add* and *my_words.add.spl*. The second file is a compiled version of the first. In fact *Vim* compiles all its dictionaries for speed. In this case *my_words.add.spl* is compiled automatically.

Vim not only keeps track of good words, but bad words as well. For example, if your file contains the words "ain't" and you consider that an abomination, then you can put in the bad word list by positioning the cursor on the word and entering the command **zw**.

As with **zg**, the word marked by **zw** goes into the spell file. However it is flagged as a badly spelled word.

If you accidentally put a bad word on the good list, the **zug** command removes the from the list. The **zuw** command does the same thing only it removes a word that flagged as wrong.

Vim also maintains a internal word list. This list is stored in memory and goes away after each editing session. To add a word to the internal good list, enter the command **zG**. For bad words, use the command **zw**.

And of course the **zuG** and **zuw** remove words from the internal list.

The spelling commands (**zg**, **zG**, **zw**, **zW**, **zug**, **zuG**, **zuw**, **zuW**) have their own command mode versions.

The Vim Tutorial and Reference

The following table shows these various commands

Command	Command Mode Version
<code>zg</code>	<code>:spellgood {word}</code> <code>:spellg {word}</code>
<code>zG</code>	<code>:spellgood! {word}</code> <code>:spellg! {word}</code>
<code>zw</code>	<code>:spellwrong {word}</code> <code>:spellw {word}</code>
<code>zW</code>	<code>:spellwrong! {word}</code> <code>:spellw! {word}</code>
<code>zug</code>	<code>:spellundo {word}</code> <code>:spellu {word}</code>
<code>zuG</code>	<code>:spellundo! {word}</code> <code>:spellu! {word}</code>

Multiple word lists

Vim allows you to use multiple word lists for spelling. The '**spellfile**' option actually takes a comma separated list of word files.

```
:set spellfile=global.add,local.add
```

Each of the word commands (**zg**, **zw**, **zug**, **zuG**, **:spellgood**, **:spellwrong**, **:spellundo**) takes a numeric argument. For example, the **zg** command adds the word under the cursor to the first word file. The command **2zg** will add it to the second file (*local.add*). The command **:2spellgood {word}** will do the same thing.

The **:spellinfo** (**:spell**) command lists the dictionary files being used.

```
:spellinfo  
file: /usr/local/share/vim/vim71/spell/en.latin1.spl  
  
Press ENTER or type command to continue
```

File Formats

Back in the early days, the old Teletype machines took two character times to do a newline. If you sent a character to the machine while it was moving the carriage back to the first position, it tried to print it on-the-fly, leaving a smudge in the middle of the page. The solution was to make the newline two characters: **<Return>** to move the carriage to column 1, and **<Line Feed>** to move the paper up.

The Vim Tutorial and Reference

When computers came out, storage was expensive. Some people decided that they did not need two characters for end-of-line. The UNIX people decided they could use <Line Feed> only for end-of-line. The Apple people standardized on <Return>. The MS-DOS (and Microsoft Windows) folks decided to keep the old <Return><Line Feed>.

This means that if you try to move a file from one system to another, you have line problems. The *Vim* editor automatically recognizes the different file formats and handles things properly behind your back.

Note: If you are an old *Vi* user and tried to edit an MS-DOS format file, you would have found that each line ended with a ^M character. (^M is <Return>.) Fortunately, *Vim* handles UNIX, MS-DOS and Apple file formats automatically.

The option '**fileformats**' ('**ffs**') contains the various formats that will be tried when a new file is edited. The following option, for example, tells *Vim* to try UNIX format first and MS-DOS format second:

```
:set fileformats=unix,dos
```

The detected file format is stored in the '**fileformat**' ('**ff**') option. To see which format you have, execute the following command:

```
:set fileformat?
```

You can use the '**fileformat**' option to convert from one file format to another. Suppose, for example, that you have an MS-DOS file named *readme.txt* that you want to convert to UNIX format. Start by editing the MS-DOS format file:

```
$ vim README.TXT
```

Now change the file format to UNIX:

```
:set fileformat=unix
```

When the file is written, it will be in UNIX format.

Changing How the Last Line Ends

The *Vim* editor assumes that your file is made up of lines. This means that *Vim* assumes that the last line in the file ends in an <EOL> character. Sometimes you will encounter a strange file that contains an incomplete line. When *Vim* encounters this type of file, it sets the '**noendofline**' ('**noeol**') option. (If your file ends in a complete line, the '**endofline**' ('**eol**') option is set.) If you want to change whether or not your file ends in an <EOL>, use the command

```
:set endofline
```

(Last line ends in <EOL>.) or

```
:set noendofline
```

(Last line does not have an <EOL>.) This option only works when the '**binary**' option is set.

Troff-Related Movement

A number of commands enable you to move through text. The **)** command moves forward one sentence. The **(** command does the same thing backward.

The **}** command moves forward one paragraph, and **{** moves one paragraph backward. Figure 11-6 shows how these commands work.

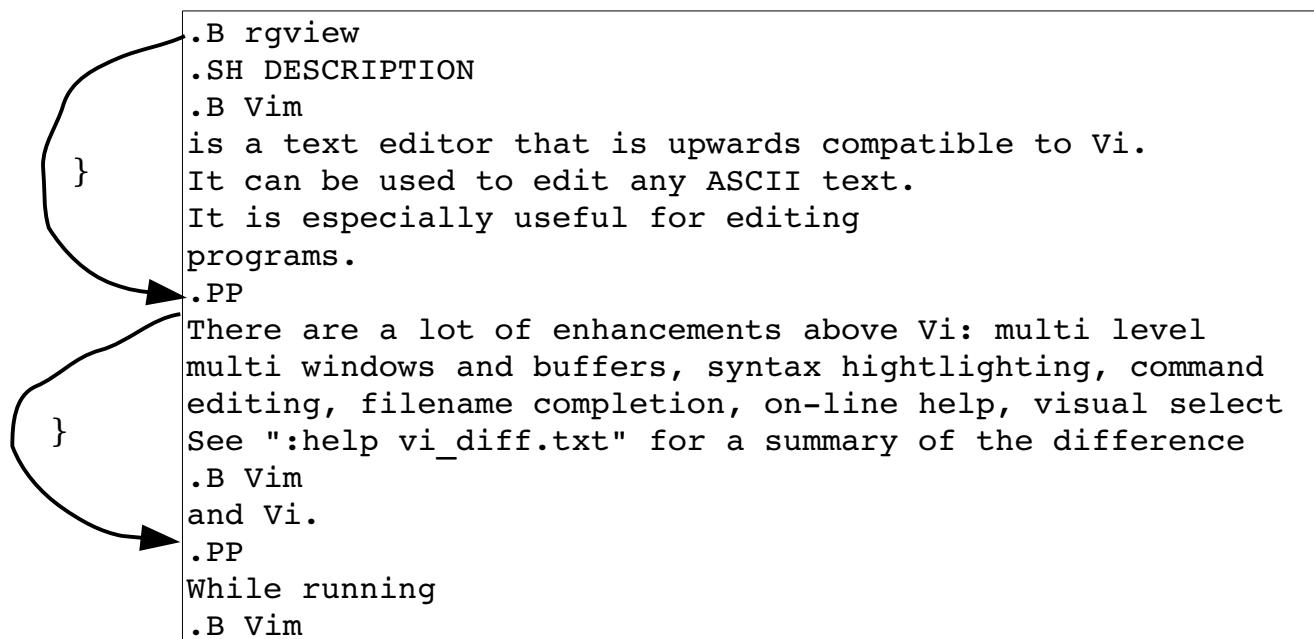


Figure 11-6: **}** command.

At one time the *troff* program was the standard UNIX word processor. It takes as input a text file with processing directives in it and formats the text. Although *troff* is rarely used these days, the *Vim* editor still contains an option for dealing with this formatter.

The *troff* program uses macros to tell it what to do. Some of these macros start paragraphs. In the following example, for instance, the macro **.LP** starts each paragraph:

The Vim Tutorial and Reference

Because *troff* uses lots of different macro packages, *Vim* needs to know which macros start a paragraph. The '**paragraphs**' ('**para**') option does this. The format of this option is as follows:

```
:set paragraphs="macromacromacro..."
```

Each macro is the two-character name of a *troff* macro. For example:

```
:set paragraphs="P<Space>LP"
```

tells *Vim* that the macros **.P** and **.LP** start a paragraph. (Note that you use **P<Space>** to indicate the **.P** macro.)

By default, the '**paragraphs**' option is as follows:

```
:set paragraphs=IPLPPPQPPP LIPplpipbp
```

This means that the following macros

.IP	.LP	.pp	.lp
.PP	.ip	.QP	.bp
.P	.LI		

start a new paragraph.

Section Moving

The **[[** and **]]** commands move a section backward. A section is defined by any text separated by a page break character (CTRL-L). The reason there are two movement commands is that these commands also move to the beginning and end of procedures. (*Chapter 7: Commands for Programmers* contains information on programming commands.)

The **] [** and **]]** commands perform the forward movements as seen in Figure 11-7.



```
This is page 1
^L
This is page 2
Another paragraph
^L
This is page 3
with a two line paragaph
And another one
~
~
~
~
~
```

Figure 11-7: The J] command.

Defining Sections

You can also define a section using troff macros. The '**sections**' ('**sect**') option acts much like the '**paragraph**' option, except that it defines the macros that separate sections rather than paragraphs. The default is:

```
:set sections=SHNHH HUNhsh
```

Encrypting with rot13

If you want to encrypt a block of text with the rot13 algorithm, use the **g? motion** command. The rot13 encryption is an extremely weak encryption scheme designed to obscure text. It is frequently used in news posting for potentially offensive material. Naturally **g?g?** or **g??** encrypts the current line. You can decrypt the rot13 encryption by encrypting the text twice.

Answer to the question on question on page 186.

The word is “dog”.

Chapter 12: Automatic Completion

The *Vim* editor can automatically complete words on insertion. This is where you type the first part of a word, press **CTRL-P**, and *Vim* guesses at the rest. How it decides what to use for completion is both simple and flexible. This chapter covers every aspect of this function. This chapter discusses the following:

- Automatic completion
- How to customize the automatic completion feature
- How to use different types of completions

Automatic Completion

When you are entering text, *Vim* can assist you to complete words. Suppose, for example, that you are creating a C program and want to type in the following:

```
total = ch_array[0] + ch_array[1] + ch_array[2];
```

You start by entering the following:

```
total = ch_array[0] + ch_
```

At this point, you tell *Vim* to complete the word using the command **CTRL-P**. This command tells *Vim* to search for a word to complete. In this case, it is the word `ch_array`. So typing **CTRL-P** gives you the following:

```
total = ch_array[0] + ch_array
```

After a little more typing, you get this:

```
total = ch_array[0] + ch_array[1] +
```

If you now type **CTRL-P** again, *Vim* will search again for a word that completes the word before the cursor. (In this case, it is ""). The first word that matches is `ch_array`.

Typing **CTRL-P** again gives you the next word that matches (in this case, `total`).

A third **CTRL-P** causes the editor to run out of words, so it returns to the original entry: "".

A fourth **CTRL-P** causes the editor to start over again with `ch_array`.

How Vim Searches for Words

The *Vim* editor goes through a lot of effort to find words to complete. By default, it uses the following search algorithm:

1. Current file
2. Files in other windows
3. Other files in the loaded (hidden) buffers
4. Files in unloaded buffers
5. The current "tags" list
6. All files #included by the current file

You can customize the search order. (This is described in the section *Automatic Completion Details* below.)

Searching Forward

When you type **CTRL-P**, *Vim* searches backward for a word to complete. The **CTRL-N** command searches forward. Other than the direction, it acts just like the **CTRL-P** command.

Automatic Completion Details

The '**ignorecase**' ('**ic**') option tells the editor to try all words regardless of case:

```
:set ignorecase
```

Therefore, if you have the option on, when you try to complete the string `ins`, *Vim* will consider `INSERT`, `Inside`, and `instep`, all as candidates (assuming that they appear in the text).

When you match words in this way, the completed word takes on the case of the matched word. Therefore, the completion list for `ins` is as follows: `insteP Inside, INSERT`.

What if you want to match `INSERT`, but put the word `insert` (lowercase) in your text. You must tell *Vim* to infer the case of the new word from what is typed, not what is matched. You can set the option '**infercase**' ('**inf**'),

```
:set infercase
```

The Vim Tutorial and Reference

Then the match list is as follows: `instep inside insert`

Automatic Completion Details

For the most part, the *Vim* editor does the right thing when it comes to automatic completion. At times, however, you might want to fine-tune your completions. The '**complete**' ('**cpt**') option controls where *Vim* searches for words. The form of this option is as follows:

```
:set complete=key,key,key,...
```

key is a key letter (and possible argument). The possible key values are as follows:

- . Current file.
- b** Files in loaded buffers, not in a window.
- d** Definitions in the current file and in files included by a #include directive.
- i** Files included by the current file through the use of a #include directive.
- k** The file defined by the 'dictionary' option (discussed later in this chapter).
- kfile** The file named *file*.
- t** The "tags" file. (The **] character can be used as well.)**
- u** Unloaded buffers.
- f** Files in other windows.

The Include Path

Vim uses the '**path**' ('**pa**') option to tell it where to look for files that were included in the current file. (Note that the '**path**' option also is used for other commands such as :**find**.)

Specifying a Dictionary

The '**dictionary**' ('**dict**') option defines a file to be searched when you press **CTRL-P** and **CTRL-N** to match words. The format of this command is:

```
:set dictionary=file,file,...
```

The Vim Tutorial and Reference

To use on Linux, for example, the dictionary file is in `/usr/dict/words`. Therefore, to add this file to the list of dictionaries searched for, use the following command:

```
:set dictionary=/usr/dict/words
```

If you have a local list of words, you can search this too:

```
:set dictionary=/home/oualline/words,/usr/doc/words
```

You can also specify a dictionary by putting the file after the **k** (key). For example:

```
:set complete=k/usr/oualline/words
```

You can use the **k** flag multiple times, each with a different file:

```
:set complete=k/usr/dict/words,k/usr/share/words
```

A dictionary file is simply a list of words to check for **CTRL-X** **CTRL-K**. The one restriction is that no line can be longer than 510 characters.

The words used for a thesaurus search are specified by '**thesaurus**' ('**tsr**') option are used. The files are a little different than the dictionary files. Each line contains a bunch of words which have similar meanings. When *Vim* finds a word in the thesaurus it then lists all the similar words as possible completion candidates.

Controlling What Is Searched For

CTRL-P and **CTRL-N** enable you to perform a wide variety of searches. What if you want to restrict yourself to just one type of search, however? For that you use the **CTRL-X** command. When you type **CTRL-X**, you enter the *CTRL-X submode*. You can then fine tune your search using one of the commands:

CTRL-]	Tags
CTRL-D	Macro definitions
CTRL-F	Filenames
CTRL-K	Dictionary
CTRL-I	Current files and #included files
CTRL-L	Whole lines
CTRL-N	Same as CTRL-N without the CTRL-X (find next match)
CTRL-O	Use omni completion by calling the function name in the ' omnifun ' option.

The Vim Tutorial and Reference

CTRL-P	Same as CTRL-P without the CTRL-X (find previous match)
CTRL-T	Thesaurus
CTRL-U	Perform user defined completing by calling the function defined by the ' completfunc ' option.
s	Use the spelling system to find the given word.

The **CTRL-X** **CTRL-D** command searches for a `#define` macro definition. It will search included files as well. After typing this command, you can type **CTRL-N** to search for the next definition and **CTRL-P** for the previous. Take a look at how this works on the following test file.

File *include.h*

```
#define MAX(x, y) ((x) < (y) ? (y) : (x))
#define MIN(x, y) ((x) < (y) ? (x) : (y))
int sum(int i1, int i2) {
    return(i1+i2);
}
```

File *main.c*

```
#include "include.h"
#define MORE "/usr/ucb/more"
```

You can start by editing *main.c*. If you type **CTRL-X**, you enter **CTRL-X** mode. The editor now displays a mini-prompt at the bottom of the screen (see Figure 12-1).

```
#include "include.h"
#define MORE "/usr/ucb/more"
~
~
~
-- ^X mode (^] ^D ^E ^F ^I ^K ^L ^N ^O ^P s ^U ^V ^Y)
```

Figure 12-1: **CTRL-X** mode.

Suppose that you want to look for a macro definition. You would now type **CTRL-D**. The screen displays the fact that there are three matches, and then displays a new menu (see Figure 12-2).

The Vim Tutorial and Reference

```
#include "include.h"
#define MORE "/usr/ucb/more"
MAX
MAX include.h
MIN include.h
MORE
~
~
~
-- Definition completion (^D/^N/^P) --
```

Figure 12-2: **CTRL-X CTRL-D**.

At this point, **CTRL-N** searches for the next match (and **CTRL-P** searches for the previous one). The **CTRL-D** key acts just like **CTRL-P**. Using these keys, you can cycle through the list of definitions until you find the one that you want.

Tag Search

The **CTRL-X CTRL-]** command searches for the next tag. A tag is a C/C++ function definition. The program *ctags* generates a list of C/C++ function definitions (tags) and stores them in the tags file. We have generated our tags file using the following command:

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

Now when we enter **CTRL-X CTRL-]** in insert mode, we get what is shown in Figure 12-3

The Vim Tutorial and Reference

```
#include "include.h"
#define MORE "/usr/ucb/more"
MORE
MORE
!_TAG_FILE_FORMAT
!_TAG_FILE_SORTED
!_TAG_PROGRAM_AUTHOR
!_TAG_PROGRAM_NAME
!_TAG_PROGRAM_URL
!_TAG_PROGRAM_VERSION
MAX
MIN
sum

~
~
~

-- Tag completion (^]/^N/^P) --
```

Figure 12-3: **CTRL-X CTRL-J**.

The result of typing **CTRL-P** a couple of times is shown in Figure 12-4.

```
#include "include.h"
#define MORE "/usr/ucb/more"
MAX
MORE
!_TAG_FILE_FORMAT
!_TAG_FILE_SORTED
!_TAG_PROGRAM_AUTHOR
!_TAG_PROGRAM_NAME
!_TAG_PROGRAM_URL
!_TAG_PROGRAM_VERSION
MAX
MIN
sum

~
~
~

-- Tag completion (^]/^N/^P) --
```

Figure 12-4: Finding previous completions.

By default, the *Vim* editor just displays the name alone. If you execute the following command, the entire tag (the function prototype) displays:

The Vim Tutorial and Reference

```
:set showfulltag
```

('**sft**' is the short form of '**showfulltag**')

If you repeat your matching commands with this option enabled, you get the results shown in Figure 12-5.

```
#include "include.h"
#define MORE "/usr/ucb/more"
int sum(int i1, int i2)
MORE
MORE
!_TAG_FILE_FORMAT
xtended format; --format=1 will not append ;" to lines/
!_TAG_FILE_SORTED
=unsorted, 1=sorted, 2=foldcase/
!_TAG_PROGRAM_AUTHOR
hiebert@users.sourceforge.net/
!_TAG_PROGRAM_NAME
!_TAG_PROGRAM_URL
fficial site/
!_TAG_PROGRAM_VERSION
MAX
MIN
sum
int sum(int i1, int i2) {
~
~
~
-- Tag completion (^]/^N/^P)--
```

Figure 12-5: The results.

@@ TODO: Investigate why the junk in the previous screen.

Finding Filenames

If you use the **CTRL-X CTRL-F** command, the *Vim* editor will match a filename. It scans the current directory for files and displays each one that matches the word in front of the cursor. Suppose, for example, that you have the following files in the current directory:

main.c *sub_count.c* *sub_done.c* *sub_exit.c*

Now enter insert mode and start typing:

The Vim Tutorial and Reference

```
The exit code is in the file sub
```

At this point, you enter the command **CTRL-X CTRL-F**. The editor now completes the current word `sub` by looking at the files in the current directory. The first match is `sub_count.c`. This is not the one you want, so you match the next file by typing **CTRL-N**. This match is `sub_done.c`. Typing **CTRL-N** again takes you to `sub_exit.c`.

The result:

```
The exit code is in the file sub_exit.c
```

Line Mode

All the commands discussed so far work on words only. The **CTRL-X CTRL-L** command works on lines. If you enter **CTRL-X CTRL-L** in the example, for instance, you get the results shown in Figure 12-6. You can now use **CTRL-N** and **CTRL-P** to go up and down lines.

```
#include "include.h"  
#define MORE "/usr/ucb/more"  
#define MORE "/usr/ucb/more"  
#include "include.h"  
#define MORE "/usr/ucb/more" ~  
~  
-- Whole line completion (^L/^N/^P)--
```

Figure 12-6: **CTRL-X CTRL-L**.

Dictionary and Thesaurus

The command **CTRL-X CTRL-K** searches the dictionary files for the given word.

The **CTRL-X CTRL-T** command does the same thing except that the word files specified by '`thesaurus`' ('`tsr`') option are used.

Guessing

The **CTRL-X CTRL-V** command guesses what type of word is before the cursor and performs the approbate completion.

@@ How does it guess.

User and Omni completion

The **CTRL-X CTRL-U** command calls the function defined by the '**completerfunc**' ('**cfu**') option. The command **CTRL-X CTRL-O** does the same thing only the function specified by the '**omnifunc**' ('**ofu**') is called.

Writing a completing function is a advanced specialized task that's beyond the scope of the book.

@@ Maybe add this topic to the book

Adjusting the Screen

There are two more **CTRL-X** commands which surprisingly don't do completion. After typing **CTRL-X**, you can move the screen up and down a little. The **CTRL-Y** command scrolls down, whereas the **CTRL-E** scrolls up.

Chapter 13: Autocommands

One of the nice attributes of the *Vim* editor is its flexibility. One of the features that makes it so flexible is the autocommand. An autocommand is a command executed automatically in response to some event, such as a file being read or written or a buffer change. Through the use of autocommands, for example, you can train *Vim* to edit compressed files. (You define an autocommand that uncompresses the file on read and another one to compress the file on write. See the file `$VIMRUNTIME/vimrc_example.vim` in your *Vim* installation.)

In this chapter, you learn about the following:

- Basic autocommands
- Autocommand groups
- Listing and removing autocommands

Basic Autocommands

Suppose you want to put a date stamp on the end of a file every time it is written. One way you could do this is to define a function:

```
:function DateInsert()  
:    $read !date " Insert the date at the  
:                " end ($) of the file.  
:endfunction
```

Now when you want to write the file all you have to do is to call this function:

```
:call DateInsert()
```

Then you write the file.

That may be a little difficult, so you can map this to a key:

```
:map <F12> :call DateInsert()<CR> \| :write<CR>
```

This makes things "easy" because now you just have to press `<F12>` every time you want to write the file.

If you forget, however, and use the normal *Vim* file writing commands, you screw things up. It would be nice if you could do things automatically. That is where autocommands come in.

The Vim Tutorial and Reference

The command

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

causes the command `:call DateInsert()` to be executed for all files (*) just before writing the file (`FileWritePre`).

(`:autocmd` can be abbreviated as `:au`.)

You do not need to put in a `:write` command because this autocmd is executed just before each `:write`. In other words, with this command enabled, when you do a `:write`, Vim checks for any `FileWritePre` autocmds and executes them, and then it performs the `:write`. The general form of the `:autocmd` command is as follows:

```
:autocmd group events file_pattern nested command
```

The *group* name is optional. It is used in managing and calling the commands (more on this later). The *events* parameter is a list of events (comma separated) that trigger the command. (A complete list of events appears later in this chapter.) The *file_pattern* is a filename (including wildcards). The nested flag allows for nesting of autocmds, and finally, the *command* is the command to be executed.

Groups

The `:augroup` (`:aug`) command starts the definition of a group of autocmds. The group name is just a convenient way to refer to the set of autocmds. For example:

```
:augroup cprograms
:    autocmd FileReadPost *.c :set cindent
:    autocmd FileReadPost *.cpp :set cindent
:augroup END
```

Because the `:autocmd` definitions are inside the scope `:augroup`, they are put in the *cprograms* group. The commands in this group are executed after reading a file that ends in *.c* or *.cpp*. If you want to add another command to this group for headers, you can use the `:augroup` command or just include a group name in your specification:

```
:autocmd cprograms FileReadPost *.h :set cindent
```

Now suppose you are editing a file called *sam.cx* that you would like treated as a C program. You can tell Vim to go through all the *cprograms* `autogroup` commands and execute the ones that match **.c* for the `FileReadPost` event. The command to do this is:

```
:doautocmd cprograms FileReadPost foo.c
```

(`:doautocmd` can be abbreviated `:do`.)

The general form of the `:doautocmd` command is this:

```
:doautocmd group event file_name
```

This executes the autocmd command group pretending that the current file is *file_name* rather than the current one. If the *group* is omitted, all groups are used and if *file_name* is left off, the current filename is used. The event must be specified and is the event that *Vim* pretends has just happened.

The following command does the same thing as `:doautocmd` except it executes once for each buffer:

```
:doautoall group event file_name
```

(Abbreviation `:doautoa`.)

Note: Do not use this to trigger autocmds that switch buffers, create buffers, or delete them. In other words, when using this command, leave the buffers alone.

Events

You can use the following events to trigger an autocmd:

BuFNewFile		Triggered when editing a file that does not exist.
BuFReadPre	BuFReadPost	Triggered before (BifReadPre) / after (BufReadPost) reading a buffer.
	BuFRead	Alias for BuFReadPost .
BuFFilePre	BuFFilePost	Before / after changing the name of a buffer with the <code>:file</code> (<code>:f</code>) command.

The Vim Tutorial and Reference

FileReadPre	FileReadPost	Before / after reading a file with the :read command. For FileReadPost , the marks '[' and ']' will be the beginning and end of the text read in.
FilterReadPre	FilterReadPost	Before / after reading a file with a filter command.
FileType		When the ' filetype ' option is set.
Syntax		When the syntax option is set.
StdinReadPre	StdReadPost	Before / after reading from the standard input. (The editor must have been started as vim -.)
BufWritePre	BufWritePost	Before / after writing the entire buffer to a file.
BufWrite		Alias for BufWritePre .
FileWritePre	FileWritePost	Before / after writing part of a buffer to a file.
FileAppendPre	FileAppendPost	Before / after appending to a file.
FilterWritePre	FilterWritePost	Before / after writing a file for a filter command.
FileChangedShell		This is triggered when Vim runs a shell command and then notices that the modification time of the file has changed.

The Vim Tutorial and Reference

FocusGained	FocusLost	Triggered when <i>Vim</i> gets or loses input focus. This means that <i>Vim</i> is running in GUI mode and becomes the current window or something else becomes the current window.
CursorHold		Occurs after the user pauses typing for more than the timeout specified by the ' updatetime ' option.
BufEnter	BufLeave	When a buffer is entered or left.
BufUnload		Triggered just before a buffer is unloaded.
BufCreate	BufDelete	Just after a buffer is created or just before it is deleted.
WinEnter	WinLeave	Going into or out of a window.
GuiEnter		The GUI just started.
VimEnter		The <i>Vim</i> editor just started and the initialization files have been read.
VimLeavePre		The <i>Vim</i> editor is exiting, but has not written the <i>.viminfo</i> file.
VimLeave		The <i>Vim</i> editor is exiting, and the <i>.viminfo</i> file has been written.
FileEncoding		The ' fileencoding ' option has just been set.
TermChanged		The term option changed.
User		Not a real event, but used as a fake event for use with : doautocmd .

The Vim Tutorial and Reference

When writing a file, *Vim* triggers only one pair of the following events:

BufWritePre	BufWritePost
FilterWritePre	FilterWritePost
FileAppendPre	FileAppendPost
FileWritePre	FileWritePost

When reading a file, one of the following set of events will be triggered:

BufNewFile	
BufReadPre	BufReadPost
FilterReadPre	FilterReadPost
FileReadPre	FileReadPost

File Patterns

The filename pattern matching uses the UNIX standard system. The following list identifies the special characters in the file matching patterns.

*	Match any characters, any length
?	Match any single character
,	Separates alternate patterns such as: <i>one,two,three</i> which matches the string <i>one, two, or three</i> .
\?	The question mark (?).
\,	The comma (,).
[abc]	Match one of the given characters.
[^abc]	Match any character except the given character.
{string}	The first time this occurs, match <i>string</i> and set \1 to the value of <i>string</i> . Thus {text}\1 and texttext match the same thing. The second {} will set \2, the third \3 and so on.
{one,two}	Match <i>one</i> or <i>two</i> and set \x to the value of what was matched.
\character	Treat character as a search pattern character. For example, a\+ matches a, aa, aaa, and so on.

Nesting

Generally, commands executed as the result of an autocmd event will not trigger any new events. For example, suppose you define a `:autocmd` which responds to the `Syntax` event by reading a file. You also have a `:autocmd` defined for the event `FileReadPre`. So a `Syntax` event is triggered, this causes a file read which would normally trigger a `FileReadPre` event. Except because you are already in a `:autocmd` and events do not nest, the second event is ignored. However, if you include the keyword `nested`, then events within events will trigger a `:autocmd`. For example:

```
:autocmd FileChangedShell *.c nested e!
```

Listing Autocommands

The following command lists all the autocommands:

```
:autocmd
```

For example:

```
:autocmd
--- Auto-Commands ---
filetype BufEnter
    *.xpm if getline(1) =~ "XPM2" | set ft=xpm2 | endif
    *.xpm2 set ft=xpm2
...
FileType
    *      set formatoptions=tcql nocindent comments&
            c      set formatoptions=croql cindent
...
filetype StdinReadPost
    * if !did_filetype() | so scripts.vim | endif
Syntax
    OFF syn clear
    abc so $VIMRUNTIME/syntax/abc.vim
(Listing truncated.)
```

From this, you can see a number of commands under the group `filetype`. These commands are triggered by the `BufEnter` and `StdinReadPost` events. There are also a couple of commands with no group name triggered by the `FileType` event. If you want a subset of all the commands, try the following:

```
:autocmd group event pattern
```

The Vim Tutorial and Reference

If *group* is specified, only the commands for that group are listed. *Event* can be one of the previously defined events or * for all events. The *pattern* specifies an optional file matching pattern. Only the commands that match are listed.

For example:

```
:autocmd filetype BufEnter *.xpm
--- Auto-Commands ---
filetype BufEnter
    *.xpm if getline(1) =~ "XPM2" | set ft=xpm2 | endif
```

Removing Commands

The command :**autocmd!** removes autocommands. The matching rules are the same for listing commands, so the following removes all the autocommands:

```
:autocmd!
```

To remove the commands for a specific group, execute this command:

```
:autocmd! group
```

You can also specify events and patterns for the group, as follows:

```
:autocmd! group event pattern
```

Again, event can be * to match all events. You can use the :**autocmd!** command to remove existing commands and define a new one in one command. The syntax for this is as follows:

```
:autocmd! group event pattern nested command
```

This is the equivalent of the following:

```
:autocmd! group event pattern
:autocmd group event pattern nested command
```

Ignoring Events

At times, you will not want to trigger an autocmd .The **eventignore** option contains a list of events that will be totally ignored. For example, the following causes all Window Enter and Leave events to ignore:

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

('**eventignore**' can be abbreviated as '**ie**')

To ignore all events, use the following command:

The Vim Tutorial and Reference

```
:set eventignore=all
```

Chapter 14: File Recovery and Command-Line Arguments

The *Vim* editor is designed to survive system crashes with minimum losses of data. This chapter discusses how to use *Vim*'s crash recovery procedures. In this chapter, you learn about the following:

- Command-line arguments for file recovery
- Encryption
- Batch files and scripts
- Additional command-line arguments
- Backup file options
- How to do file recovery
- Advanced swap file management

Command-Line Arguments

There are several useful command-line arguments. The most useful is **--help**, which displays a short help screen listing all the command-line arguments:

```
$ vim --help
VIM - Vi IMproved 7.0 (2006 May 7, compiled Jun 27 2006 19:50:12)

usage: vim [arguments] [file ..]          edit specified file(s)
      or: vim [arguments] -              read text from stdin
      or: vim [arguments] -t tag       edit file where tag is defined
      or: vim [arguments] -q [errorfile] edit file with first error

Arguments:
  --                                     Only file names after this
  -g                                     Run using GUI (like "gvim")
  -f  or  --nofork                      Foreground: Don't fork when starting GUI
-- lots more help --
```

To find out which version of *Vim* you have as well as to list the compilation options, use the following command:

The Vim Tutorial and Reference

```
$ vim --version
VIM - Vi IMproved 7.0 (2006 May 7, compiled Jun 27 2006 19:50:12)
Compiled by oualline@www.oualline.com
Normal version with GTK2 GUI. Features included (+) or not (-):
+arabic +autocmd +balloon_eval +browse +builtin_terms +byte_offset +cindent
+clientserver +clipboard +cmdline_compl +cmdline_hist +cmdline_info +comments
-- more compile options --
+wildignore +wildmenu +windows +writebackup +X11 -xfontset +xim +xsmp_interact
+xterm_clipboard -xterm_save
    system vimrc file: "$VIM/vimrc"
    user vimrc file: "$HOME/.vimrc"
    user exrc file: "$HOME/.exrc"
    system gvimrc file: "$VIM/gvimrc"
    user gvimrc file: "$HOME/.gvimrc"
    system menu file: "$VIMRUNTIME/menu.vim"
fall-back for $VIM: "/usr/local/share/vim"
Compilation: gcc -c -I. -Iproto -DHAVE_CONFIG_H -DFEAT_GUI_GTK -DXTHREADS
-- long command --
```

To view a file, you can "edit" it in read-only mode by using the -R command:

```
$ vim -R file.txt
```

On most systems, the following command does the same thing:

```
$ view file.txt
```

This actually starts "editing" the file with the '**modifiable**' ('**ma**') option turned off. You can turn on this option if you want to change the file.

Encryption

The **-x** argument tells *Vim* to encrypt the file. For example, create a file that contains something you want to keep secret:

```
$ vim -x secret.txt
```

The editor now prompts you for a key used for encrypting and decrypting the file:

```
Enter encryption key:
```

You can now edit this file normally and put in all your secrets. When you finish editing the file and tell *Vim* to exit, the file is encrypted and written.

If you try to print this file using the cat or type commands, all you get is garbage.

Switching Between Encrypted and Unencrypted Modes

The option '**key**' contains your encryption key. If you set this option to the empty string (""), you turn off encryption:

```
:set key=
```

If you set this to a password, you turn on encryption. For example:

```
:set key=secret (Not a good idea!)
```

Setting the encryption key this way is not a good idea because the password appears in the clear. Anyone shoulder surfing can read your password.

To avoid this problem, the **:x** command was created. It asks you for an encryption key and sets the key option to whatever you type in. (Note that the password will not be echoed. Instead * is printed for each character entered.)

```
:x
```

```
Enter encryption key:
```

Limits on Encryption

The encryption algorithm used by *Vim* is weak. It is good enough to keep out the casual prowler, but not good enough to keep out a cryptology expert with lots of time on his hands. Also you should be aware that the swap file is not encrypted; so while you are editing, people with superuser privileges can read the unencrypted text from this file.

One way to avoid letting people read your swap file is to avoid using one. If the **-n** argument is supplied on the command line, no swap file is used (instead, *Vim* puts everything in memory). For example, to edit the encrypted file *file.txt* and to avoid swap file problems use the following command:

```
$ vim -x -n file.txt
```

Note: If you use the **-n** argument, file recovery is impossible.

Also while the file is in memory, it is in plain text. Anyone with privileges can look in the editor's memory and discover the contents of the file. If you use a .viminfo file, be aware that the contents of text registers are written out in the clear as well. If you really want to secure the contents of a file, edit it only on a portable computer not connected to a network, use good encryption tools, and keep the computer locked up in a big safe when not in use.

Executing Vim in a Script or Batch File

Suppose you have a lot of files in which you need to change the string `--person--` to `Jones`. How do you do that? One way is to do a lot of typing. The other is to write a shell script or batch file to do the work.

The *Vim* editor does a superb job as a screen-oriented editor when started in normal mode. For batch processing, however, it does not lend itself to creating clear, commented command files; so here you will use ex mode instead. This mode gives you a nice command-line interface that makes it easy to put into a batch file.

The ex mode commands you need are as follows:

```
:%s/--person--/Jones/g  
:write  
:quit
```

You put these commands in the file `change.vim`. Now to run the editor in batch mode, use this command:

```
$ vim -es file.txt <change.vim
```

This runs the *Vim* editor in ex mode (`-e` flag) on the file `file.txt` and reads from the file `change.vim`. The `-s` flag tells *Vim* to operate in silent mode. In other words, do not keep outputting the `:` prompt, or any other prompt for that matter.

Additional Command-Line Arguments

A number of command-line arguments are designed to control the behavior of the editor. For example, you may want to restrict what you can do in *Vim*. The arguments to support this are as follows:

- R** Open the file for read-only.
- m** Modifications are not allowed. This argument is more of a recommendation than a restriction because all it does is set the '`nowrite`' option. It does not prevent you from setting the '`write`' option and modifying the file.
- z** Restricted mode. This prevents the user from using `:shell` or other commands to run an external shell. It does not prevent the user from trying to edit another file using the `:vi file` command.

The other arguments enable you to choose which initialization files you read:

The Vim Tutorial and Reference

- u file** Use file rather than *.vimrc* for initialization. If the filename is **NONE**, no initialization file is used.
- U file** Use file rather than *.gvimrc* for initialization. If the filename is **NONE**, no initialization file is used.
- i file** Use file rather than the *.viminfo* file.

In UNIX, the *Vim* editor is actually one file with several different names (links). The editor starts in different modes, depending on with which name it is started. The names include the following:

- vim** Start *Vim* in console mode. (Edits inside the current window.)
- gvim** Start *Vim* in GUI mode. (The editor creates its own window for editing.)
- ex** Start in ex mode. (Some systems link this to the system command *ex*.)
- view** Start in normal mode, read-only.
- gview** Start in GUI mode, read-only.
- rvim** Start in console mode, restricted.
- rview** Start in console mode, read-only, restricted.
- rgvim** Start in GUI mode, restricted.
- rgview** Start in GUI mode, read-only, restricted.
- vi** Linux and other similar systems. Alias for vim.

You can use command-line arguments to set the initial mode as well:

- g** Start *Vim* in GUI mode (same as using the command *gvim*).
- v** Start *Vim* in visual mode (same as using the command *vim*).
- e** Start *Vim* in ex mode (same as using the command *ex* on most systems).

You can use a number of command-line arguments to debug and test, including the following:

- v number** Display extra messages letting you know what is going inside the editor. The higher the number, the more output you get. This is used for debugging your *Vim* scripts.
- f** Foreground. Do not start a GUI in the background. This proves useful when gvim is run for another program that wants to wait until the program finishes. It is also

The Vim Tutorial and Reference

	extremely useful for debugging.
-w script	Write all characters entered by the user into the script file. If the script file already exists, it is appended to.
-w! script	Like -w , but overwrite any existing data.
-s script	Play back a script recorded with -w .
-T terminal	Set the terminal type. On UNIX, this overrides the value of the \$TERM environment variable. (Of course, if the \$TERM environment is wrong, lots of other programs will be screwed up as well.)

You also have compatibility arguments. These are of use only if you really want *Vim* to act like *Vi*.

-N	Non-compatible mode. This argument makes <i>Vim</i> act like <i>Vi</i> rather than <i>Vi</i> . This argument is set by default when a <i>.vimrc</i> file is present.
-C	Compatible. This turns off many of the <i>Vim</i> special features and makes the editor look as much like <i>Vi</i> as possible.
-l	Lisp mode. This mode is an obsolete holdover from the old <i>Vi</i> days. It sets the ' lisp ' and ' showmatch ' (' sm ') options. The <i>Vim</i> file-type-related commands do a much better job of handling Lisp programs, and they do it automatically.

Finally, you have a few arguments that cannot be classified any other way:

-d device	Amiga only. Open the given device for editing.
-b	Binary mode. Sets the options noexpandtab , textwidth=0 , nomodeline , and systems)..

Foreign Languages

The *Vim* editor can handle a variety of languages. Unfortunately, to edit these languages, you do not only need a *Vim* editor with the language features compiled in, but you also need special fonts and other operating system support. This means that unfortunately foreign language support is beyond the scope of this book.

But the command-line arguments for the foreign languages are as follows:

-F	Farsi
-H	Hebrew

Backup Files

Usually *Vim* does not produce a backup file. If you want to have one, all you need to do is execute the following command:

```
:set backup
```

('bk' is the short name for 'backup'.)

The name of the backup file is the original file with a "~" added to the end. If your file is named *data.txt*, for example, the backup file name is *data.txt~*. If you do not like the fact that the backup files end with ~, you can change the extensions by using the following:

```
:set backupext=string
```

('bex' is the short name for 'backupext'.)

If 'backupext' is .bak, *data.txt* is backed up to *data.txt.bak*.

The *Vim* editor goes you one better when it comes to the backup file. If you set the 'patchmode' ('pm') option, *Vim* backs up the file being edited to a file with the same name, but with the 'patchmode' string appended to it. This will be done only if the file does not exist. For example, suppose you execute this command:

```
:set patchmode=.org
```

Now you edit the existing file *data.txt* for the first time. When you exit *Vim* checks to see whether the file *data.txt.org* exists. It does not, so the old file is saved under that name. The next time you edit, the file does exist; so the backup is written to *data.txt~*. The file *data.txt.org* is not used from now on. Instead, all backups will go to *data.txt~*.

Usually *Vim* puts the backup file in the same directory as the file itself. You can change this with the 'backupdir' ('bdir') option. For example, the following causes all backup files to be put in the ~/tmp directory:

```
:set backupdir=~/tmp/
```

This can create problems if you edit files of the same name in different directories. That is because their backup files will all go to the ~/tmp directory and the name collision will cause the old backup files to disappear.

The 'backupdir' option can actually take a series of directories, separated by comma. The editor puts the backup file in the first directory where a backup file can be created.

Skipping the backup

For some directories it is not useful to write a backup file. For example you don't really need to backup temporary files in `/tmp`. The list of such files is defined by the '**backupskip**' ('**bks**') option. It is a list of strings. Any file matching any one of these strings will not be backed up.

Controlling How the File Is Written

Generally when *Vim* writes a file, the following operations are performed:

1. *Vim* checks to see whether the file has been changed outside of *Vim*. For example, someone could have overwritten the file with a new one. If this happens, a warning is issued and the editor asks if you want to continue.
2. If the '**writebackup**' ('**wb**') or '**backup**' option is set, any old backup file is removed. The current file is then copied to the backup file.
3. The buffer is written out to the file.
4. If the '**patchmode**' option is set and no patch file exists, the backup file is renamed to become the patch file.
5. If the '**backup**' option is not set, and '**writebackup**' is set, remove the backup file.

The reason that *Vim* overwrites the existing file is to preserve any hard links that you might have on a UNIX system. On non-UNIX systems the backup is created by renaming the original file instead of making a copy.

Note: If you set the '**nobackup**' and '**nowritebackup**' options, *Vim* just overwrites the existing file. This can cause loss of data if the disk fills up during the file update.

By default, the '**writebackup**' option is set. This means that the system *Vim* uses to write a file makes it very difficult to lose data. By using this method, there is no chance you will lose your file if the disk fills up. You may not be able to write out the new version of the file, but at least you do not lose the old one.

There are two ways to create a backup file. The one discussed above is triggered when '**backupcopy**' ('**bkc**') is set to **no**. The backup method is:

1. Write out the data to a temporary file.

2. Delete old backup.
3. Rename *current* --> *backup*
4. Rename *temp* -> *current*

There are a number of subtle problems with this system. First any hard links that pointed to the original file, now point to the backup. Second if the original file was owned by someone else, it is now owned by you.

To avoid these problems (and create others) you can set '**backupcopy**' to **yes**. If this happens the *Vim* creates the backup file using the following steps:

1. Copy *current* -> *backup*
2. Truncate *current*
3. Write out all data to *current*.

Because the file is not deleted or renamed, all the links and permissions remain. However, it does take time to copy the original file to the backup.

The '**backupcopy**' option has a third value: auto. When set to **auto** *Vim* will figure out the most efficient way of creating the backup and then use it.

Basic File Recovery

Suppose that you want to edit a file called *sample.txt*. You start *Vim* with the following command:

```
$ gvim sample.txt
```

The editor now creates a swap file to temporarily hold the changes you make until you write the file. When you finish editing, the swap file is deleted.

If the editor is aborted during mid-edit, however, it does not get a chance to delete the swap file. This means that if you are in the middle of *Vim* sessions and your system locks, forcing a reboot, the swap file will not be deleted.

When *Vim* first starts editing a file, it checks for a swap file. If it finds one, that means that either another editing session is in progress or another editing session was started and the editor got aborted. Therefore, *Vim* issues a warning (see Figure 14-1), and gives you a chance to decide what to do.

The Vim Tutorial and Reference

```
~  
~  
~  
~  
~  
ATTENTION  
Found a swap file by the name ".sample.txt.swp"  
          dated: Thu Feb 17 22:44:00 2000  
          owned by: sdo  
          file name: /tmp/sample.txt  
          modified: no  
          host name: www.oualline.com  
          user name: sdo  
          process ID: 8449 (still running)  
While opening file "sample.txt"  
          dates: Thu Feb 17 22:45:33 2000  
(1) Another program may be editing the same file.  
    If this is the case, be careful not to end up with two  
    different instances of the same file when making changes.  
    Quit, or continue with caution.  
(2) An edit session for this file crashed.  
    If this is the case, use ":recover" or "vim r sample.txt"  
    to recover the changes (see ":help recovery").  
    If you did this already, delete the swap file ".sample.txt.swp"  
    to avoid this message.
```

Figure 14-1: File in use warning.

At this point, you have a number options:

Open Read-Only

This option causes *Vim* to open the file read-only. You should choose this option if you want to look at the file and there is another editing session still running.

Edit anyway

A.K.A. Damn the torpedoes, full steam ahead. If you select this option, you can edit the file. Do not choose this option unless you really know what you are doing. Note that if you have two or more edit sessions running on a single file, the last session to write the file wins.

Recover

If you were editing the file and the editor got aborted due to a system crash or some other reason, choose this option. It examines the swap file for changes and attempts to restart your session from where you left off. It usually comes close, but examine your file carefully because the last few edits may have disappeared.

Quit

Do not edit this file.

Delete

Delete the swap file. (Appears if no one else is using the swap file.)

After selecting one of these options, you can edit normally. Be careful if you choose Recover, because all your changes may not have been saved.

Recovering from the Command Line

If you know the name of the file you were editing when your editing session was aborted, you can start *Vim* in recovery mode using the **-r** argument. If you were editing the file *commands.c* when you were rudely interrupted, for example, you can recover with the following command:

```
$ vim -r commands.c
```

If you want to get a list of recoverable editor sessions, use this command:

```
$ vim -r
```

This causes *Vim* to check for swap files in the current directory and the standard temporary directories. For example:

```
$ vim -r
Swap files found:
  In current directory:
    -- none -
  In directory ~/tmp:
    -- none -
  In directory /var/tmp:
    -- none -
  In directory /tmp:
    1. .script.txt.swp
          dated: Fri Feb 18 19:48:46 2000
          owned by: sdo
          file name: /tmp/script.txt
          modified: no
          host name: www.oualline.com
          user name: sdo
          process ID: 26473 (still running)
```

In this example, you see that there is a swap file for the file */tmp/script.txt*. The process number of the editor that created the swap file is 26473. The process is still running, so you probably do not want to try to edit or recover the file using this edit session. You probably want to find the window for process 26473 and use it instead.

Several options and other commands affect file recovery. See the next section for more information.

Advanced Swap File Management

The *Vim* editor goes to a great deal of trouble not to overwrite any old swap files. The first time a file is edited, the swap file name is *.file.txt.swp*. If the editor is aborted and you start editing again, the next swap file is called *.file.txt.swo*, and then *.file.txt.swn*, and so on. You can tell *Vim* to recover using a specific swap file by specifying the name of the swap file with the command:

```
$ vim -r file.txt.swo
```

To find out the name of the swap file you are currently using, execute the following command:

```
:swapname
```

(*;swapname* can be abbreviated as *:sw.*)

This displays the name of the swap file.

Controlling When the Swap File Is Written

Usually the swap file is written every 4 seconds or when you type 200 characters. These values are determined by the '**updatecount**' ('**uc**') and '**updatetime**' ('**ut**') options. To change the amount of time *Vim* waits before writing the swap file to 23 seconds, for example, use the following command:

```
:set updatetime=23000
```

Note: The '**updatetime**' is specified in milliseconds.

To change the number of characters you have to type before *Vim* writes stuff to the swap file to 400, for instance, use this command:

```
:set updatecount=400
```

If you change the '**updatecount**' to 0, the swap file will not be written. However, the decision whether to write a swap file is better controlled by the '**swapfile**' option. If you have this option set, a swap file will be created (the default):

```
:set swapfile
```

If you do not want a swap file, use the following command:

```
:set noswapfile
```

('**swf**' is the short name for '**swapfile**'.)

The Vim Tutorial and Reference

This option can be set/reset for each edited file. If you edit a huge file and don't care about recovery, set '**noswapfile**'. If you edit a file in another window, it will still use a swap file.

On most operating systems, when you "write" a file, the data usually goes into a memory buffer and is actually written to the disk when the operating system "thinks" it is appropriate. This usually takes only a few seconds. If you want to make sure that the data gets to disk, however, you want to use the following command:

```
:set swapSync
```

('**sfs**' is the short name for '**swapSync**')

This command tells *Vim* to perform a sync operation after each writing of the swap file to force the data onto the disk. The '**swapSync**' option can be empty, **fsync**, or **sync**, depending on what system call you want to do the writing.

Controlling Where the Swap File Is Written

Generally, *Vim* writes the swap file in the same directory as the file itself. You can change this by using the '**directory**' ('**dir**') option. For example, the following tells *Vim* to put all swap files in */tmp*:

```
:set directory=/tmp (Not a good idea)
```

This is not a good idea because if you try to edit the file *readme.txt* in two different directories at the same time, you encounter a swap file collision.

You can set the '**directory**' option to a list of directories separated by a comma (,). It is highly recommended that you use a period (.) as the first item in this list. The swap file will be written to the first directory in the list in which *Vim* can write the file. For example, the following tells *Vim* to write the swap file in the current directory, and then to try */tmp*:

```
:set directory=.,/tmp
```

Advanced File Writing

Normally when *Vim* writes a file, it just writes a file. If the '**fsync**' ('**fs**') option is on, it will write a file, then do an fsync() system call to make sure it gets written to disk. This helps protect you against loss of data when there's a power outage. However on some laptops it will cause the disk to spin up thus draining the battery.

Saving Your Work

Suppose you have made a bunch of changes and you want to make sure they stick around even if *Vim* or the operating system crashes. One way to save your changes is to use the following command to write out the file:

```
:write
```

(`:w` does the same thing.)

However, this command overwrites your existing file with all your changes. The following is a related command:

```
:preserve
```

(`:preserve` can be written as `:pre.`)

This command writes all the edits to the swap file. The original file remains unchanged and will not be changed until you do a `:write` or exit with `zz`. If the system crashes, you can use the swap file to recover all your edits. Note that after a `:preserve`, you can recover even if the original file is lost. Without this command, you need both the original file and the swap file to recover.

The `:recover` Command

The following command tries to recover the file named *file.txt*:

```
:recover file.txt
```

(`:rec` is the short form of `:recover.`)

It is just like this command:

```
$ vim -r file.txt
```

If the file you are trying to recover is currently being edited this command fails. If no filename is specified, it defaults to the file in the current buffer.

If you want to discard any changes you have made to the file and attempt to recover, use the following command:

```
:recover! file.txt
```

MS-DOS Filenames

If you are on an MS-DOS or Windows 3.1 machine, you are stuck with very limited filenames. The *Vim* editor detects this and limits the swap filename to something that can be used on this type of machine. Whereas the normal swap file for *foo.txt* is *.foo.txt.swp*, for example, if you are in short name mode, it is *foo_txt.swp*.

You can set the '**shortname**' option to force *Vim* to use this convention. This is useful if have a Linux or other system and are editing files on an MS-DOS partition. In this case, the operating system (Linux) supports long filenames, but the actual disk you are working on (MS-DOS format) does not. Therefore, you need to tell *Vim* to use the short swap names by giving it the following command:

```
:set shortname
```

This option is not available for the MS-DOS version of *Vim* because it would be always on. Instead, it is used when you are cross-platform editing.

readonly and modified Options

The '**modified**' ('**mod**') flag is set if the buffer has been modified. You probably do not want to set this option yourself because it is handled automatically. You can use the value of this option in macros, however.

The '**readonly**' ('**ro**') flag is also set automatically if the file is read-only. In only one circumstance should you reset this: when you are using a source control system that normally leaves files in read-only mode. You want to edit the file, so you start *Vim*.

The editor warns you that the file is read-only and sets the '**readonly**' option. At this point, you realize that you forgot to tell the source control system that you want to edit the file. So you use **:shell** (**:sh**) to go to the command prompt and execute the commands needed to tell the system that you want to edit the file. The RCS system uses the *co -l* command to do this, for example; the SCCS system uses *sccs edit*.

After getting permission to edit the file, you use the exit command to return to

Vim, where you execute the following command to mark the file as editable:

```
:set noreadonly
```

Chapter 15: Miscellaneous Commands

This chapter discusses all the commands that do not quite fit in any other chapter. In this chapter, you learn about the following:

- Getting character number information
- How to go to a specific byte in the file
- Redrawing the screen
- Sleeping
- Terminal control
- Suspending the editor
- Reshowing the introduction screen

Printing the Character

The command **:ascii (:as)** or **ga** prints the number of the character under the cursor. The output looks like this:

```
<*> 42, Hex 2a, Octal 052
```

If editing a multibyte (Japanese or Chinese for example) file, and the character under the cursor is a double-byte character, the output shows both bytes.

Going to a Specific Character in the File

The **countgo** command goes to byte number count of the file. The command **g **CTRL-G**** displays the current byte number of a file (along with the current line, column, and other information).

The command **:goto offset (:go)** also positions the cursor to a given byte location within the file.

The **gg** command acts much like the **G** command. It goes to the line specified by its count. For example, **5gg** goes to line 5. The difference between **gg** and **G** is that if no count is specified, **gg** goes to the first line and **G** goes to the last.

Screen Redraw

The **CTRL-L** command redraws the screen. This proves useful when you are on a terminal and some system message or other text screws up your screen. With the advent of the dedicated GUI, the need for this command is greatly diminished.

Sleep

The **:sleep time (:s1)** command does nothing for the specified number of seconds. If time ends in **m**, it is specified in milliseconds. This command proves useful when you want to pause during the execution of a macro.

The **countgs** command also sleeps for count seconds.

Terminal Control

On most terminals, the **CTRL-S** command stops output. To restart it again, you type **CTRL-Q**. These commands are not part of *Vim*; to avoid keyboard conflicts, however, they are not used by any *Vim* commands.

You should not try to use these commands in a **:map** command because your terminal might interpret them and they might never get to *Vim*.

Suspending the Editor

If you are on UNIX in terminal mode, you can suspend the editor with the normal mode command **CTRL-Z**. To continue editing, use the shell command *fg*. This works only on shells that have job control. The **:suspend** command does the same thing.

Note: **CTRL-Z** in insert mode inserts the character **CTRL-Z**; it does not suspend the editor.

General Help

The **:help**, **:h**, **<F1>** and **<Help>** commands all display the general help screen.

Other Help Commands

The **:helpgrep** (**:helpg**) command does a **:vimgrep** on the help files and lets you go through the results with **:cc**, **:cn**, **:cp**, and the other **:vimgrep** related commands. The **:lhelpgrep** (**:lh**) command does the same thing only using the local version of the **:vimgrep** command,

The **:helptags** (**:helpt**) command generates a help tags files for all the files in a given directory. This is useful only if you are writing help files.

Nvi Compatibility Commands

For *Nvi* compatibility some additional commands have been added. The **:exusage** (**:exu**) command displays help on the command mode commands (aka the *Ex* commands.) The **:viuslage** (**:viu**) command displays help on the normal mode commands. Both these commands do not work as well as **:help** but were included for compatibility.

Window Size

The **z height <CR>** command resizes the current window to height. If there is only one window open, *Vim* will display only height lines. (The rest will be blank.) This is useful for slow terminals.

Executing commands without changing things

The **:keepalt** (**:keepa**) command executes a command without changing the fname of the alternate file. For example:

```
:keepalt :next
```

Edits the next file but does not make the current one the alternate file.

The **:keepjumps** (**:keepj**) command executes a command without changing the marks that are normally changed by a jump.

Signs

A sign is small marker on the left side of the screen. The sign feature is designed to help *Vim* work with visual debuggers or IDEs. It gives the IDE the ability to show things like breakpoints and other annotations.

Before you can use a sign, it must be defined. In this example we are defining a sign named *Here* that uses the text “=>” to point to a location.

The Vim Tutorial and Reference

```
:sign Here icon=here.xpm text==>
\    linehl=Search texthl=DiffChange
```

(**:sign** can be abbreviated **:sig.**)

The parameters to this command are:

icon	The name of the icon to use (if an icon can be displayed.) This only works for the GTK and Motif versions of the GUI.
text	The text to display.
linehl	Highlight to use for the line on which the sign has been placed.
texthl	Highlight for the sign text.

(See the section *Customizing the Syntax Highlighting* in *Chapter 23: Advanced Commands for Programmers* for information on highlighting.)

Now that we've defined a sign, we can put it somewhere. In this case we are going to put it on line 3 of the file *test.txt*.

```
:sign place 1 line=3 name=Here file=test.txt
```

The **1** is the sign id number. The **line** is the line on which to place the sign. The last parameter is the name of the file.

```
Line 1
..Line 2
=>Line 3
Line 4
Line 5
```

Figure 15-1: Sign placement

A sign can be placed by file name or by buffer number. For example:

```
:sign place 1 line=3 name=Here buffer=1
```

If a sign is already in place you can change the name of the sign with the command:

```
:sign place 1 name=There file=test.txt
```

This changes the sign from *Here* to *There*. (Assuming of course we have done a **:sign define** on *There*.)

The Vim Tutorial and Reference

Again we can use a buffer instead of a file name:

```
:sign place 1 name=There buffer=1
```

To remove a sign, use the `:sign unplace` command. This can take a file name or buffer number as an argument.

```
:sign unplace 1 file=test.txt  
:sign unplace 1 buffer=1  
:sign unplace 1
```

The last form removes all the signs for that id for all buffers.

To remove the sign at the cursor location, use the command:

```
:sign unplace
```

Finally to remove all the sign, the command is:

```
:sign unplace *
```

You can move to a sign within a given buffer or file with the `:sign jump` command:

```
:sign jump 1 file=test.txt  
:sign jump 1 buffer=1
```

To find out what signs are defined, use the `:sign list` command.

```
:sign list
```

To limit results to a single sign, just specify it on the command line:

```
:sign line Here
```

To find out where the signs are located use the command:

```
:sign list
```

You can limit yourself to a single buffer or file, by specifying them on the command line:

```
:sign list file=test.txt  
:sign list buffer=1
```

Viewing the Introduction Screen

If you start *Vim* without a filename, you will see an introductory flash screen. This screen disappears when you type the first character. If you want to see it again, issue the following command:

```
:intro
```

The Vim Tutorial and Reference

(`:int` does the same thing.)

Open Mode

The *Vim* editor has all the capabilities of the *Vi* editor except one: **open** mode. This mode is *Vi*'s way of coping with terminals it does not understand. It is difficult to get into this mode, difficult to use it, and the fact that *Vim* does not have it is no great loss.

Vim does have a command to enter open mode, but when you issue the command

:open

all you get is an error message.

(`:o` will also not get you into open mode.)

Chapter 16: Cookbook

This chapter presents a cookbook full of short recipes for doing some common (and not so common) *Vim* editing. The "recipes" include the following:

- Character twiddling
- Replacing one word with another using one command
- Interactively replacing one word with another
- Moving text
- Copying a block of text from one file to another
- Sorting a section
- Finding a procedure in a C program
- Drawing comment boxes
- Reading a UNIX man page
- Trimming the blanks off an end-of-line
- Oops, I left the file write-protected
- Changing Last, First to First Last
- How to edit all the files containing a given word
- Finding all occurrences of a word

Character Twiddling

If you type fast, your fingers can easily get ahead of your mind. Frequently people transpose characters. For example, the word `the` comes out `teh`. To swap two characters (for example, `e` with `h`), put the cursor on the `e` and type `xp`. The `x` command deletes a character (the `e`), and the `p` pastes it after the cursor (which is now placed over the `h`).

Replacing One Word with Another Using One Command

Suppose you want to make all `idiots` into `managers`. Execute the following command:

```
:1,$s/idiots/managers/g
```

The colon (:) indicates that you are going to execute an ex type command. All ex commands begin with range of line numbers on which the command operates. In this case, the whole document is chosen, from line 1 to the last line (\$). The shorthand for 1,\$ is simply % as shown previously.

The :s (abbreviation for :**s**ub**s**t**it**ute) command performs a substitution. The old text follows enclosed in slashes (/idiots/). The replacement text comes next, also delimited by the slashes (/managers/). The g flag tells the editor that this is a global change and so if the word idiots appears more than once on a line, to change them all.

The Virgin What!?

A church just bought its first computer and was learning how to use it. The church secretary decided to set up a form letter to be used in a funeral service. Where the person's name was to be, she put in the word <name>. When a funeral occurred, she would change this word to the actual name of the departed.

One day, there were two funerals, first for a lady named Mary, and then later one for someone named Edna. So the secretary used global replace to change <name> to Mary. So far, so good.

Next, she generated the service for the second funeral by changing the word Mary to Edna. That was a mistake

Imagine the minister's surprise when he started reading the part containing the Apostles' Creed and saw,

"Born of the Virgin Edna."

Interactively Replacing One Word with Another

Suppose you want to replace every occurrence of the word **idiot** with the word **manager**, but you want the chance to review each change before you do it.

To do so, follow these steps:

1. Use **1G** to go to the top of the document.
2. Execute **/idiot** to find the first occurrence of the word **idiot**.
3. Issue the command **cwmanager<Esc>**. Change the word (**cw**) to **manager**.
4. Use the **n** command to repeat the last search (find the next idiot).

The Vim Tutorial and Reference

5. Execute the . (dot) command to repeat the last edit (change one word to manager). If you do not want to change the word, skip this step.
6. Repeat steps 4 and 5 until you have replaced all occurrences of `idiot` to `manager`.

Alternate Method

Execute the following command:

```
:%s/idiot/manager/cg
```

This starts an ex-mode command `:substitute` (abbreviated `:s`). The `%` tells *Vim* to apply this command to every line in the file. You change `idiot` to `manager`. The `c` flag tells `:substitute` to get confirmation before each change. The `g` flag tells the command to change all occurrences on the line. (The default is to change just the first occurrence.)

Moving Text

Suppose you want to move a bunch of paragraphs from the top of the document to the bottom.

To do so, follow these steps:

1. Move the cursor to the top of the paragraph you want to move.
2. Use `ma` to place a mark named `a` at this location.
3. Move the cursor to the bottom of the paragraph to be moved.
4. Execute `d'a` to delete to mark `a`. This puts the deleted text in a register.
5. Move the cursor to the line where the text is to go. The paragraph will be placed after this one.
6. Use the `p` command to paste the text below the cursor.

Another method consists of the following steps:

1. Select the first line as in the previous list; make it mark `a`.
2. Move to the bottom of the paragraph (use the `;` command). Mark `b`.

3. Move to the line above where you want to put the text, and type the command:

```
: 'a, 'b move .
```

(`:m` is short for `:move`.)

Copying a Block of Text from One File to Another

The old *Vi* editor did not handle multiple files very well. Fortunately, *Vim* does a superb job of dealing with more than one file. There are lots of different ways to copy text from one to another. If you are used to the traditional *Vi*-style commands, you can use a method based around that style. On the other hand, *Vim* has a very nice visual mode. You can use it as well. Finally, *Vim* can make use of the system Clipboard to move text from one *Vim* program to another. All these methods work, and work well. Which one you should use is a matter of taste.

Method 1: Two Windows with Traditional Vi-Style Commands

To copy a block of text between files, follow these steps:

1. Edit the first file.
2. Execute `:split second_file` to go to the second file. This opens another window and starts editing the second file in it.
3. Use `CTRL-W p` to go to the "previous" window, the one with the original file.
4. Go to the top line to be copied.
5. Mark this line as mark a by using the `ma` command.
6. Go to the bottom line to be copied
7. Execute `y'a` to yank (copy in Microsoft parlance) the text from the current cursor location to mark a ('a) into the default register.
8. Use `CTRL-W p` to go to the file that will receive the text.
9. Go to the line where the insert is to occur. The text will be placed before this line.
10. Issue the `P` command to put (paste in Microsoft terminology) the text in the default register above the current line.

Method 2: Two Windows Using Visual Mode

To copy a block of text between files, follow these steps:

1. Edit the first file.
2. Execute **:split** to edit the second file.
3. Use **CTRL-W p** to go to the "previous" window, the one with the original file.
4. Go to the start of the text to be copied.
5. Issue the **v** command to start visual mode.
6. Go to the end of the text to be copied. The selected text will be highlighted.
7. Execute **y** to yank (Copy in Microsoft parlance) the text into the default register.
8. Use **CTRL-W p** to go to the file that will receive the text.
9. Go to the line where the insert is to occur. The text will be placed before this line.
10. Issue the **P** command to put (paste in Microsoft terminology) the text in the default register above the current line.

Method 3: Two Different Vim Programs

In this method, you start up two *Vim* programs and copy text from one to another. You do this by using the system Clipboard register ("*").

1. Edit the first file.
2. Start another *Vim* program to edit the second file. (If you are using Linux or UNIX, the second editor can be started on another machine. However, both programs must use the same X Windows display.)
3. Go to the window with the first file in it.
4. Go to the start of the text to be copied.
5. Issue the **v** command to start visual mode.

The Vim Tutorial and Reference

6. Go to the end of the text to be copied. The selected text will be highlighted.
7. Use the `"*y` command to yank (copy in Microsoft parlance) the text into the system Clipboard register (*).
8. Change to the other editing command. (Make that editor your active window.)
9. Go to the line where the insert is to occur. The text will be placed before this line.
10. Issue the command `"*p` to put (paste in Microsoft terminology) the text in the system Clipboard register (*) above the current line.

Note: This method enables you to not only move text between two *Vim* applications, but also to "yank" and "put" between *Vim* and other applications as well. For example, you can select text in an *xterm* window using the mouse and paste it into a *Vim* editing using `"*p`. Or you can copy text into the system register in a *Vim* session and paste it into a Microsoft Word document using the Edit, Paste commands.

Or if you're editing a book about *Vim* using OpenOffice, you can yank the text using `"*p` and paste it into a figure inside your book using the middle mouse button.

Sorting a Section

Frequently you will be editing a file with a list of names in it (for example, a list of object files that make up a program). For example:

```
version.o
pch.o
 getopt.o
 util.o
 getopt1.o
 inp.o
 patch.o
 backupfile.o
```

This list would be nice in alphabetic order (or at least ASCII order). To do this using *Vim* commands execute the following:

1. Move the cursor to the first line to be sorted.
2. Issue the `v` command to enter visual mode.

The Vim Tutorial and Reference

3. Move to the bottom of the text to be sorted. The text will be highlighted.

4. Execute the **:sort** command.

Warning: In actual practice, what you see in most Makefiles (files used by UNIX to control compilation) looks more like this:

```
OBJS = \
    version.o \
    pch.o \
    getopt.o \
    util.o \
    getopt1.o \
    inp.o \
    patch.o \
    backupfile.o
```

Notice that the backslash (\) is used to indicate a continuation line. After sorting this looks like the following:

```
OBJS = \
    backupfile.o
    getopt.o \
    getopt1.o \
    inp.o \
    patch.o \
    pch.o \
    util.o \
    version.o \
```

The names are in order, but the backslashes are wrong. Do not forget to fix them using normal editing before continuing:

The Vim Tutorial and Reference

```
OBJS = \
    backupfile.o \
    getopt.o \
    getopt1.o \
    inp.o \
    patch.o \
    pch.o \
    util.o
```

(added)

(removed)

Sorting the old Vi way:

In this method we avoid using *Vim*'s internal **:sort** command and instead use an external command. This system also avoids visual mode.

1. Move the cursor to the first line to be sorted.
2. Use the command **ma** to mark the first line as mark **a**.
3. Move to the bottom of the text to be sorted.
4. Execute the **! 'asort** command. The **!** command tells *Vim* to run the text through a command. The **'a** tells the editor that the text to be worked on starts at the current line and ends at mark **a**. The command that the text is to go through is *sort*.

The result looks like this:

```
backupfile.o
getopt.o
getopt1.o
inp.o
patch.o
pch.o
util.o
version.o
```

Finding a Procedure in a C Program

The *Vim* program was designed by programmers for programmers. You can use it to locate procedures within a set of C or C++ program files.

First, however, you must generate a table of contents file called a tags file. (This file has been given the obvious name *tags*.) The *ctags* command generates this table of contents file.

The Vim Tutorial and Reference

To generate a table of contents of all the C program files in your current working directory, use the following command:

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

For C++, use this command:

```
$ ctags *.cpp *.h
```

If you use an extension other than *.cpp* for your C++ files, use it rather than *.cpp*.

After this file has been generated, you tell *Vim* that you want to edit a procedure and it will find the file containing that procedure and position you there. If you want to edit the procedure `write_file`, for example, use the following command:

```
$ gvim -t write_file
```

Now suppose as you are looking at the `write_file` procedure that it calls `setup_data` and you need to look at that procedure.

To jump to that function, position the cursor at the beginning of the word `setup_data` and press **CTRL-]**. This tells *Vim* to jump to the definition of this procedure. This repositioning occurs even if *Vim* has to change files to do so.

Note: If you have edited the current file and not saved it, *Vim* will issue a warning and ignore the **CTRL-]** command.

There are a number of tag-related commands enable you to jump forward/backward through tags, split the windows and put the called procedure in the other window, find inexact tags, and many more things. These can be found in Chapter 7.

Drawing Comment Boxes

I like to put a big comment box at the top of each of my procedures. For example:

The Vim Tutorial and Reference

```
*****  
* Program -- Solve it -- Solves the worlds problems. *  
* All of them. At once. This will be a great *  
* program when I finish it. *  
*****
```

Drawing these boxes like this is tedious at best. But *Vim* has a useful feature called abbreviations that makes things easier. First, you need to create a *Vim* initialization file called `~/.vimrc`. The `~/.vimrc` file must contain the following lines:

```
:ab #b /*****  
:ab #e <Space>*****
```

These commands define a set of *Vim* abbreviations. Abbreviations were discussed in *Chapter 8: Basic Abbreviations, Keyboard Mapping, and Initialization Files*. To create a comment box, enter `#b<Enter>`. The screen looks like this:

```
*****
```

Enter the comments, including the beginning and ending * characters. Finally, end the comment by typing `#e<Enter>`. This causes the ending comment to be entered.

Another (better) option is to use an external program like boxes (see <http://www.vim.org>), which generates all kinds of ASCII art boxes and can be customized. Here, one might visually select the text and then issue the command `:<, '> !boxes -r`, which would remove an existing box and put a new box around the text.

Note: This page was written in *Vim*. So how did we enter the `#b` and `#e`? Easy, we typed in `#bb` and then deleted a character. (We could not enter `#b` or it would have been expanded.) The actual command was `i#bb<Esc>x`.

Another good tool for this sort of thing is *tal*, which lines up the final character (the *, here) so it looks nice.

Reading a UNIX man Page

You can use the *Vim* editor to browse through text files. One of the most useful sets of files to browse through is the man pages. Unfortunately, man pages try to simulate formatting by underlining characters using a sequence such as `_<BS>x` for x. This make viewing of the man page in *Vim* difficult. If you try to read a man page directly, you will see something like this:

The Vim Tutorial and Reference

```
N^HNA^HAM^HME^HE  
date - print or set the system date and time
```

To get rid of these characters, use the standard UNIX command *ul -i*. This is a formatting program that removes the hard-to-read control characters. The result looks like this:

```
NAME  
!!!!  
date - print or set the system date and time
```

Now all that is needed is to put three commands together: the *man* command to get the manual page, the *ul -i* command to fix the formatting, and *vim* to read the page. The resulting command is as follows:

```
$ man date | ul -i | vim -
```

Another technique is to use *Vim* on the raw page and then execute:

```
:%s/.\\b//g
```

This will remove all characters followed by the backspace (\b), rendering the file readable.

Trimming the Blanks off an End-of-Line

Some people find spaces and tabs at the end of a line useless, wasteful, and ugly. To remove whitespace at the end of every line, execute the following command:

```
:%s/ \s*/$/ /
```

The colon (:) tells *Vim* to enter command mode. All command-mode commands start with a line range; in this case, the special range % is used, which represents the entire file.

The first set of slashes encloses the "from text." The text is "any whitespace" (\s), repeated zero or more times (*), followed by "end-of-line" (\$). The result is that this pattern matches all trailing whitespace.

The matching text is to be replaced by the text in the next set of slashes. This text is nothing, so the spaces and tabs are effectively removed.

Oops, I Left the File Write-Protected

Suppose you are editing a file and you have made a lot of changes. This is a very important file and to preserve it from any casual changes, you write-protected it, even against yourself. The *Vim* editor enables you to edit a write-protected file with little or no warning. The only trouble is that when you try to exit using **zz** you get the following error:

```
file.txt File is read-only
```

and *Vim* does not exit.

So what can you do? You do not want to throw away all those changes, but you need to get out of *Vim* so that you can turn on write permission. Use the command **:w!** to force the writing of the file.

Another option: Use the **:w otherfilename** command to save your work in a different file so you can fix the file permissions if that is what is required.

Changing Last, First to First Last

You have a list of names in the following form:

```
Last, First
```

How do you change them to

```
First Last
```

You can do so with one command:

```
:1,$s/\(\[^,\]*\), \(.*\$)/\2 \1/
```

The colon (**:**) tells *Vim* that this is an ex-style command.

The line range for this command is the whole file, as indicated by the range **1,\$**.

The **s** (abbreviations for **:substitute**) tells *Vim* to perform a string substitution.

The old text is a complex regular expression. The **\(. . . \)** delimiters are used to inform the editor that the text that matches the regular expression inside the parentheses is to be remembered for later use. The text in the first **\(. . . \)** is assigned to **\1** in the replacement text. The second set of text inside **\(. . . \)** is assigned **\2**, and so on.

The Vim Tutorial and Reference

In this case, the first regular expression is any bunch of characters that does not include a comma. The `[^,]` means anything but a comma, and the `*` means a bunch (zero or more characters). Note: This means that all leading spaces will also be matched, which may not be what is desired.

The second expression matches anything (`.*`) up to the end-of-line: `(\$)`.

The result of this substitution is that the first word on the line is assigned to `\1` and the second to `\2`. These values are used in the end of the command to reverse the words.

Figure 16-1 example shows the relationship between the `\(\)` enclosed strings and the `\1`, `\2` markers.

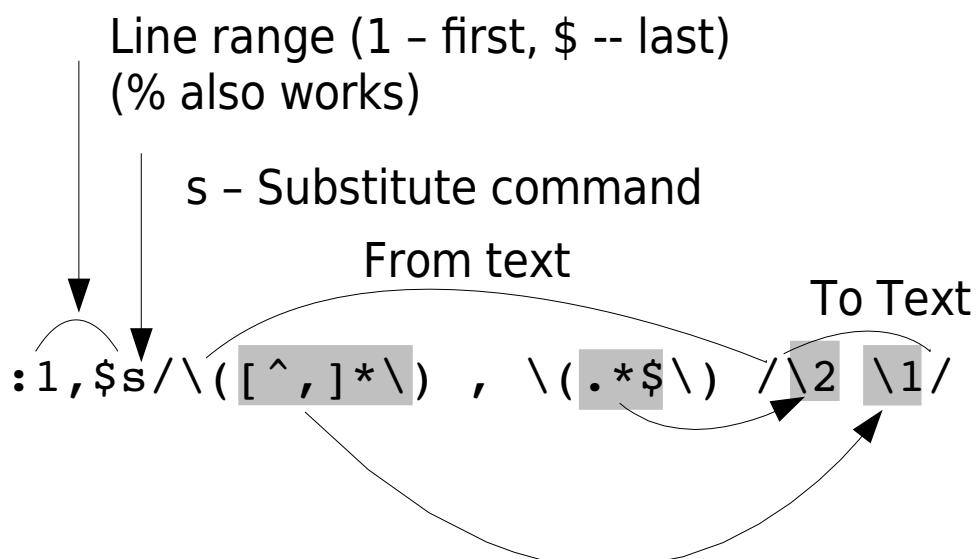


Figure 16-1: Regular expression explained

Match anything but comma Any Character Repeated 0 or more times The end of the line

Figure 16-2 breaks out the various parts of the regular expressions used in this illustration:

```
:1,$s/\(\([^\,]*\), \(.*\$\\)\)/\2 \1/
```

The Vim Tutorial and Reference

1	First line of the range
\$	Last line of the range
s	:substitute command
/ <from> / <to> /	From and to text for substitution
\(....\)	Put contents into \1
[^,]	Anything except (^) comma
*	Repeat previous zero or more times
\)	End of \1
,	Literal comma and space
\(...\)	Put contents in \2
\2	To text, contents of second \(..\)
\1	To text, contents of first \((...)\)

Figure 16-2: Substitute command to switch words

How to Edit All the Files that Contain a Given Word

If you are a UNIX user, you can use a combination of *Vim* and *grep* to edit all the files that contain a given word. This is extremely useful if you are working on a program and want to view or edit all the files that contain a specified variable.

For example, suppose you want to edit all the C program files that contain the word `frame_counter`. To do this you use the command:

```
$ vim `grep -l 'frame_counter' *.c`
```

Let's look at this command in detail. The *grep* command searches through a set of files for a given word. Because the `-l` option is specified, the command will only list the files containing the word and not print the line itself. The word it is searching for is `frame_counter`. Actually, this can be any regular expression. (Note:What *grep* uses for regular expressions is not as complete or complex as what *Vim* uses.)

The entire command is enclosed in backticks (`).This tells the UNIX shell to run this command and pretend that the results were typed on the command line.

So what happens is that the *grep* command is run and produces a list of files, these files are put on the *Vim* command line. This results in *Vim* editing the file list that is the output of *grep*.

You can then use the commands `:n` and `:rewind` to browse through the files.

The Vim Tutorial and Reference

Why show this here? This is a feature of the UNIX shell (such as 'bash'), and isn't part of *Vim*'s repertoire. The way to accomplish something similar within *Vim*, and which works on Win32 as well is:

```
:args `grep -l 'frame_counter' *.c`
```

which will set the *argument list*, in other words, the files "on the command line."

Finding All Occurrences of a Word Using the Built-in Search Commands

The *Vim* editor has a built-in *grep* command that you can use to search a set of files for a given string. If you want to find all occurrences of `error_string` in all C program files, for example, enter the following command:

```
:vimgrep error_string *.c
```

This causes *Vim* to search for a string (`error_string`) in all the specified files (`*.c`).

The editor will now open the first file where a match is found and position the file on the first matching line.

To go to the next matching line (no matter what file), use the `:cnext` command. To go to the previous match, use the `:cprev` command.

Chapter 17: Topics Not Covered

The *Vim* editor has a tremendously rich set of features. Unfortunately, a few areas are beyond the scope of this book, and therefore are not covered here. These include support for commercial applications that I do not have access to and foreign languages.

This chapter briefly describes the commands omitted from the rest of the book.

Interfaces to Other Applications

The *Vim* editor is designed to interface with many common commercial packages as well as a few Open Source packages. Because I do not have these packages installed, I could not examine and test these commands. Some brief documentation appears here.

Cscope

The *cscope* command is designed to examine a set of C or C++ programs and produce a database containing information about the location of functions and variables in the programs. You can then use the *Cscope* program to query this database to locate where an identifier is defined or used. *Cscope* is available from <http://cscope.sourceforge.net>.

For full information use the following command:

```
:help cscope
```

Cscope-Related Command Reference

:cs arguments
:cscope argument
:scs arguments
:scscope argument
:lcs arguments
:lcscope argument
:cst procedure
:cstag procedure

Handle various activities associated with the <i>Cscope</i> program.
Split the window then handle various activities associated with the <i>Cscope</i> program.
Handle various activities associated with the <i>Cscope</i> program, but use the local list instead of the quick fix list.
Go to the tag in the <i>CScope</i> database named <i>procedure</i> .

```
:set csprg=program  
:set cscopeprg=program  
  
:set cst  
:set cscopetag  
:set nocst  
:set nocscopetag  
  
:set csqf  
:set cscopequickfix  
:set nocsqf  
:set nocscopequickfix  
  
:set csto=flag  
:set cscopetagorder=flag  
  
  
:set csverb  
:set cscopeverbose  
:set nocsverb  
:set nocscopeverbose  
  
  
:set cspc=number  
:set csopepathcomp=number
```

Define the name of the *Cscope* program (Default=**cscope**).

If set, this option causes the commands that do tag navigation (**:tags**, **CTRL-J**, and so on) to use the *Cscope* database rather than tags.

If set, use the quickfix window for *Cscope* output.

Define the search order for the *CScope* tag-searching commands. If *flag* is 0, the default, search the *CScope* database, followed by tags. If *flag* is 1, search tags first.

If set, output error messages occur when *Vim* looks for a *CScope* database and fails to find it. This proves useful when debugging initialization files that try to load a set of databases (default=**nocscopeverbose**).

Sets the number of path components to use for *Cscope* file specifications. (Default = 0.)

MzScheme

The MzScheme interface enables you use PLT Scheme scripts from within *Vim*. For information use the command:

```
:help mzscheme
```

MzScheme Interface Command Reference

```
:[range]mz statement  
:[range]mzscheme statement  
  
:[range]mz <>pattern  
:[range]mzscheme <>pattern  
  
:[range]mzf file  
:[range]mzfile file
```

Execute a single *MzScheme* statement.

Execute the following lines as *MzScheme* statements until a line consisting of only *pattern* is seen.

Execute the *MzScheme* script contained in the given file.

The Vim Tutorial and Reference

```
:set mzquantum={value}  
:set mzq={value}
```

Sets the polling interval to `{value}` milliseconds.

Netbeans

Netbeans is a Java IDE developed by Sun. It's available from <http://www.netbeans.org>. There is a add on to Netbeans called Viex which lets you use Vim as an external editor. See <http://viex.sourceforge.net/>.

Netbeans use the `-nb` command line parameter to start *Vim* and provide Vim with information concerning the connection between the two programs. One of the communications systems involves *Vim* sending hot keys back to Netbeans through the `:nbkey` (`:nbk`) command.

Since all this communication is handled internally normal user will not have to worry about the `-nb` command line parameter or the `:nbkey` command.

OLE

The OLE system is a method by which programs running under Microsoft Windows can communicate with each other. The *Vim* editor can act as an OLE server. This means that you can write Microsoft Windows programs that interface to it. For those of you who know how to write Visual Basic or other Microsoft-based applications, you can find more information by using the command:

```
:help ole-interface
```

Perl

The Perl interface enables you to execute *perl* command from *Vim* and also gives the Perl programs an interface that they can use to access some of *Vim*'s functions. For a complete description of what is available, execute the following command:

```
:help perl
```

Perl Interface Command Reference

```
:pe command  
:perl command  
  
:pe <<pattern  
:perl <<pattern  
  
:{range}perld command
```

Execute a single *perl* command.

Execute the following lines as *perl* commands until a line consisting of only *pattern* is seen.

Execute a perl command on a range of lines.

The Vim Tutorial and Reference

:{range}perldo command The *perl* variable `$_` is set to each line in range.

Python

The Python interface enables you to execute Python statements and programs from within *Vim*. Like Perl, the Python interface provides you with lots of functions and objects that enable you to access parts of the *Vim* editor.

For complete help, execute the following:

```
:help python
```

Python Interface Command Reference

:{range}py statement Execute a single Python statement.
:{range}python statement

Ruby

Ruby is yet another scripting language you can use with Vim. For a complete description of what is available, execute the following command:

```
:help ruby
```

Ruby Interface Command Reference

:rub command Execute a single *Ruby* command.
:ruby command
:rub <<pattern Execute the following lines as *Ruby* commands until a line consisting of only *pattern* is seen.
:ruby <<pattern
:{range}rubyd command Execute a *Ruby* command on a range of lines. The *Ruby* variable `$_` is set to each line in range.
:{range}rubydo command
:{range}rubyf file Execute the *Ruby* program contained in file.
:{range}rubyfile file

Sniff+

Sniff+ is a commercial programming environment. The *Vim* editor enables you to interface with this program. To get complete help on using this programming tool, execute the following command:

The Vim Tutorial and Reference

```
:help sniff
```

Sniff+ Interface Command Reference

:sni command
:sniff command

Perform a command using the interface to Sniff+. If no command is present, list out information on the current connection.

Sun Visual WorkShop

Visual WorkShop is an IDE which has the ability to specify an external editor. Vim supports the Sun Visual WorkShop interface with a number of commands and options. These have been designed to be as general as possible to support other IDE and debugging systems.

- 14 For more information use the command:

```
:help sniff
```

Sun Visual WorkShop reference

- 15

:ws verb
:wsverb verb

Send **verb** to the workshop. (If you are Visual WorkShop expert this will make sense to you. Otherwise not.)

Option '**balloondealy**' ('**bdealy**') The time in ms that a cursor must hover over a word before a help balloon appears.

Option '**ballooneval**' ('**beval**') If set, enables balloon help.

Option '**balloonexpr**' ('**bexpr**') Expression to use to figure out what balloon help to display.

Tcl

Tcl is another scripting language. As usual, *Vim* provides you with a way to execute Tcl scripts as well as an interface for accessing parts of *Vim* from within Tcl. For full information, use the following command:

```
:help tcl
```

Tcl Interface Command Reference

:tc command

Execute a single Tcl command.

:tcl command

- :{range}tcl_d command** Execute a Tcl command once for each line in the range. The variable line is set to the contents of the line.
- :tclf file** Execute the Tcl script in the given file.
- :tclfile file**

Foreign Languages

The *Vim* editor can handle many different types of foreign languages. Unfortunately, the author cannot. Here is a very short listing of the commands available to you for editing in other languages. For complete information, you need to consult the *Vim* documentation and as well as the documentation that came with your computer system. Although *Vim* contains many language-specific options, a few are fairly generic.

<F8>

:set r1
:set rightleft
:set norl
:set norightleft

:set rlc=search
:set rightleftcmd=search

:set ari
:set allowrevins
:set noari
:set noallowrevins

:set ri
:set revins
:set nori
:set norevins

:set gfs=f1,f2
:set guifontset=f1,f2

:set gfw=f1,f2

Toggle between left-to-right and right-to-left modes.

When set, indicates that the file is displayed right to left rather than left to right (default=**norightleft**).

Allow right to left input for the given mode. The only legal mode is **search** which is also the default.

When set, let **CTRL-_-** toggle the **revins** option. This enables you to input languages that run from right to left rather than left to right.

When set, insert mode works right to left rather than left to right. The **CTRL-_-** command toggles this option if the option **allowrevins** is set.

Define a font *f1* for English and another *f2* for a foreign language. This works only if *Vim* was compiled with the '**fontset**' enabled and only applies on UNIX systems.

Specifies a list of fonts to use for

```
:set guifontwide=f1,f2
```

double wide characters.

[When is this used?](#)

```
:set lmap=ch1ch2,ch1ch2  
:set langmap=ch1ch2,ch1ch2  
  
:set ambiwidth=size  
:set ambw=size
```

Define a keyboard mapping for a foreign language.

Set the width of ambiguous characters when using a double byte encoding such as some east Asian fonts. See the *Vim* reference document or just play around with the setting until things look right. Possible values: **single** or **double**.

```
:set bomb  
:set nobomb
```

If set Vim checks for a beginning of file (BOM) marker which tells it how the file is encoded. This marker will also be placed at the beginning of the file when it is written.

```
:set casemap=internal,keepascii  
:set cmp=internal,keepascii
```

Defines the upper and lower case mappings for sorting and other operations. If set to internal, an internal mapping function is used. If **keepascii** is present, the ASCII characters (0x00 0x7F) use the ASCII mapping.

```
:set charconvert=function  
:set ccv=function
```

Define a function to be used for converting one character set to another. If no function is specified the internal function **iconv()** is used.

```
:set termbidi  
:set notermbidi  
:set tbidi  
:set notbidi  
  
:set termencoding={encoding}  
:set tenc={encoding}
```

If set, the terminal is bidirectional.

Set the terminal encoding.

Some languages combine characters. For example, Hebrew combines constants and vowels into one character. When editing such languages, the '**delcombine**' ('**delco**') option controls how deletes works on such characters.

The Vim Tutorial and Reference

If this option is set, then delete (**x**) will delete each part of the combination on its own. If the option is off ('**nodelcombine**') then delete removes the entire character.

Input Method

The following options control the current input method works with *Vim*.

keymap The name of the current keyboard mapping.

kmp

imactivekey Tells *Vim* what the input mode activation key is. The option is of the form **{modifier}-{key}** where the modifier is zero or more of:

- S** Shift key
- L** Lock key
- C** Control key
- 1** Mod1 key
- 2** Mod2 key
- 3** Mod3 key
- 4** Mod4 key
- 5** Mod5 key

imcmdline When this option is set, Vim will always use the input method to enter or edit a command mode command.

imc If set, the input method is never used.

imdisable

imd

imininsert Tell *Vim* when to turn the language dependent input

mapping (See :**lmap** page 425). The possible values are

0 :**lmap** is off and input method is off

1 :**lmap** is on and input method is off

2 :**lmap** is off and input method is on

imsearch Like '**imininsert**', only for searching, not insert.

ims

maxcombine The maximum number of combining characters for display.

mco

Arabic

Arabic is a complex flowing script language which requires special settings to get right.

Option

Meaning

The Vim Tutorial and Reference

<code>:set arabic</code>	Sets other options to make Arabic editing possible.
<code>:set arab</code>	
<code>:set noarabic</code>	
<code>:set noarab</code>	
<code>:set arabicshape</code>	When set make the visual character corrections needed to do Arabic
<code>:set arshape</code>	
<code>:set noarabicshape</code>	
<code>:set noarshape</code>	

Chinese

Written Chinese is a beautiful pictographic language where one character can convey a world of meaning. Unfortunately typing that one character can be extremely difficult given the limits of a computer keyboard. Chinese can be written left to right, right to left, or up to down. The *Vim* editor supports right to left and left to right. It also supports both traditional Chinese characters as well as simplified Chinese.

Note: The *Vim* documentation does not contain help on the subject of Chinese input. That is operating system dependent.

Chinese-Related Command Reference

<code>:set fe=encoding</code>	Set the file encoding to be used for this file. For the Chinese language, this can be taiwan for the traditional Chinese character set or prc for simplified Chinese.
<code>:set fileencoding=encoding</code>	

Farsi

The Farsi language is supported, although you must explicitly enable Farsi when you compile the editor. It is not enabled by default. To edit in a Farsi file, start *Vim* in Farsi mode by using the **-F** option.

For example:

```
$ vim -F file.txt
```

You can get complete information on Farsi editing by executing the following command:

```
:help farsi
```

The Vim Tutorial and Reference

Farsi-Related Command Reference

<code>:set fk</code>	If set, this option tells <i>Vim</i> that you are using a Farsi keyboard (default= <code>nofkmap</code>).
<code>:set fkmap</code>	
<code>:set nofk</code>	
<code>:set nofkmap</code>	
<code>:set akm</code>	
<code>:set altkeymap</code>	When ' <code>altkeymap</code> ' is set, the alternate keyboard mapping is Farsi. If <code>noaltkeymap</code> is set, the default alternate keyboard mapping is Hebrew (default= <code>noaltkeymap</code>).
<code>:set noakm</code>	
<code>:set noaltkeymap</code>	
CTRL-_-	Toggle between Farsi and normal mode (insert-mode command).
<F9>	Toggles the encoding between ISIR-3342 standard and <i>Vim</i> extended ISIR-3342 (supported only in right-to-left mode).

Hebrew

Hebrew is another language that goes right to left. To start editing a Hebrew file, use the following command:

```
$ vim -H file.txt
```

To get help editing in this language, execute the following command:

```
:help hebrew
```

Hebrew-Related Command Reference

<code>:set hk</code>	Turn on (or off) the Hebrew keyboard mapping (default=' <code>nohkmap</code> ').
<code>:set hkmap</code>	
<code>:set nohk</code>	
<code>:set nohkmap</code>	
<code>:set hkp</code>	
<code>:set hkmapp</code>	When set, this option tells <i>Vim</i> that you are using a phonetic Hebrew keyboard, or a standard English keyboard (default=' <code>nohkmap</code> ').
<code>:set nohkp</code>	
<code>:set nohkmap</code>	
<code>:set al=number</code>	Define the numeric value of the first character in the Hebrew alphabet. The value used depends on how your system encodes the Hebrew characters. (Default: Microsoft DOS: 128. Other systems: 224)
<code>:set aleph=number</code>	
CTRL-_-	Toggle between reverse insert and normal insert

The Vim Tutorial and Reference

modes. Hebrew is usually inserted in reverse, like Farsi .

```
:set akm  
:set altkeymap  
:set noakm  
:set noaltkeymap
```

If '**altkeymap**' is set, the alternate keyboard mapping is Farsi. If '**noaltkeymap**' is set, the default alternate keyboard mapping is Hebrew (default='**noaltkeymap**').

Japanese

Japanese-encoded files are supported. Unfortunately there is no specific online help for Japanese.

Japanese-Related Command Reference

```
:set fe=japan  
:set fileencoding=japan
```

Tells *Vim* that the current file is encoded using the Japanese character set.

Korean

To edit in Korean, you need to tell *Vim* about your keyboard and where your fonts reside. You can obtain information on how to do this by executing the following command:

```
:help hangul
```

Korean-Related Command Reference

```
:set fe=korea  
:set fileencoding=korea
```

Tells *Vim* that the current file is encoded using the Korean character set.

Binary Files

Editing binary files using a text editor is tricky at best and suicidal at worst. If you have the right combination of expertise and desperation, you can use *Vim* to edit binary files.

```
:set bin  
:set binary  
:set nobin  
:set nobinary
```

If set, then set things up for editing a binary file.

Note: I realize that some people out there need to edit a binary file and that you know what you are doing. For those of you in that situation, *Vim* is an excellent editor. Fortunately for those who need to, *Vim* comes with a great utility, *xxd* which allows one to edit binary files almost painlessly. See the online docs for more information:

```
:help xxd
```

Modeless (Almost) Editing

If you enable the '**insertmode**' option, insert mode is the default. If you want to switch to normal, use the **CTRL-O** command to execute a normal-mode command. This option is for people who do not like modes and are willing to deal with the confusing setting it generates.

:set im	Make insert mode the default.
:set insertmode	
:set noim	
:set noinsertmode	
CTRL-L	Leave insert mode if ' insertmode ' is set.

Operating System File Modes

Some operating systems keep file type information around on the file. The two operating systems covered in this book do not have this feature. If the OS does determine the file type, the result will be saved in the '**osfiletype**' option.

:set osf=type	This is set to the file type detected by an OS capable of doing so.
:set osfiletype=type	
:set st=type	Define the shell type for an Amiga.
:set shelltype=type	

16

Part II

17

Reference

Chapter 18: Complete Basic Editing

In *Chapter 2: Editing a Little Faster*, you were introduced to some of the basic editing commands. You can do 90% of most common edits with these commands. If you want to know everything about basic editing, however, read this chapter.

Chapter 2, for example, discussed the **w** command to move forward one word. This chapter devotes a page to discussing in detail how to define exactly what a word is. *Chapter 2* described how to use two commands (**CTRL-D** and **CTRL-U**) to move screen up and down through the text. There are actually six commands to do this, with options, and that does not include the other dozen or so positioning commands. This chapter discusses the complete command list.

Generally you will not use all the commands in this chapter. Instead you will pick out a nice subset you like and use them. *Chapter 2* presented one subset. If you prefer to pick your own, however, read this chapter and have fun.

Word Movement

The *Vim* editor has many different commands to enable you move to the beginning or end of words. But you can also customize the definition of a word through the use of some of the *Vim* options. The following sections explore in depth the word movement commands.

Move to the End of a Word

The **w** command moves forward one word. You actually wind up on the beginning of the next word. The **e** command moves forward one word, but leaves you on the end of the word.

The **ge** command moves backward to the end of the preceding word. Figure 18-1 shows how the various word-movement commands work.

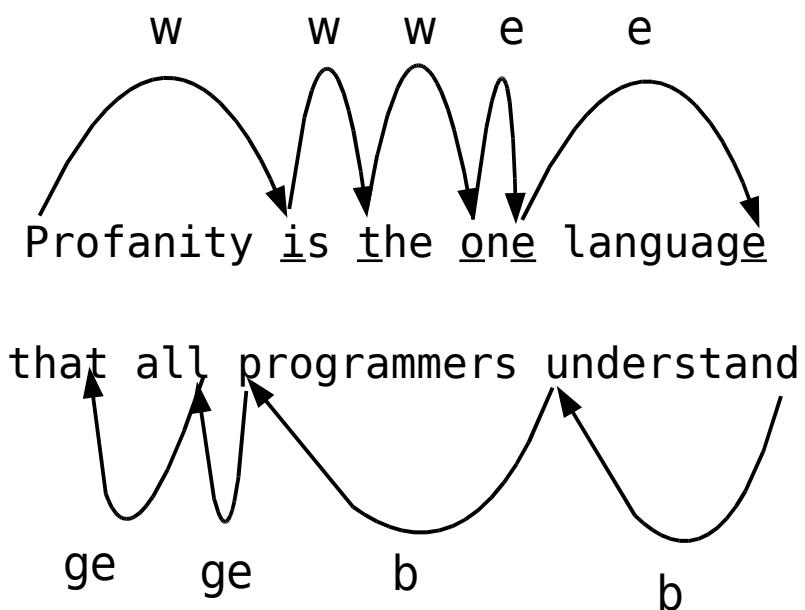


Figure 18-1: Word movement commands.

Defining What a Word Is

So what is a word, anyway? There are two answers to this question:

1. The *Vim* editor does a good job of defining a sane answer to this question. (Skip to the next section.)
2. If you want to know the details, read the rest of this section.

Your first answer to "What is a word?" might be the easy answer: It is a series of letters. However, a C programmer might consider something like `size56` to be word. Therefore, another answer might be to use the C identifier definition: the letters, digits, and the underscore.

But LISP programmers can use the dash (-) in a variable name. They consider the word `total-size` a single word. C programmers consider it two words. So how do you resolve this conflict?

The *Vim* solution is to create an option that defines what is in a word and what is not. The following command defines the characters that belong in a word:

```
:set iskeyword=specification
```

The Vim Tutorial and Reference

Note: You can abbreviate the '`iskeyword`' option as '`isk`'.

To see what the current value of this option is, use this command:

```
:set iskeyword?
```

The following represents a typical value:

```
iskeyword=@,48-57,_,192-255
```

This is all the letters (@), the digits (ASCII characters numbered **48-57** or **0-9**), the underscore (_) and the international letters (**192-255**, à through ý).

The specification consists of characters separated by commas. If you want the word characters to be exclusively vowels, for instance, use this command:

```
:set iskeyword=a,e,i,o,u
```

You can specify a range of characters by using a dash. To specify all the lowercase letters, for example, issue the following command:

```
:set iskeyword=a-z
```

For characters that cannot be specified directly (such as comma and dash), you can use a decimal number. If you want a word to be the lowercase letters and the dash (character #45), use the following command:

```
:set iskeyword=a-z,45
```

The @ character represents all characters where the C function `isalpha()` returns true. (This might vary, depending on the setting of the C locale, and also depending on the C compiler and OS you are using to build *Vim*!)

To exclude a character or set of character, precede it with a circumflex (^). The following command defines a word as all the letters except lowercase q:

```
:set iskeyword=@,^q
```

The @ character is represented by @-@.

Special Characters for the `iskeyword` Option

a	Character a .
a-z	Character range (all characters from a to z).
45	Character number 45 (in this case, -).
@	All letters (as defined by <code>isalpha()</code>).
@-@	The character @.
^x	Exclude the character x .
^a-c	Exclude the characters in the range a through c .

Other Types of Words

The '**iskeyword**' option controls what is and is not a keyword. Other types of characters are controlled by similar options, including the following:

- '**isfname**' Filenames. (Abbreviation '**isf**').
- '**isident**' Identifiers. (Abbreviation '**isi**').
- '**isprint**' Printing characters. (Abbreviation '**isf**').

The '**isfname**' option is used for commands such as the **gf** command, which edits the file whose name is under the cursor. (See *Chapter 23: Advanced Commands for Programmers*, for more information on this command.)

The '**isident**' option is used for commands such as **[d**, which searches for the definition of a macro whose identifier is under the cursor. (See *Chapter 7: Commands for Programmers*, for information on this command.)

The '**isprint**' option defines which characters can be displayed literally on the screen. Careful: If you get this wrong the display will be messed up. This is also used by the special search pattern **\p**, which stands for a printable character. (See *Chapter 19: Advanced Searching Using Regular Expressions*, for information on search patterns.)

There Are "words," and Then There Are "WORDS"

So now you know what words are, right? Well, the *Vim* editor also has commands that affect WORDS. These two terms, WORDS and words, represent two different things. (Only a programmer could think up terms that differ only by case.)

The term word means a string of characters defined by the '**iskeyword**' option. The term WORD means any sequence of non-whitespace characters. Therefore

that-all

is three words("that", "-", and "all"), but it is one WORD. The **w** command moves forward WORDS and the **B** command moves backward WORDS. The complete list of WORD-related commands is as follows:

- | | |
|------------------------------|--|
| [count]<C-Left> | |
| [count] B | Move <i>count</i> WORDS backward. |
| [count] E | Move <i>count</i> WORDS forward to the end of the WORD. |
| [count] gE | Move <i>count</i> WORDS backward to the end of the WORD. |

[count]<C-Right>

[count] w Move *count* WORDS forward.

Note: <C-Left> is the same as **CTRL<LEFT>**. See *Appendix B: The <> Key Names* for a list of <> key names.

Beginning of a Line

The ^ command moves you to the first non-blank character on the line. If you want to go to the beginning of the line, use the 0 command. Figure 18-2 shows these commands.

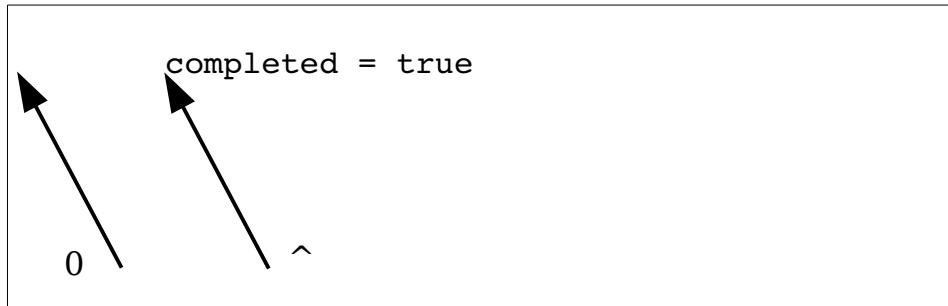


Figure 18-2: ^ and 0 commands.

Repeating Single-Character Searches

The **f**x command searches for the first x after the cursor on the current line. To repeat this search, use the ; command. As usual, this command can take an argument that is the number of times to repeat the search.

The ; command continues the search in the same direction as the last f or F command. If you want to reverse the direction of the search, use the , command. Figure 18-3 shows several typical searches.

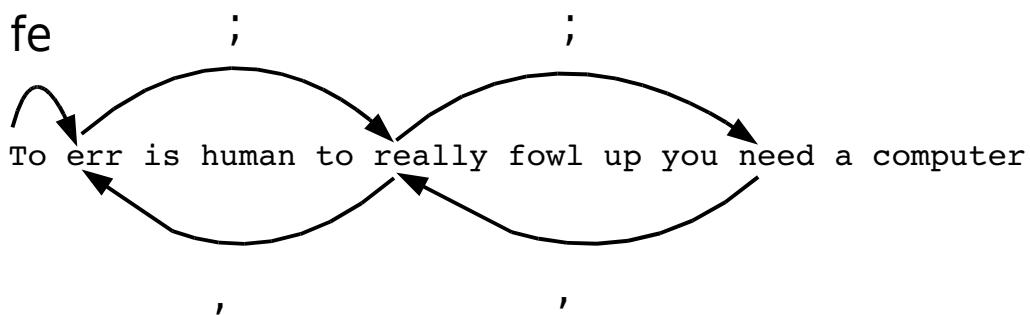


Figure 18-3: Repeat single-character search.

Moving Lines Up and Down

The **-** command moves up to the first non-blank character on the preceding line. If an argument is specified, the cursor moves up that many lines.

The **+** (<Cr>, <Enter>, **CTRL-M**) command moves down to the beginning of the next line. If an argument is specified, the cursor moves down that number of lines. Figure 18-4 shows these commands.

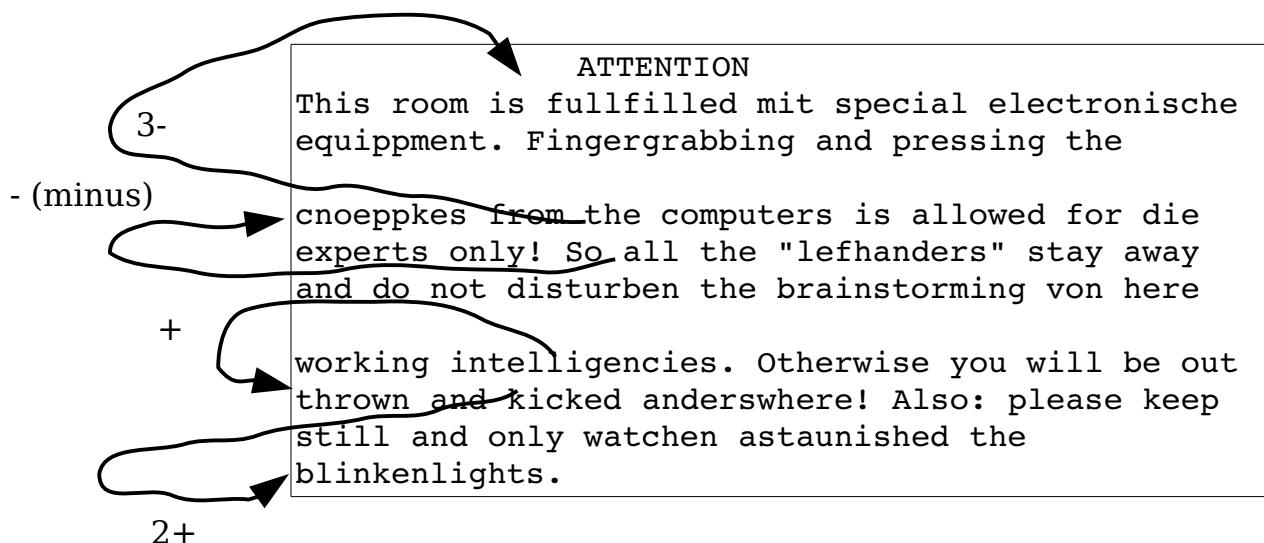


Figure 18-4: The **+** and **-** commands.

The **_** command moves to the first non-blank character of the line. If a count is specified, it moves the first character of the count-1 line below the cursor.

Cursor-Movement Commands

Several commands enable you to move to different parts of the screen. The **H** command moves to the top of the screen. If a count is specified, the cursor will be positioned to the count line from the top. Therefore, **1H** moves to the top line, **2H** the second line, and so on.

The **L** command is just like **H** except that the end of the screen is used rather than the start.

The **M** command moves to the middle of the screen. Figure 18-5 summarizes the cursor-positioning commands.

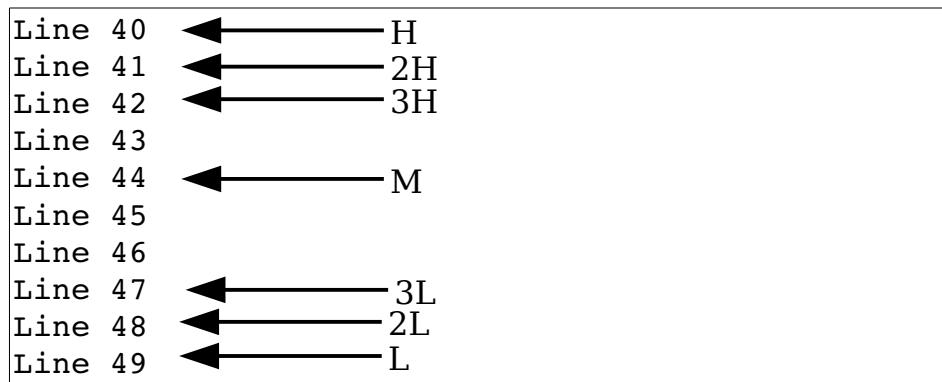


Figure 18-5: Cursor-positioning commands.

Jumping Around

The *Vim* editor keeps track of where you have been and enables you to go back to previous locations. Suppose, for example, that you are editing a file and execute the following commands:

1G	Go to line 1
10G	Go to line 10
20G	Go to line 20

Now, you execute this command:

```
:jumps
```

(The short version is :ju.)

You get the following:

The Vim Tutorial and Reference

```
jump line col file/text
 2 1    0   Dumb User Stories
 1 10   0   ventilation holes. Many terminals
>
```

From this you see that you have recorded as jump 1, line 10, column 0, the "ventilation holes" line. Jump 2 is at line 1, column, the "Dumb User Stories" line.

Line 20 is not recorded in the jump list yet, because you are on it. The jump list records only things after you jump off of them. The > points to the current item in the list; in this case, it points to the blank line at the end indicating an unrecorded location.

Now that you know what the jump list is, you can use it. The **CTRL-O** command jumps back one line. Executing this command takes you back to line 10. The jump list now looks like this:

```
jump line col file/text
 2 1    0   Dumb User Stories
> 0 10   0   ventilation holes. Many terminals
 0 20   0
```

The > has moved up one line. If you use the **CTRL-O** command again, you move to line 1. The **CTRL-I** or **<TAB>** command moves you to the next jump in the list. Thus you have the following:

- 1G** Go to line 1.
- 10G** Go to line 10.
- 20G** Go to line 20.
- CTRL-O** Jump to previous location (line 10).
- CTRL-O** Jump to previous location (line 1).
- <TAB>** Jump to next location (line 10).

Using these commands, you can quickly navigate through a series of jumping off points throughout your file.

Using the Change List

Error: Reference source not foundThe jump list keeps track of where the cursor has been. The change list does something similar but only records the location of the cursor when a change is made. The does the same thing, only it's limited to cursor locations where something changed.

The Vim Tutorial and Reference

To go to the previous change, use the command **g;**. This can be proceeded with a count to go back multiple changes. To go to the next change, use the command [**count**]g, (**g<comma>**).

To get a list of changes use the command :**changes**.

Controlling Some Commands

Normally *Vim* stops any left or right movement at the beginning or end of the line. The '**whichwrap**' ('**ww**')option controls which characters are allowed to go past the end and in which modes. The possible values for this option are as follows:

Character	Command	Mode(s)
b	<BS>	Normal and visual
s	<Space>	Normal and visual
h	h	Normal and visual
l	l	Normal and visual
<	<Left>	Normal and visual
>	<Right>	Normal and visual
~	~	Normal
[<Left>	Insert and replace
]	<Right>	Insert and replace

Figure 18-6 shows how '**whichwrap**' affects cursor movement.

'whichwrap' no b
<bs> stops here

Carelessly planned projects take three time
longer to complete than expected. Carefully
planned projects take four times longer to
complete than expected, mostly because the
planners expect their planning to reduce the
time it takes.

'whichwrap'
with b
<bs> wraps

Figure 18-6: Effects of the '**whichwrap**' option.

Where Am I, in Detail

The **CTRL-G** command displays summary information at the bottom of the screen telling you where you are in the file (see *Chapter 2: Editing a Little Faster*). However, you can get more detailed information if you ask. The basic **CTRL-G** output looks like this:

The Vim Tutorial and Reference

```
"c02.txt" [Modified] line 81 of 153 --52%-- col 1
```

To get more information, give **CTRL-G** a count. The bigger the count, the more detailed information you get. The **1CTRL-G** command gives you the full path of the file, for example:

```
"/usr/c02.txt" [Modified] line 81 of 153 --52%-- col 1
```

The **2CTRL-G** command lists a buffer number as well. (You can read more on buffers in *Chapter 5: Windows and Tabs*)

```
buf 1: "/usr/c02.txt" [Modified] line 81 of 153 --52%-- col 1
```

The **gCTRL-G** command displays another type of status information indicating the position of the cursor in terms of column, line, and character:

```
Col 1 of 0; Line 106 of 183; Char 3464 of 4418
```

If you are interested in having the current cursor location displayed all the time, check out the '**ruler**' ('**ru**') option in *Chapter 28: Customizing the Editor*.

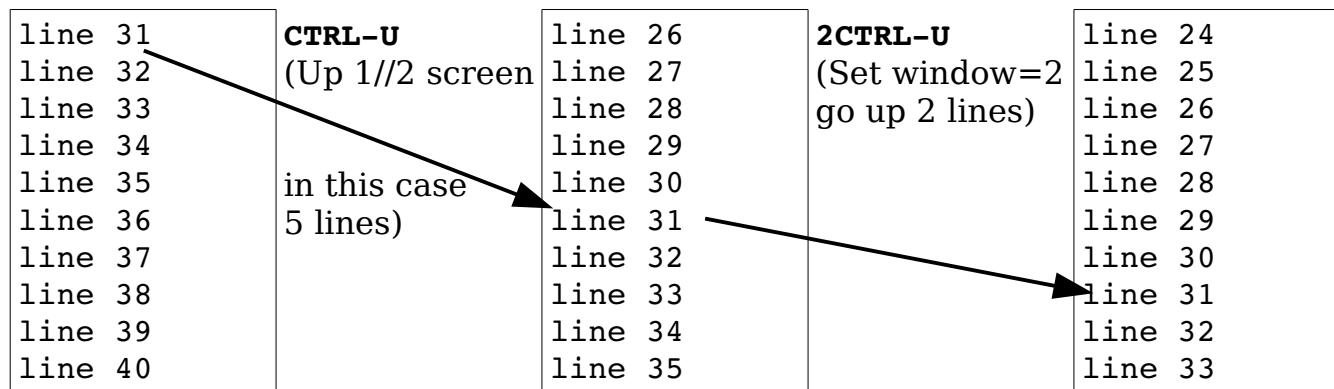
Scrolling Up

As discussed in *Chapter 2: Editing a Little Faster*, the **CTRL-U** command scrolls up half a screen.

To be precise, the **CTRL-U** command scrolls up the number of lines specified by the '**scroll**' option. You can explicitly set this option with a :set command:

```
:set scroll=10
```

You can also change the value of this option by giving an argument to the **CTRL-U** command. For example, **2CTRL-U** changes the scroll size to 2 and moves up 2 lines. All subsequent **CTRL-U** commands will only go up 2 lines at a time, until the scroll size is changed. Figure 18-7 shows the operation of the **CTRL-U** command.



The Vim Tutorial and Reference

Figure 18-7: The **CTRL-Y** command.

To scroll the window up one line at a time, use the **CTRL-Y** command. This command can be multiplied by an argument. For example, **5CTRL-Y** scrolls up 5 lines (see Figure 18-8).

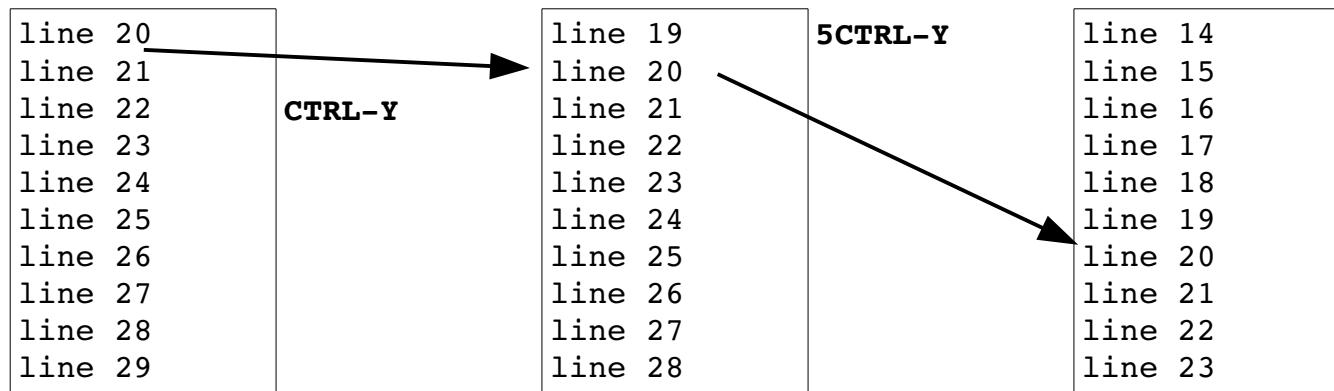


Figure 18-8: The **CTRL-Y** command.

The **CTRL-B** command scrolls up an entire screen at a time (see Figure 18-9).

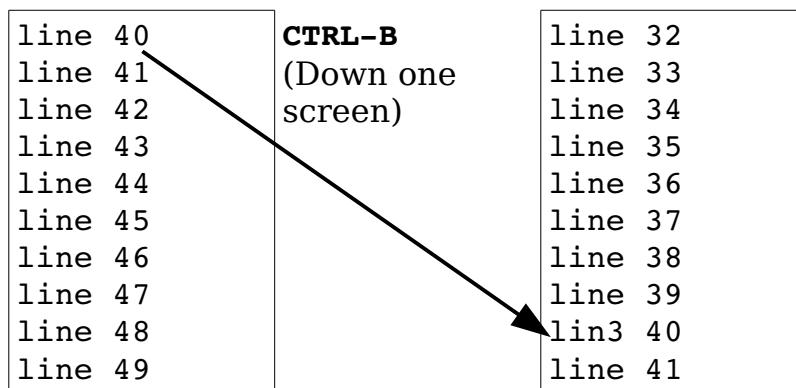


Figure 18-9: The **CTRL-B** command.

You can specify this command as **<PageUp>** or **<S-Up>**. (**<S-Up>** is the *Vim* notation for the Shift+up-arrow key.)

Actually the **CTRL-B** command scrolls up the number of lines specified by the '**window**' ('**wi**') option. By default, this is an entire screen. So by change the value of this option you can customize the command.

Scrolling Up Summary

Figure 18-10 illustrates the various scrolling commands. Commands move the top line to the indicated location.

Commands will scroll the screen making the indicated line the top line.

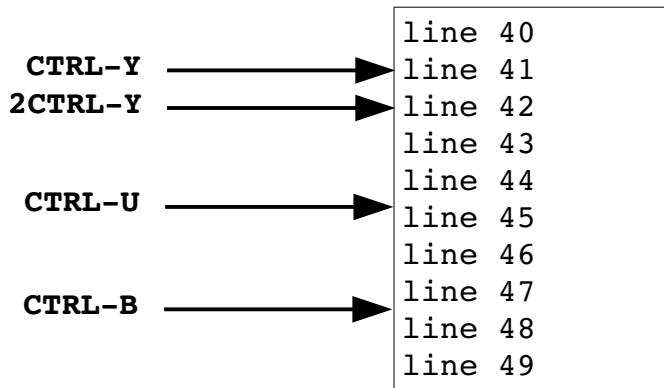


Figure 18-10: Scrolling commands.

Scrolling Down

There are similar commands for moving down as well, including the following:

- CTRL-D** Move down. The amount is controlled by the '**scroll**' option.
- CTRL-E** Move down one line.
- CTRL-F** Move down one screen of data (also **<PageDown>** or **<S-Down>**).

Figure 18-11 summarizes the scrolling commands.

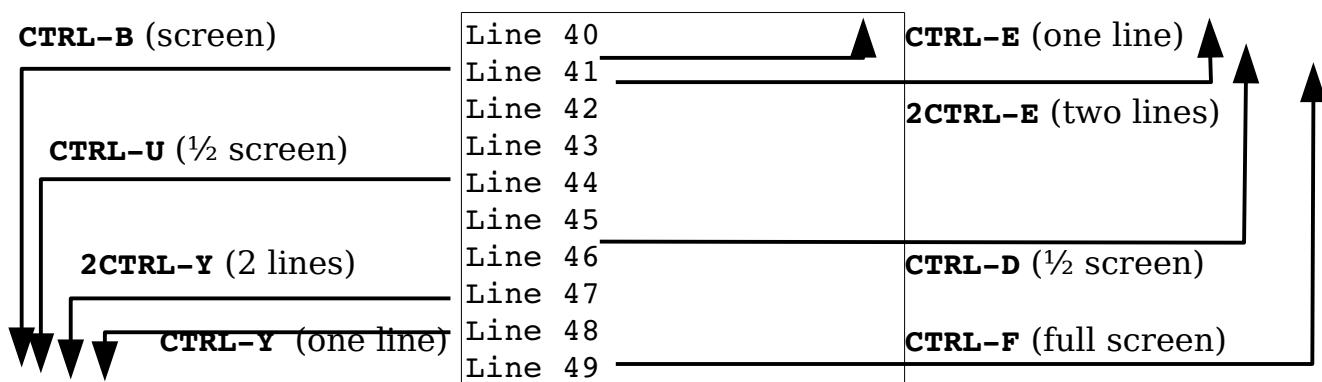


Figure 18-11: More scrolling commands.

The Vim Tutorial and Reference

Both **CTRL-B** and **CTRL-F** scroll a full screen by default. You can adjust the amount of scrolling by setting the '**window**' ('**wi**') option to the number of lines you want to scroll.

Define How Much to Scroll

When you move the cursor off the top or bottom, the window scrolls. The amount of the scrolling is controlled by the '**scrolljump**' ('**sj**') option. By default this is a single line; if you want more scrolling, however, you can increase this to a jump of 5, as follows:

```
:set scrolljump=5
```

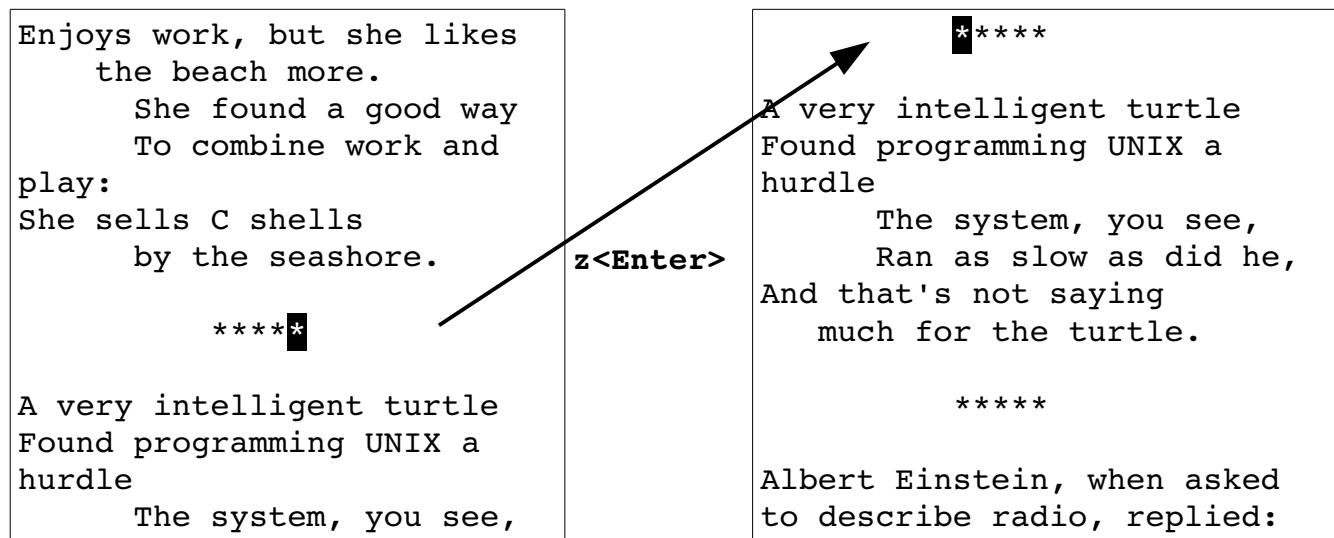
The '**sidescroll**' ('**ss**') option does the same thing, except in the horizontal direction. Usually the cursor must reach the top or bottom line of the screen for scrolling to occur. If you want to add a little padding to this margin, you can set the '**scrolloff**' ('**so**') option. To make sure that there are at least 3 lines above or below the cursor, use the following command:

```
:set scrolloff=3
```

The option '**sidescrolloff**' ('**siso**') does the same thing only horizontally.

Adjusting the View

Suppose you want a given line at the top of the screen. You can use the **CTRL-E** (up one line) and **CTRL-Y** (down one line) commands until you get the proper line at the top. Or you can position the cursor on the line and type the command **z<Enter>**. Figure 18-12 shows how this command changes the screen.



The Vim Tutorial and Reference

```
Ran as slow as did he,  
And that's not saying  
much for the turtle.  
  
*****
```

```
"You see, wire telegraph is  
a kind of a very, very long  
cat. You pull his tail in  
New York and his head is  
meowing in Los Angeles.  
Do you understand this?
```

Figure 18-12: The `z<Enter>` command.

If you supply an argument to this command, it will use that line rather than the current one. `z<Enter>` positions the current line at the top of the screen, for instance, whereas `88z<Enter>` positions line 88 at the top.

The `z<Enter>` command not only positions the line at the top of the screen, it also moves the cursor to the first non-blank character on the line. If you want to leave the cursor where it is on the line, use the command `zt`. (If you change the current line by giving the command an argument, *Vim* will try to keep the cursor in the same column.) Figure 18-13 shows the `zt` command.

```
Enjoys work, but she likes  
the beach more.  
She found a good way  
To combine work and  
play:  
She sells C shells  
by the seashore.  
  
*****  
  
A very intelligent turtle  
Found programming UNIX a  
hurdle  
The system, you see,  
Ran as slow as did he,  
And that's not saying  
much for the turtle.
```

`zt`

```
A very intelligent turtle  
Found programming UNIX a  
hurdle  
The system, you see,  
Ran as slow as did he,  
And that's not saying  
much for the turtle.  
  
*****  
  
Albert Einstein, when asked  
to describe radio, replied:  
"You see, wire telegraph is  
a kind of a very, very long  
cat. You pull his tail in  
New York and his head is  
meowing in Los Angeles.  
Do you understand this?
```

Figure 18-13: The `zt` command.

If you want to position a line to the end of the screen, use the `zb` or `z-` command. The `z-` positions the cursor on the first non-blank column, whereas `zb` leaves it alone. Figure 18-14 shows the effects of these commands.

```
the beach more.
```

```
Equals nine squared
```

The Vim Tutorial and Reference

<p>She found a good way To combine work and play: She sells C shells by the seashore.</p> <p>*****</p> <p>A very intelligent turtle Found programming UNIX a hurdle The system, you see, Ran as slow as did he, And that's not saying much for the turtle.</p>	<p>plus zero, no more. *****</p> <p>A UNIX sales lady, Lenore, Enjoys work, but she likes the beach more. She found a good way To combine work and play: She sells C shells by the seashore.</p> <p>*****</p> <p>A very intelligent turtle</p>
--	--

Figure 18-14: The **zb** and **z-** commands.

Finally, the **zz** and **z.** commands position the line at the center of the window. The **zz** command leaves the cursor in its current column, and **z.** moves it to the first nonblank column. Figure 18-15 shows what these commands do.

<p>Equals nine squared plus zero, no more. *****</p> <p>A UNIX sales lady, Lenore, Enjoys work, but she likes the beach more. She found a good way To combine work and play: She sells C shells by the seashore.</p> <p>*****</p> <p>A very intelligent turtle</p>	<p>She found a good way To combine work and play: She sells C shells by the seashore.</p> <p>*****</p> <p>A very intelligent turtle Found programming UNIX a hurdle The system, you see, Ran as slow as did he, And that's not saying much for the turtle.</p> <p>*****</p> <p>Albert Einstein, when asked</p>
--	--

Figure 18-15: The **z.** and **zz** commands.

Delete to the End of the Line

The **D** command deletes to the end of the line. If preceded by a count, it deletes to the end of the line and count-1 more lines. (The **D** command is shorthand for **d\$**.) See Figure 18-16 for some examples of this command.

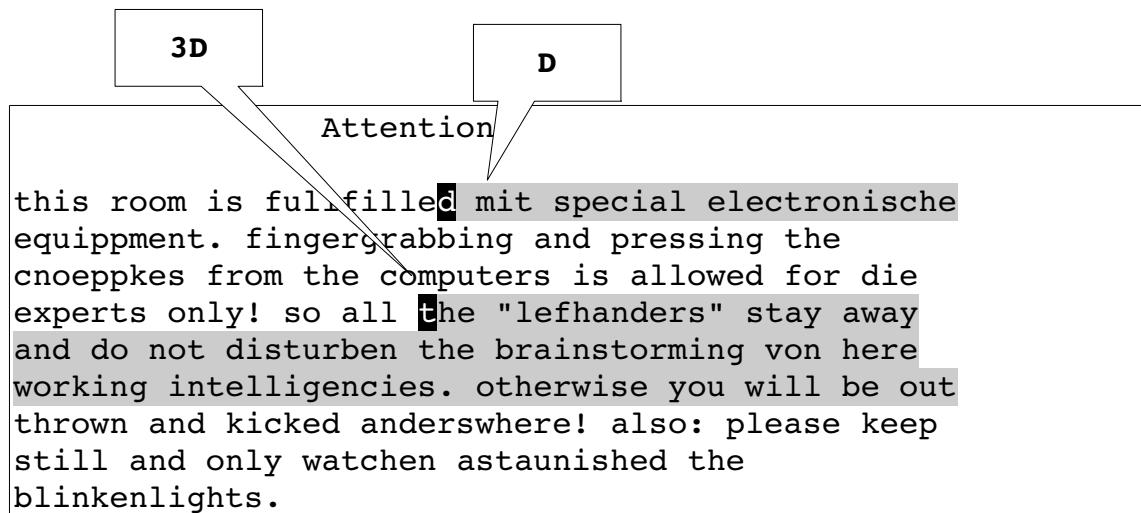


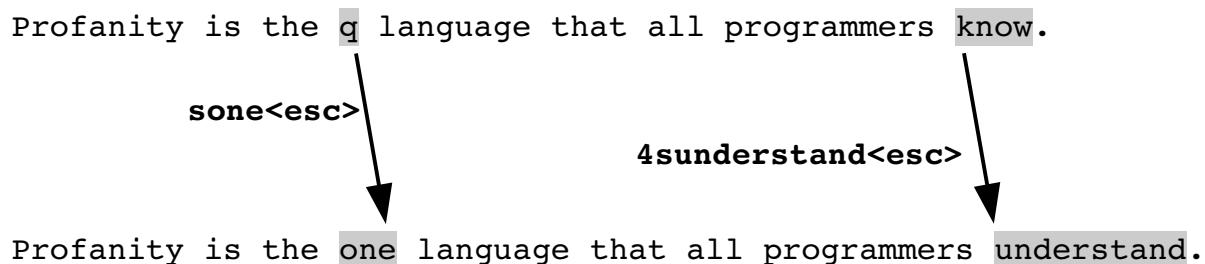
Figure 18-16: The **D** command.

The **C** Command

The **C** command deletes text from the cursor to the end of the line and then puts the editor in insert mode. If a count is specified, it deletes an additional count-1 lines. In other words, the command works like the **D** command, except that it puts you in insert mode.

The **s** Command

The **s** (substitute) command deletes a single character and puts the editor in insert mode. If preceded by a count, then count characters are deleted. Figure 18-17 illustrates this command.



*Figure 18-17: The **s** command.*

The S Command

The **s** command deletes the current line and puts the editor in insert mode. If a count is specified, it deletes count lines. This differs from the **c** command in that the **c** command deletes from the current location to the end of the line, whereas the **s** command always works on the entire line. Figure 18-18 illustrates the use of the **s** command.

Text affected by C	Text affected by S
Over the years system installers have developed many different ways to string cables above false ceilings. One of the more innovative is the "small dog" method. One person takes a small dog, ties a string to its collar and puts the dog in the ceiling. The owner then goes to the spot where they want the cable to come out and calls the dog. Dog runs to owner. The attach a cable to the string, pull it through, and the cable is installed.	2C S (Even if the cursor is in the middle of the line) 2S

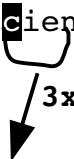
*Figure 18-18: The **s** command.*

Deleting Text

The [*count*] **x** command deletes characters starting with the one under the cursor moving right. The **x** command deletes characters to the left of the cursor. Figure 18-19 shows how these two commands work.

The Vim Tutorial and Reference

1 is equal to 2 for sufficiently large values of 1.



1 is equal to 2 for suffintly large values of 1.

1 is equal to 2 for sufficiently large values of 1.



1 is equal to 2 for suciently large values of 1.

Figure 18-19: The **x** and **X** commands.

Insert Text at the Beginning or End of the Line

The **I** command inserts text like the **i** command does. The only difference is that the **I** command inserts starting at the beginning of the line. (In this case, "beginning" means at the first non-blank character.) To insert at the first character of the line (space or not), use the **gI** command. The **A** command appends text like the **a** command, except the text is appended to the end of the line.

Arithmetic

The *Vim* editor can perform simple arithmetic on the text. The **CTRL-A** command increments the number under the cursor. If an argument is specified, that number is added to the number under the cursor. Figure 18-20 shows how various types of numbers are incremented.

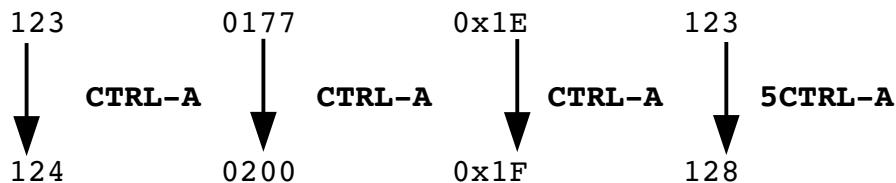


Figure 18-20: Incrementing.

The Vim Tutorial and Reference

If a number begins with a leading 0, it is considered an octal number. Therefore, when you increment the octal number 0177, you get 0200. If a number begins with 0x or 0X, it is considered a hexadecimal number. That explains 0x1E to 0x1F.

The *Vim* editor is smart about number formats, so it will properly increment decimal, hexadecimal, and octal.

The **CTRL-X** command works just like the **CTRL-A** command, except the number is decremented; or if an argument is present, the number is subtracted from the number. Figure 18-21 shows how to decrement numbers.

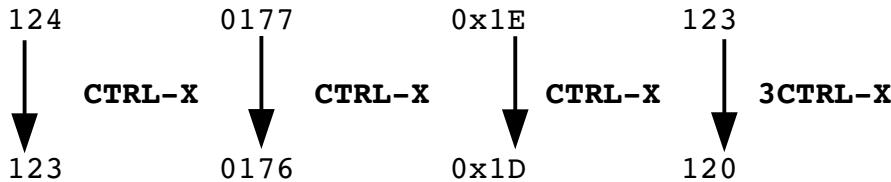


Figure 18-21: Decrementing.

By default, *Vim* recognizes the octal and hexadecimal numbers. Which formats are recognized is controlled by the '**nrformats**' ('**nf**') option. If you want to recognize just decimal numbers, for instance, execute the following command:

```
:set nrformats=""
```

If you want to recognize octal and decimal numbers, use this command:

```
:set nrformats=octal
```

Note: Decimal is always recognized. Unlike hexadecimal and octal, there is no way to turn off decimal recognition.

The default recognizes decimal, hexadecimal, and octal:

```
:set nrformats=octal,hex
```

The *Vim* editor can do more sophisticated calculations. See *Chpater 27: Expressions and Functions* for information on the "**=** register.

Joining Lines with Spaces

The **J** command joins the current line with the next one. A space is added to the end of the first line to separate the two pieces that are joined. But suppose you do not want the spaces. Then you use the **gJ** command to join lines without spaces (see Figure 18-22). It works just like the **J** command, except that no space is inserted between the joined parts.

The Vim Tutorial and Reference

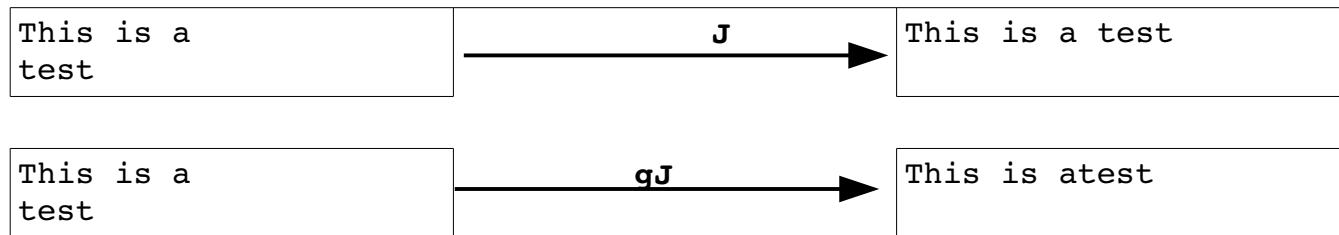


Figure 18-22: The **gJ** command.

Note: If the first line ends with some trailing spaces, the **gJ** command will not remove them.

Replace Mode

The **R** command causes *Vim* to enter replace mode. In this mode, each character you type replaces the one under the cursor. This continues until you type **<Esc>**. Figure 18-23 contains a short example.

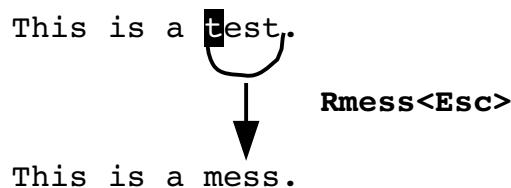


Figure 18-23: The **R** command.

If a count is specified, the command will be repeated count times (see Figure 18-24).

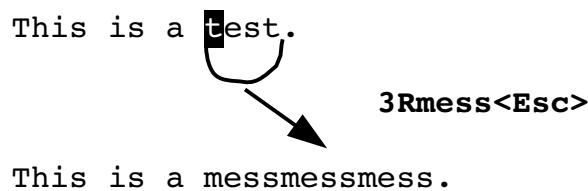


Figure 18-24: **R** command with a count.

You may have noticed that this command replaced 12 characters on a line with only 5 left on it. The **R** command automatically extends the line if it runs out of characters to replace.

Virtual Editing

Normally *Vim* treats tab as a single character. This means when you move the cursor across a line containing the cursor will “jump” from one position to the other as it moves across the tab. Figure 18-25 shows how this works.

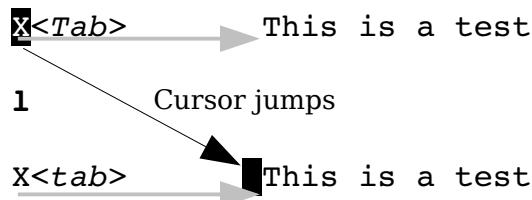


Figure 18-25: Normal cursor movement

You can enable virtual mode by setting the '**virtualedit**' ('**ve**') option to **all**. Now when you move you left you move one character left on the screen. In other words, the cursor does not jump. See Figure 18-26.)

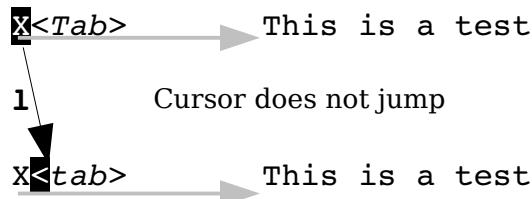


Figure 18-26: Normal cursor movement

Virtual editing also lets you move the cursor past the end of line. This makes thing because the \$ command moves to the end of line, and in some cases the \$ command could move the cursor to the *left!*.

One of the problems with replace in normal (non-virtual) mode is where you have a <Tab> in the text. If you are sitting on a <Tab> and execute the command **rx**, the <Tab> will be replaced by x. This can shift your line around (see Figure 18-27).

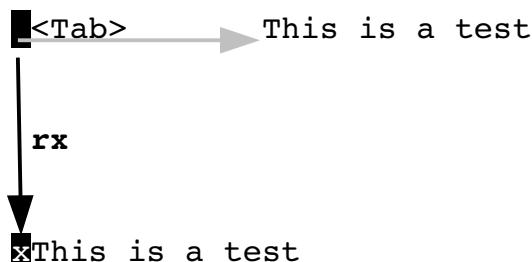


Figure 18-27: Simple non-virtual replace.

With virtual editing *Vim* is smart enough to figure out how to replace a single character in screen space, preserving the spacing.

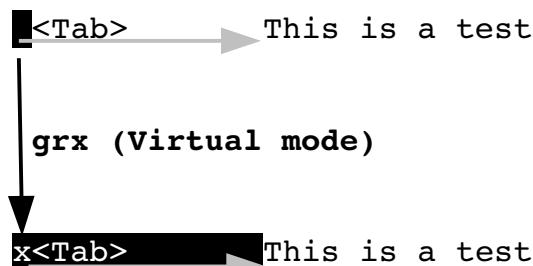


Figure 18-28: Virtual replacement.

The '**virtualedit**' option is actually a comma separated list of values telling *Vim* which modes to edit virtually. The possible value are:

- all** All modes
- block** Block visual editing mode
- insert** Insert mode
- onemore** Allow the cursor to move only one character past the end of line.

When '**virtualedit**' include the **all** value, then normal cursor movement commands move as if <tabs> were spaces. You can also move past the end of the physical line. Inserts can occur anywhere, *Vim* will add spaces as need to make the file match what you've put on the screen.

The **block** keyword tells *Vim* to allow virtual movement in block visual mode. All other modes will act normally. If you wish to do virtual editing in insert mode, you need the **insert** keyword. These options can be combined if you want both modes:

```
:set virtualedit=block,insert
```

Finally there is the **onemore** keyword. If this is not set, you move the cursor to the right as far as you want to. There is no limit. However, if the **onemore** keyword is present, right movement is limited to character past the end of line.

Replace Mode

If you use the virtual replace command, **gr{character}**, you replace the "virtual character" under the cursor (see Figure 18-28) regardless of the state of the '**virtualedit**' option. If the real character under the cursor is part of a tab, only the space representing the tab jumped over is replaced. The **gR** command enters virtual replace mode. Each character you type will replace one character in screen space until you finish things off with <Esc>.

The Vim Tutorial and Reference

The **:startgreplace** (**:startg**) command does the same thing as the **gR** command. If the override (!) option is present, the command acts like **\$gR**.

Digraphs

As learned in *Chapter 2: Editing a Little Faster*, executing **CTRL-K character1 character2** inserts a digraph. You can define your own digraphs by using the following command:

```
:digraphs character1 character2 number
```

(**:dig** for short.)

This tells *Vim* that when you type **CTRL-K character1 character2** that you should insert the character whose character number is **number**. If you are entering a lot of digraphs, you can turn on the '**digraph**' option by using this command:

```
:set digraph
```

(The option can be abbreviated as 'dig')

This means that you can now enter digraphs by using the convention **character1<BS>character2**. (<BS> is the backspace character.)

This mode has its drawbacks, however. The digraph **c<BS>o** is the copyright character (©). If you type **x** but want to type **y**, you can correct your mistake by typing **x** (oops), <BS>, and **y**. If you do that with **c** and **o**, however, it does not erase the **c** and put in the **o**; instead, it inserts ©.

Therefore, you need to type **c** (oops), <BS>, **o** (darn, © appeared), <BS>, and **o**.

To turn off digraph mode, use the following command:

```
:set nodigraph
```

Changing Case

The ~ command changes a character's case. The behavior of the ~ command depends on the value of the '**tildeop**' ('**top**') option. With the option unset, the command behaves normally:

```
:set notildeop
```

If you set the following option, however, the syntax of the command changes to **~motion**:

```
:set tildeop
```

For example the command **~fq** changes the case of all the characters up to and including the first **q** on the line. Figure 18-29 shows some examples.



Figure 18-29: **~motion** commands.

Error: Reference source not found
The **g~motion** command changes the case of the indicated characters. It is just like the **~motion** command except that it does not depend on the '**tildeop**' option. Figure 18-30 shows some examples.



Figure 18-30: The **g~** command.

A special version of this command, **g~~** or **g~g~**, changes the case of the entire line (see Figure 18-31).

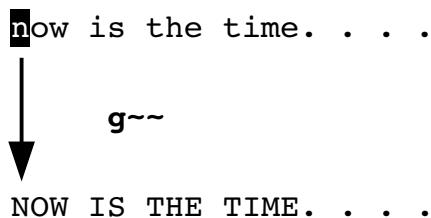


Figure 18-31: The **g~~** command.

Other Case-Changing Commands

The **gU{motion}** command makes the text from the cursor to motion all uppercase. The command **guu** or **gugu** works on a single line. If count is specified, count lines are changed.

The **gumotion**, **guu**, and **gugu** act just like their **gu** counterparts, except that they make the text lowercase. Figure 18-320 illustrates these commands.

This is Mixed Case line.	<u>g~~</u>	THIS IS mIXED cASE LINE.
This is Mixed Case line.	<u>gUU</u>	THIS IS MIXED CASE LINE.

Figure 18-32: Case-changing commands.

Advanced Undo

Vim has an advanced undo system that lets you forget about your mistakes in a lot of different ways.

Undo Time Machine

Let's say that you are working on a program starting at 9:00 in the morning. You try something new and make lots of changes to your program implementing the brand new methodology that your friend just told you about.

About 10:00 you figure out that what you just did was total garbage and won't work. But you've made all those changes. How many times do you have to press **u** (undo) to get back to the 9:00 state?

The answer is none. All you have to do is to tell the Vim you want the file you had an hour ago. This is done with the **:earlier (:ea)** command.

```
:earlier 1h
```

Time for the **:earlier** command can be specified as a number of seconds (**s**), minutes (**m**), or hours (**h**). If no unit is specified it is the number of change sets (see below for the definition of this term.)

Of course if you realize that you went back to far, you can always go forward with the **:later (:lat)** command.

Undo Level

You can execute only so many undo commands. This limit is set by the **'undolevels' ('ul')** option. To set this limit to 5,000 changes, use the following command:

```
:set undolevels=5000
```

Change Sets and Branching

To understand undo branching, we'll start with an example. Figure 18-33 shows the original file:

```
Change A  
Change B  
Change C  
Change D  
Change E  
Change F
```

Figure 18-33: Original file

We now execute the following commands:

1. Create the file, save it, and exit *Vim* and start a new editing session.
2. Move to the "A" line, delete the word "Change"
3. Move to the "B" line, delete the word "Change"
4. Move to the "C" line, delete the word "Change"
5. Move to the "D" line, delete the word "Change"
6. Undo change #5 with the u command.
7. Undo change #4 with the u command.
8. Move to the "E" line, delete the word "Change"
9. Move to the "F" line, delete the word "Change"

The resulting file appears in Figure 18-34.

```
A  
B  
Change C  
Change D  
E  
F
```

Figure 18-34: Edited File

The change tree for this editing session appears in Figure 18-35.

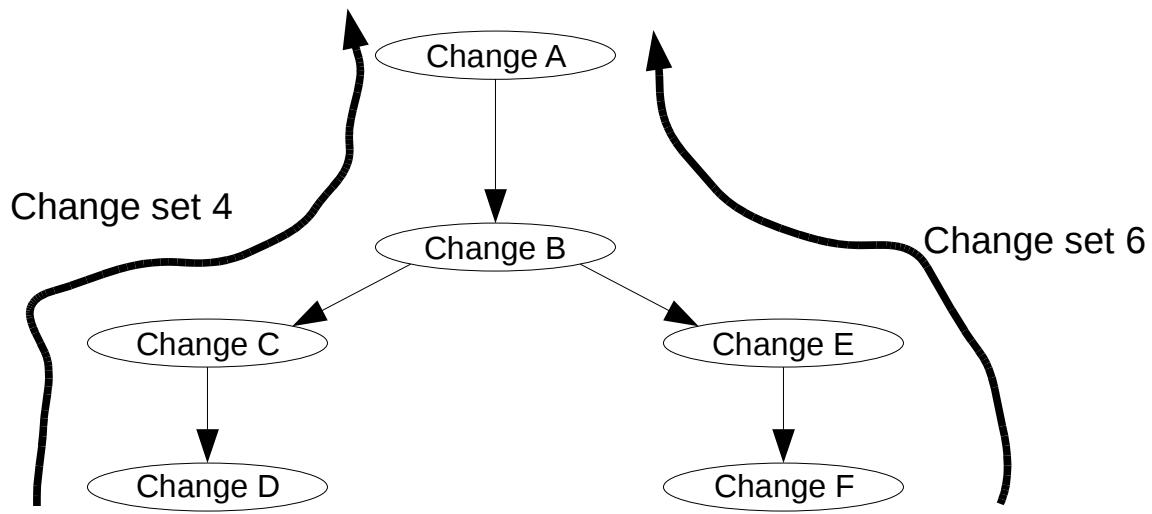


Figure 18-35: Change tree

We can now list the change sets with the `:undolist` (`:undol`) command.

```
:undolist
number changes time
 4      4      12 seconds ago
 6      4      6 seconds ago
Press ENTER or type command to continue
```

This listing shows our two change sets. Right now we are at the bottom of change set number 4. Pressing undo (**u**) four times causes each of the changes (F, E, B, A) to be undone. Pressing redo (**CTRL-R**) four times redoes the changes.

Now let's switch to change set number 4 with the `:undo` (`:u`) command:

:undo 4

All of a sudden the file changes to the state it was in after we completed the last step of change set 4 (Change D). Figure 18-36 shows the result:

A
B
C
D
Change E
Change F

Figure 18-36: Result of :undo 4

The Vim Tutorial and Reference

Now the undo (**u**) command goes up and down the changes in change set number 4. (Changes D, C, B, A.)

Lets switch to the second change set (**:undo 6**). The screen should look like Figure 18-37.

```
A  
B  
Change C  
Change D  
E  
F  
~  
0 changes; before #6 17:16:44
```

Figure 18-37: Screen after **:undo 6**

Now lets undo a change using the **g-** command. The first **g-** undoes change F. See Figure 18-38.

```
A  
B  
Change C  
Change D  
E  
Change F
```

Figure 18-38: Screen after first **g-**

The next **g-** undoes change E, but moves over to a new part of the change tree and redoes changes C and D. See Figure 18-39.

```
A  
B  
C  
D  
Change E  
Change F
```

Figure 18-39: File after second **g-**.

Pressing **g-** again undoes Change D. So we are now going up the undo stack for change #4. The **g+** command goes down the change stack and redoes each change in a similar manner.

Figure 18-40 shows changes traversed by the **g-** command.

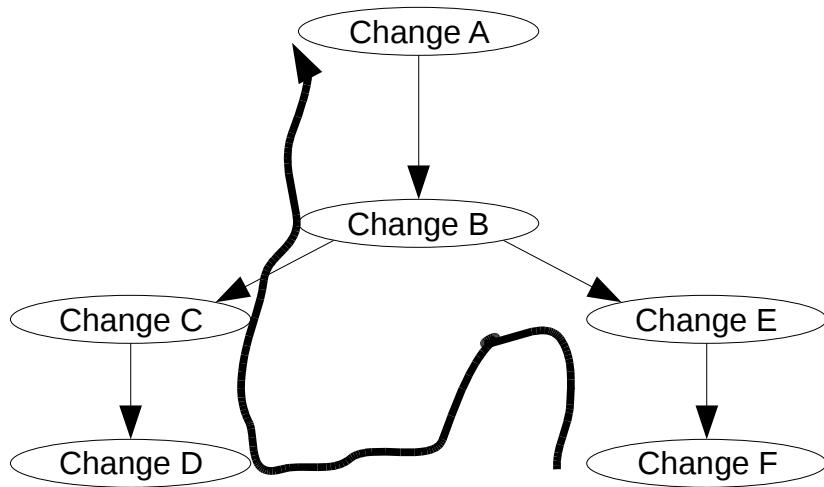


Figure 18-40 **g-** change path.

The **:undojoin** (**:undoj**) command tells Vim to make all further changes part of the same undo block. This undo block ends when the user executes the next command. This command is useful for scripts and functions that want all their commands to be part of the same undo block.

Getting Out

The **zq** command is an alias for the **:q!** or **:quit!** Command. The command exits and discards all changes.

The **:write** command writes out the file. The **:quit** command exits. You can use a shorthand for these commands:

```
:wq
```

This command can take a filename as an argument. In this case, the edit buffer will be written to the file and then *Vim* will exit. To save your work under a new filename and exit, for instance, use the following command:

```
:wq count.c.new
```

This command will fail with an error message if the file *count.c.new* exists and is read-only. If you want *Vim* to overwrite the file, use the override option (!):

```
:wq! count.c.new
```

The Vim Tutorial and Reference

Finally, you can give the **:wq** command a line-range argument. (See *Chapter 25: Complete Command-Mode (:) Commands* for more on ranges.) If a line range is present, only those lines are written to the file. To write out only the first 10 lines of a file and exit, for example, execute the following command:

```
:1,10wq count.c.new
```

The **:xit** (**:x**) command acts much like the **:wq** command except that it only writes the file if the buffer has been modified.

Chapter 19: Advanced Searching Using Regular Expressions

Vim has a powerful search engine that enables you to perform many different types of searches. In this chapter, you learn about the following:

- Turning on and off case sensitivity
- Search options
- Instant word searching
- How to specify a search offset
- A full description of regular expressions

Searching Options

This section describes some of the more sophisticated options that you can use to fine-tune your search. The '**hlsearch**' ('**hls**') option has been turned on to show you how these options affect the searching.

Case Sensitivity

By default, *Vim*'s searches are case sensitive. Therefore, `include`, `INCLUDE`, and `Include` are three different words and a search will match only one of them. The following example searched for `include`. Notice that `INCLUDE`, `Include` and `iNCLude` are not highlighted. Figure 19-1 shows the result of an `/include` command.

The Vim Tutorial and Reference

```
# */

#ifndef HAVE_CONFIG_H
# include "auto/config.h"
#endif
#include <stdio.h>
#include <stdlib.h>
#include <FCNTL.H>
#include <SYS/TYPES.H>
#include <NETDB.H>
#include <netinet/in.h>
#include <errno.h>
#include <sys/socket.h>
#ifndef HAVE_LIBGEN_H
# include <libgen.h>
#endif
/include
```

Figure 19-1: Case-sensitive search.

Now let's turn on the '**ignorecase**' option by entering the following command:

```
:set ignorecase
```

Now when you search for include, you will get all four flavors of the word as (see Figure 19-2).

The Vim Tutorial and Reference

```
# /*

#ifndef HAVE_CONFIG_H
#include "auto/config.h"
#endif
#include <stdio.h>
#include <stdlib.h>
#include <FCNTL.H>
#include <SYS/TYPES.H>
#include <NETDB.H>
#include <netinet/in.h>
#include <errno.h>
#include <sys/socket.h>
#endif HAVE_LIBGEN_H
#include <libgen.h>
#endif
#include
```

Figure 19-2: Non-case-sensitive search.

To turn on case sensitivity, use this command:

```
:set noignorecase
```

(Technically what you are doing is turning off case insensitivity, but it tortures the English language too much to say it this way.)

If you have '**ignorecase**' set, word matches word, WORD, and Word. It also means that WORD will match the same thing. If you set the following two options, any search string typed in lowercase is searched, ignoring the case of the search string:

```
:set ignorecase
:set smartcase
```

('scs' is the abbreviation for '**smartcase**').)

If you have a string with at least one uppercase character, however, the search becomes case sensitive. Thus you have the following matches:

String	Matches
word	word,Word,WORD, word
Word	Word
WORD	WORD

Word

Word

Wrapping

By default, a forward search starts searching for the given string starting at the current cursor location. It then proceeds to the end of the file. If it does not find the string by that time, it starts from the beginning and searches from the start of the file to the cursor location. Figure 19-3 shows how this works.

```
1 // Read at most 10MB
2 const unsigned int MAX_READ=(10*1024*1024)
3 // Size of a buffer
4 const unsigned int BUF_SIZE = (62 * 1024);
5
6
7 // Buffer to be written
8 static unsigned char buffer[BUF_SIZE];
9
10
```

Figure 19-3: Wrapping.

This example starts by searching for `unsigned`. The first search goes to line 6. The next search moves to line 9. When trying to search again, you reach the end of the file without finding the word. At this point, the search wraps back to line 1 and the search continues. The result is that you are now on line 3.

Turning Off Search Wrapping

To turn off search wrapping, use the following command:

```
:set nowrapscan
```

(`'ws'` is the abbreviation for `'wrapscan'`)

Now when the search hits the end of the file, an error message displays (see Figure 19-4).

```
1
2 // Read at most 10MB
3 const. unsigned int MAX_READ = (10 * 1024 *1024
4
5 // Size of a buffer
6 const unsigned int BUF_SIZE = (62 * 1024);
7
8 // Buffer to be written
9 static unsigned char buffer[BUF_SIZE];
10

E385: search hit BOTTOM without match for: unsigned
```

*Figure 19-4: '**nowrapscan**'.*

To go back to normal wrapping searches, use the following command:

```
:set wrapscan
```

Interrupting Searches

If you are in the middle of a long search and want to stop it, you can type **CTRL-C** on a UNIX system or **CTRL-BREAK** on Microsoft Windows. On most systems, unless you are editing a very large file, searches are almost instantaneous.

Instant Word Searches

The ***** command searches for the word under the cursor. For example, position the cursor on the first **const**. Pressing ***** moves the cursor to the next occurrence of the word, specifically line 26. Figure 19-5 shows the results.

The Vim Tutorial and Reference

```
19 #include <sys/fcntl.h>
20 #include <sys/time.h>
21 #include <errno.h>
22
23 // Read at most 10MB
24 const unsigned int MAX_READ =(10*1024*1024);
25 // Size of a buffer
26 const unsigned int BUF_SIZE = (62 *1024);
27
28 // Buffer to be written
29 static unsigned char buffer[BUF_SIZE];
```

*

Figure 19-5: * command.

The **#** or **f** command does an instant word search in the backward direction.

These commands work on whole words only. In other words, if you are on **const** and conduct a ***** search, you will not match **constant**.

The **g*** command performs an instant word search, but does not restrict the results to whole words. So whereas ***** will not match **constant**, the **g*** command will match it. The **g#** (**gf**) command does the same thing in the reverse direction.

Search Offsets

By default, the search command leaves the cursor positioned on the beginning of the pattern. You can tell *Vim* to leave it some other place by specifying an offset. For the forward search command (/), the offset is specified by appending a slash (/) and the offset, as follows:

```
/const/2
```

This command searches for the pattern **const** and then moves to the beginning of the second line past the pattern. Figure 19-6 shows how this works.

The Vim Tutorial and Reference

/const/4

Find const

Go down 4
lines

```
19 #include <sys/fcntl.h>
20 #include <sys/time.h>
21 #include <errno.h>
22
23 // Read at most 10MB
24 const unsigned int MAX_READ =(10*1024*1024);
25 // Size of a buffer
26 const unsigned int BUF_SIZE = (62 *1024);
27
28 // Buffer to be written
29 static unsigned char buffer[BUF_SIZE];
```

Figure 19-6: Search offsets.

If the offset is a simple number, the cursor will be placed at the beginning of the offset line from the match. The offset number can be positive or negative. If it is positive, the cursor moves down that many lines; if negative, it moves up.

If the offset begins with **b** and a number, the cursor moves to the beginning of the pattern, and then travels the "number" of characters. If the number is positive, the cursor moves forward, if negative, backward. The command **/const/b2** moves the cursor to the beginning of the match, for instance, and then two characters to the right (see Figure 19-7).

```
const unsigned int BUF_SIZE = (62 * 1024);  
/const /b2 (Two characters past beginning)
```

Figure 19-7: /const/b2

Note: The **b** offset is a synonym for **s**. Therefore, you can use **b** (begin), and **s** (start) for the first character of the match.

The **e** offset indicates an offset from the end of the match. Without a number it moves the cursor onto the last character of the match. The command **/const/e** puts the cursor on the **t** of **const**. Again, a positive number moves the cursor to the right, a negative number moves it to the left (see Figure 19-8).

```
const unsigned int BUF_SIZE = (62 * 1024);  
/const /e3 (Two characters after end)
```

Figure 19-8: /const/e-3

The Vim Tutorial and Reference

Finally, there is the null offset. This is the empty string. This cancels the preceding offset.

Specifying Offsets

To specify an offset on a forward search (/ command), append **/offset** to the command, as follows:

```
/const/e+2
```

If you want to repeat the preceding search with a different offset, just leave out the pattern and specify the new offset:

```
//5
```

To cancel an offset, just specify an empty offset.

```
//
```

For example:

/const/e+2	/	//	Search moves to the end of the pattern, and then to the right two characters.
/			Repeats last search, with the preceding offset.
//			Repeats the last search with no offset. (Cursor will be placed on the first character of the pattern.)

To specify an offset for a reverse search (?) command, append **?offset** to the command, as follows:

```
?const?b5
```

To repeat with the same pattern and a new offset, use the following:

```
??-2
```

To remove the offset and repeat the search with the preceding pattern, use the following:

```
??
```

One thing to remember when using search offsets, the search always starts from the current cursor position. This can get you into trouble if you use a command such as this:

```
/const/-2
```

This command searches for const and then moves up two lines. If you then repeat the search with the **n** command, it goes down two lines, finds the const you just found, and then moves the cursor back up two lines for the offset. The result is that no matter how many times you type **n**, you go nowhere.

Complete Regular Expressions

The search logic of *Vim* uses regular expressions. You saw some simple ones in *Chapter 3: Searching* but this chapter goes into them in extreme detail. Regular expressions enable you to search for more than simple strings. By specifying a regular expression in your search command, you can search for a character pattern, such as “all words that begin with t and end in ing” (regular expression = \<t[^]*ing\>).

However, the power of regular expressions comes with a price. Regular expressions are quite cryptic and terse. It may take some time for you to get used to all the ins and outs of this powerful tool. While learning regular expressions, you should execute the following command:

```
:set hlsearch
```

This causes *Vim* to highlight the text you matched with your last search. Therefore, when you search for a regular expression, you can tell what you really matched (as opposed to what you thought you matched).

A regular expression consists of a series of atoms. An atom is the smallest matching unit in a regular expression. Atoms can be things like a single character, such as **a** (which matches the letter a), or a special character, such as **\$** (which matches the end of the line). Other atoms, such as **\<** (word start, see the following section), consist of multiple characters.

Beginning (\<) and End (\>) of a Word

The atom **\<** matches the beginning of a word. The atom **\>** matches the end of a word. For example, a search for the expression **for** finds all occurrences of **for**, even those in other words, such as **Californian** and **Unfortunately**. Figure 19-9 shows the results of this search. If you use the regular expression **\<for\>**, however, you match only the actual word **for**. Figure 19-10 contains the results of this refined search.

The Vim Tutorial and Reference

```
Calls and letters to the company failed to correct  
this problem. Finally the fellow just gave up and  
wrote a check for $0.00 and the bills ceased.  
  
A Californian who loved sailing went down and applied  
for a personalized license plate. He filled in his  
three choices as 1)SAIL 2)SAILING and 3)NONE. He got  
a new plate labeled "NONE."  
Unfortunately, when the police write out a ticket  
/for
```

Figure 19-9: Search for **for**.

```
Calls and letters to the company failed to correct  
this problem. Finally the fellow just gave up and  
wrote a check for $0.00 and the bills ceased.  
  
A Californian who loved sailing went down and applied  
for a personalized license plate. He filled in his  
three choices as 1)SAIL 2)SAILING and 3)NONE. He got  
a new plate labeled "NONE."  
Unfortunately, when the police write out a ticket  
/\<for\>
```

Figure 19-10: Search for \<for\>.

Modifiers and Grouping

The modifier * is used to indicate that an atom is to be matched 0 or more times. The match is "greedy." In other words, the editor will try to match as much as possible. Thus, the regular expression **te*** matches te, tee, teee, and so on.

The expression **te*** also matches the string t. Why? Because **e*** can match a zero length string of e's. And t is the letter t followed by zero e's.

Figure 19-11 shows the results of the search for **te***.

```
This is a test.  
te tee teee teee
```

Figure 19-11: Search for **te***.

The \+ modifier indicates that the atom is to be matched one or more times. Therefore, **te\+** matches te, tee, and teee, but not t. (**te\+** is the same as **tee***.). Figure 19-12 illustrates what is matched for the search **/te\+**.

The Vim Tutorial and Reference

```
This is a test.  
te tee teee teee
```

Figure 19-12: Search for **te\+.**

Finally, there is the \= modifier. It causes the preceding atom to be matched zero or one time. This means that **te\=** matches t and **te**, but not **tee**. (Although it will match the first two characters of **tee**.) Figure 19-13 shows a typical search.

```
This is a test.  
te tee teee teee
```

Figure 19-13: Search for **te\=.**

Special Atoms

A number of special escaped characters match a range of characters. For example, the \a atom matches any letter, and the \d option matches any digit. The regular expression \a\al\al matches any three letters.

Now try a search for any four digits. Figure 19-14 displays the results.

```
19 #include <sys/fcntl.h>  
20 #include <sys/time.h>  
21 #include <errno.h>  
22  
23 // Read at most 10MB  
24 const unsigned int MAX_READ=(10*1024*1024);  
25 // Size of a buffer  
26 const unsigned int BUF_SIZE = (62 * 1024);  
27  
28 // Buffer to be written  
29 static unsigned char buffer[BUF_SIZE];
```

Figure 19-14: Search for **\d\d\d\d.**

Now try a search for any three letters followed by an underscore. Figure 19-15 displays the results.

```
19 #include <sys/fcntl.h>
20 #include <sys/time.h>
21 #include <errno.h>
22
23 // Read at most 10MB
24 const unsigned int MAX_READ=(10*1024*1024);
25 // Size of a buffer
26 const unsigned int BUF_SIZE = (62 * 1024);
27
28 // Buffer to be written
29 static unsigned char buffer[BUF_SIZE];
```

Figure 19-15: Search for \a\aa_.

Character Ranges

The \a atom matches all the letters (uppercase and lowercase). But suppose you want to match only the vowels. The range operator enables you to match one of a series of characters. For example, the range [aeiou] matches a single lowercase vowel. The string t[aeiou]n matches tan, ten, tin, ton and tun.

You can specify a range of characters inside the brackets [] by using a dash. For example, the pattern [0-9], matches the characters 0 through 9. (That is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.)

You can combine ranges with other characters. For example, [0-9aeiou] matches any digit or lowercase vowel.

The ^ character indicates a set of characters that match everything except the indicated characters. To match anything except a vowel, for example, you can specify [^aeioAEIOU].

Note: To match special characters such as *, -, ^, [, and other special characters you need to escape them. (e.g. /*\^-^\[^ matches “*-^[".)

However, in certain circumstances, *Vim* can figure out that you don't want a special character to be special. For example, ^ at the beginning of an expression matches the beginning of line. A ^ in the middle of the expression matches ^. So 2\^4 and 2^4 match the same thing. But rather try remember them all, all you need to remember is that if you escape always you will rarely go wrong.

Character Classes

Suppose you want to specify all the uppercase letters. One way to do this is to use the expression `[A-Z]`. Another way is to use one of the predefined character classes. The class `[:upper:]` matches the uppercase characters. Therefore, you can write `[A-Z]` as `[[:upper:]]`.

You can write the entire alphabet, upper- and lowercase, `[[:upper:]]` `[[:lower:]]`. There are a large number of different character classes.

Note: You cannot use the special atoms like `\a` and `\d` in a range. For example, `[\a\d]` matches the characters `\`, `a`, `\`, and `d`. It does not match the letters (`\a`) and digits (`\d`).

Repeat Modifiers

You can specify how many times an atom is to be repeated. The general form of a repeat is as follows:

```
\{minimum, maximum}
```

For example, the regular expression `a\{3,5}` will match 3 to 5 a's. (that is, aaa, aaaa, or aaaaa.) By default, the *Vim* editor tries to match as much as possible. So `a\{3,5}` will match as many a's as it can (up to 5).

The minimum can be omitted, in which case it defaults to zero. Therefore, `a\{,5}` matches 0-5 repeats of the letter. The maximum can be omitted as well, in which case it defaults to infinity. So `a\{3,}` matches at least 3 a's, but will match as many a's as you have got on the line.

If only one number is specified, the atom must match exactly that number of times. Therefore, `a\{5}` matches 5 a's, exactly.

Repeating as Little as Possible

If you put a minus sign (-) before any of the numbers, the *Vim* editor tries to match as little as possible.

Therefore, `a\{-3,5}` will match 3 to 5 a's, as little as possible. Actually if this expression is by itself, it will always match just three a's. That is because even if you have the word aaaaa, the editor will match as little as possible.

The specification `a\{-3,}` matches 3 or more a's, as little as possible. The expression `a\{-,5}` matches 0-5 letters.

The Vim Tutorial and Reference

The expression `a\{-}` matches 0 to infinity number of characters, as little as possible. Note that this pattern by itself will always match zero characters. It only makes sense when there is something after it. For example: `[a-z]\{-}x` will match `cx` in `cxcx`. Using `[a-z]*x` would have matched the whole `cxcx`.

Finally, the specification `a\{-5}` matches exactly 5 a's, as little as possible. Because as little as possible is exactly 5, the expression `a\{-5}` acts just like `a\{5}`.

Grouping (\(\))

You can specify a group by enclosing it in a `\(` and `\)`. For example, the expression `a*b` matches `b`, `ab`, `aab`, `aaab`, and so on. The expression `a\(\text{xy}\)*b` matches `ab`, `aXYb`, `aXYXYb`, `aXYXYXYb`, and so on.

When you define a group using `\(\)`, the first enclosed string is assigned to the atom `\1`. To match the string `the`, for instance, use the regular expression `\(\text{the}\) \1`. To find repeated words, you can get a bit more general and use the expression `\(\(\<\text{a}\+\>\)\) \1`. Figure 19-16 breaks this into its components.

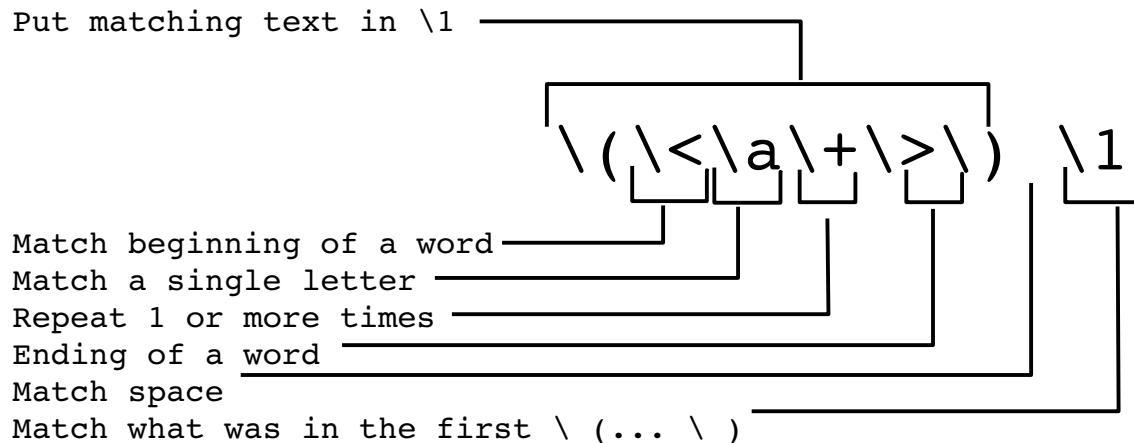


Figure 19-16: The repeat (`\1`) expression.

The first group is assigned to `\1`, the second to `\2`, and so on.

The Or Operator (\|)

The `\|` operator enables you to specify two or more possible matches. The regular expression `foo\|bar` matches `foo` or `bar`. For example, the search

```
/procedure\|function
```

searches for either procedure or function.

Putting It All Together

Let's create a regular expression to match California license plate numbers. A sample license plate looks like 1MGU103. The pattern is one digit, three uppercase letters, and three digits. There are several ways of doing this.

Start by specifying the digit as [0-9], now add the uppercase letter: [0-9][A-Z]. There are three of them, so you get [0-9][A-Z]\{3\}. Finally, you add the three digits on the end, resulting in [0-9][A-Z]\{3\}[0-9]\{3\}.

Another way to do this is to recognize that \d represents any digit and \u any uppercase character. The result is \d\u\{3\}\d\{3\}.

The experts tell us that this form is faster than using the [] form. If you are editing a file where this speed up makes a difference, however, your file might be too big.

You can accomplish this without repeats as well: \d\u\u\u\d\d\d.

Finally, you can use the character classes, yielding [[:digit:]]
[[:upper:]]\{3\}[[:digit:]]\{3\}.

All four of these expressions work. Which version should you use? Whichever one you can remember. You should remember this old adage: The simple way you can remember is much faster than the fancy way you can't.

The 'magic' Option

The expressions discussed so far assume that the '**magic**' option is on. When this option is turned off, many of the symbols used in regular expressions lose their magic powers. They only get them back when escaped. Specifically, if you execute the command

```
:set nomagic
```

the *, ., [, and] characters are not treated as special characters. If you want to use the * for "0 or more repeats," you need to escape it: *. You should keep the '**magic**' option on (the default) for portability and macro files.

Using \m in a regular expression tells *Vim* to treat all the following text as if '**magic**' were set. The special sequence |M tells *Vim* to treat the string as if '**nomagic**' were set.

The Vim Tutorial and Reference

The string `\v` turns “very magic” mode which makes almost every character magic. To turn this off use `\V`. The following table summarizes these items.

	'magic'	'nomagic'	
<code>\v</code>	<code>\m</code>	<code>\M</code>	<code>\v</code>
<code>\$</code>	<code>\$</code>	<code>\$</code>	<code>\\$</code>
<code>.</code>	<code>.</code>	<code>\.</code>	<code>\.</code>
<code>*</code>	<code>*</code>	<code>*</code>	<code>*</code>
<code>()</code>	<code>\(\)</code>	<code>\(\)</code>	<code>\(\)</code>
<code> </code>	<code>\ </code>	<code>\ </code>	<code>\ </code>
<code>\a</code>	<code>\a</code>	<code>\a</code>	<code>\a</code>
<code>\\"</code>	<code>\\"</code>	<code>\\"</code>	<code>\\"</code>
<code>\.</code>	<code>\.</code>	<code>.</code>	<code>.</code>
<code>\{</code>	<code>{</code>	<code>{</code>	<code>{</code>

Offset Specification Reference

<code>[num]</code>	
<code>+[num]</code>	Down <code>[num]</code> lines. Cursor is placed at the beginning of the line.
<code>-[num]</code>	Up <code>[num]</code> lines. Cursor is placed at the beginning of the line.
<code>e</code>	End of the match.
<code>e[num]</code>	End of the match, the move <code>[num]</code> . If <code>[num]</code> is positive, move right, negative, move left.
<code>b</code>	
<code>s</code>	Start of the match.
<code>b[num]</code>	
<code>s[num]</code>	Start of the match, then move <code>[num]</code> . If <code>[num]</code> is positive, move right; negative, move left.

Regular Expressions Reference

The following table assumes that the '`magic`' option is on (the default).

Simple Atoms

<code>x</code>	The literal character <code>x</code> .
<code>^</code>	Start of line.
<code>\$</code>	End of line.
<code>.</code>	A single character.
<code>\<</code>	Start of a word.
<code>\></code>	End of word.

Range Atoms

[abc]	Match either a, b, or c.
[^abc]	Match anything except a, b, or c.
[a-z]	Match all characters from a through z.
[a-zA-Z]	Match all characters from a through z and A through Z.

Character Classes

[:alnum:]	Match all letters and digits.
[:alpha:]	Match letters.
[:ascii:]	Match all ASCII characters.
[:backspace:]	Match the backspace character (<BS>).
[:blank:]	Match the space and tab characters.
[:cntrl:]	Match all control characters.
[:digit:]	Match digits.
[:escape :]	Matches the escape character (<Esc>).
[:graph:]	Match the printable characters, excluding space.
[:lower:]	Match lowercase letters.
[:print:]	Match printable characters, including space.
[:punct:]	Match the punctuation characters.
[:return:]	Matches the end-of-line (carriage return, <Enter>, <CR>, <NL>).
[:space:]	Match all whitespace characters.
[:tab:]	Match the tab character (<Tab>).
[:upper:]	Match the uppercase letters.
[:xdigit:]	Match hexadecimal digits.

Patterns (Used for Substitutions)

\(pattern\)	Mark the pattern for later use. The first set of \() marks a subexpression as \1, the second \2, and so on.
\1	Match the same string that was matched by the first subexpression in \(and \). For example: \([a-z]\) .\1 matches ata, ehe, tot, and so forth.
\2	Like \1, but uses second subexpression.
\9	Like \1, but uses ninth subexpression.

Special Character Atoms

\a	Alphabetic character (A-Za-z).
\A	Non-alphabetic character (any character except A-Za-z).

The Vim Tutorial and Reference

\b	<BS>.
\d	Digit
\D	Non-digit.
\e	<Esc>.
\f	Any filename character as defined by the ' isfname ' option.
\F	Any filename character, but does not include the digits.
\h	Head of word character (A-Za-z_).
\H	Non-head of word character (any character except A-Za-z_).
\i	Any identifier character as defined by the ' isident ' option.
\I	Any identifier character, but does not include the digits.
\k	Any keyword character as defined by the ' iskeyword ' option.
\K	Any keyword character, but does not include the digits.
\l	Lowercase character (a-z).
\L	Non-lowercase character (any character except a-z).
\o	Octal digit (0-7).
\O	Non-octal digit.
\p	Any printable character as defined by the ' isprint ' option.
\P	Any printable character, but does not include the digits.
\r	<CR>.
\s	Whitespace (<Space> and <Tab>).
\S	Non-whitespace character. (Any character except <Space> and <Tab>).
\t	<Tab>.
\u	Uppercase character (A-Z).
\U	Non-uppercase character (any character except A-Z).
\w	Word character (0-9A-Za-z_).
\W	Non-word character (any character except 0-9A-Za-z_).
\x	Hexadecimal digit (0-9 a-f A-F).
\X	Non-hexadecimal digit.
\~	Matches the last given substitute string.

Modifiers

The Vim Tutorial and Reference

*	Match the previous atom 0 or more times. As much as possible.
\+	Match the previous atom 1 or more times. As much as possible.
\=	Match the previous atom 0 or 1 times
\{}	Match the previous atom 0 or more times. (Same as the * modifier.)
\{ <i>n</i> }	
\{- <i>n</i> }	Match the previous atom <i>n</i> times.
\{ <i>n</i> , <i>m</i> }	Match the previous atom <i>n</i> to <i>m</i> times.
\{ <i>n</i> ,}	Match the previous atom <i>n</i> or more times.
\{, <i>m</i> }	Match the previous atom from 0 to <i>m</i> times.
\{- <i>n</i> , <i>m</i> }	Match the previous atom <i>n</i> to <i>m</i> times. Match as little as possible.
\{- <i>n</i> ,}	Match the previous atom at least <i>n</i> times. Match as little as possible.
\{-, <i>m</i> }	Match the previous atom up to <i>m</i> times. Match as little as possible.
\{-}	Match the previous atom 0 or more times. Match as little as possible.
str1\ str2	Match <i>str1</i> or <i>str2</i> .

Chapter 20: Advanced Text Blocks and Multiple Files

The Vim editor has lots of different ways of doing things. *Chapter 4: Text Blocks and Multiple Files* presented a representative subset of the commands dealing with text blocks and multiple files and described them. It is entirely possible for you to edit efficiently using only those commands.

This chapter shows you all the other ways of doing things. If you find that you do not like the limitations of the commands in *Chapter 4*, read this one; you should find a way to get around your annoyances.

For example, you learned how to yank and put (cut and paste) using a single register to hold the text. That is fine if you are dealing with a single block of text. If you want to deal with more, however, check out how to use multiple registers later in this chapter.

This chapter covers the following:

- Different ways to yank and put
- How to use special registers
- How to edit all the files containing a specific string
- Advanced commands for multiple files
- Global marks
- Advanced insert-mode commands
- How to save and restore your setting by using a VIMINFO file
- Dealing with files that contain lines longer than the screen width

Additional Put Commands

When inserting lines, **p** and **P** commands move the cursor the first non-blank character on the line. The **gp** command works just like the **p** command, except that the cursor is left just after at the end of the new text. The **gP** command does the same things for the **P** command. Figure 20-1 shows the effects of these commands.

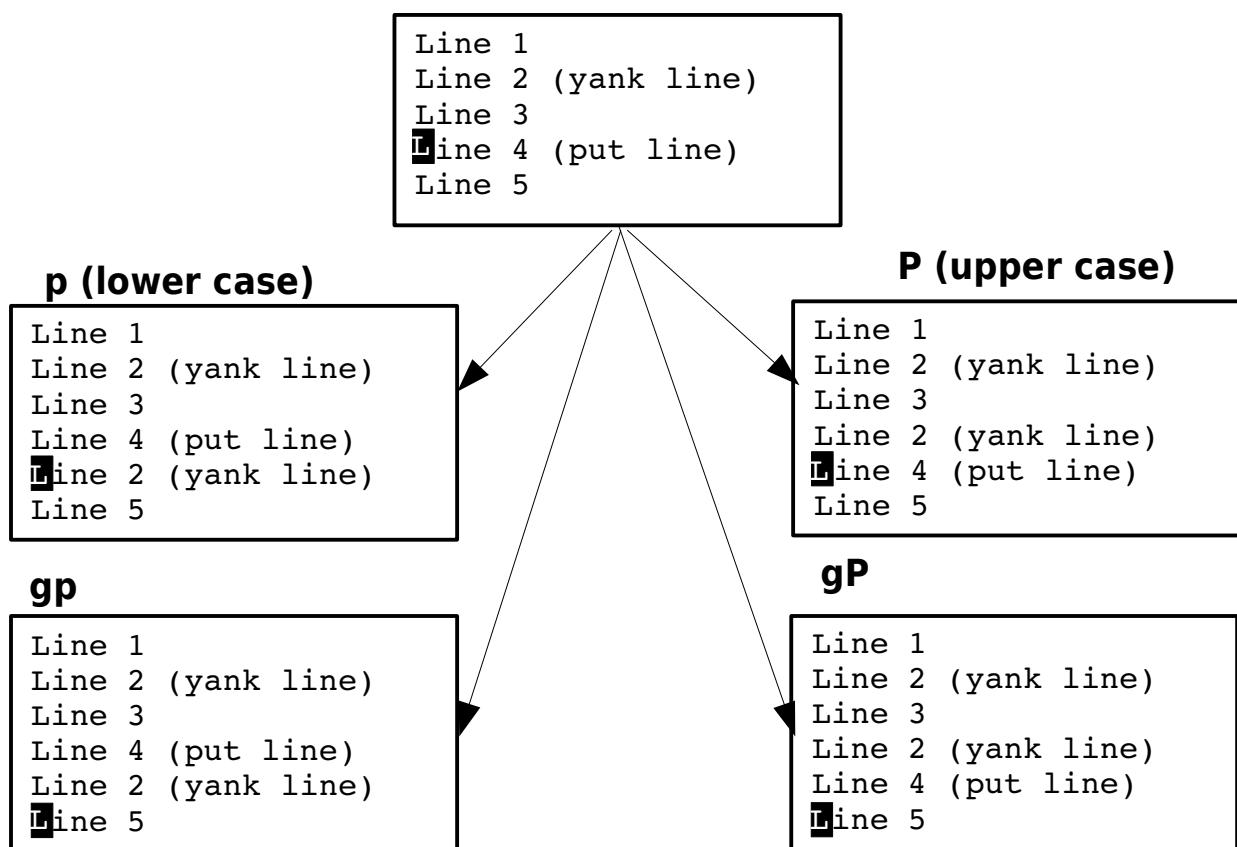


Figure 20-1: Paste (put) commands.

Special Marks

Vim has a number of special built-in marks. The first one is the single quotation (') mark. It marks the location of the cursor before the latest jump. In other words, it is your previous location (excluding minor moves such as up/down and so on). Other special marks include the following:

-] The beginning of the last inserted text
- [The end of the last inserted text
- " The last place the cursor was resting when you left the file

Manipulating Marks

The **:delmarks** (**:delm**) command deletes all the marks. If you wish to delete a specific set of marks, specify them as an argument to **:delmarks**.

The **:lockmarks** (**:loc**) command locks the location of the marks while a command is being executed. For example:

```
:lockmarks :%! sort
```

The **:lockmarks** command assumes that the command does not change the numbers of lines in the file.

The **:keepmarks** (**:ke**) does the same thing except that if the file gets shorter any marks that are located past the end of the file are deleted. This only works for filter (**:!**) commands.

Multiple Registers

So far, you have performed all your yanks and deletes without specifying which register to use. If no register is specified, the unnamed register is used. The characters that denote this register are two double quotation marks (""). The first double quote denotes a register; the second double quote is the name of the register. (Therefore, for example, "**a**" means use register **a**.)

You can specify which register the deleted or yanked text is to go into by using a register specification before the command. The format of a register specification is "**register**", where register is one of the lowercase letters. (This gives you 26 registers to play around with.)

Therefore, whereas **yy** puts the current line into the unnamed register, the command "**a****yy**" places the line in the **a** register, as seen in Figure 20-2. (The text also goes into the unnamed register at the same time.)

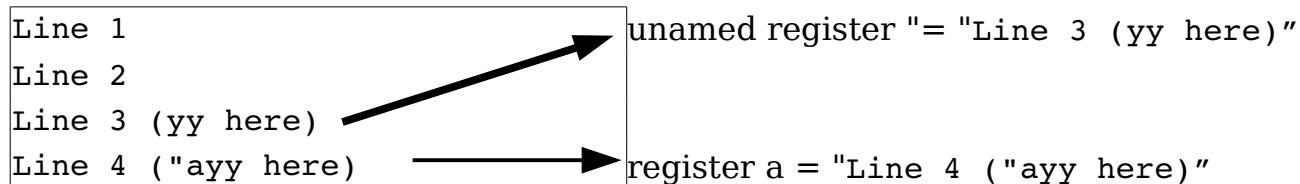


Figure 20-2: Using the **a** register for yank and put.

```
Unnnamed Register
```

The Vim Tutorial and Reference

It seems a bit silly to name the unnamed register, "The Unnamed Register," because this gives it a name. The name is a misnomer, because the unnamed register is named "The Unnamed Register." So, in fact, the unnamed register has a name even though it calls itself "The Unnamed Register."

Persons understanding the preceding paragraph have demonstrated aptitude for writing programs and are urged to enroll in their nearest engineering college.

To get an idea of what the registers contain, execute the following command:

```
:registers
```

(**:reg** is the abbreviation for **:registers**.)

Figure 20-3 shows the results of this command.

```
:registers
--- Registers ---
"" Line 3 (yy here)^J
"0 Line 3 (yy here)^J
"1 We will tdelete he word in the middle
"2 /* File for bad names */^J
"3 Line 2^J
"4 To err is human -- ^J to really scre
"5 ^J
"6 ^J
"7 to really screw up, you need a com
"a Line 4 ("ayy here)^J
" to be yy'ed)
". "ayy here)
": registers
"% test.txt
"# tmp.txt
Press RETURN or enter command to continue
```

Figure 20-3: :registers command.

This illustration shows that the unnamed register ("") contains Line 3 (**yy** here).

The **a** register contains Line 4 ("ayy here).

The alphabetic registers are the normal ones used for yanking and pasting text. Other, special registers are described in the following sections.

The Vim Tutorial and Reference

You can display the contents of specific registers by giving them as an argument to the `:registers` command. For example, the following command displays the contents of registers **a** and **x**:

```
:registers ax
```

Appending Text

When you use a command such as "**ayy**", you replace the text in the register with the current line. When you use the uppercase version of a register, say "**Ayy**", you append the text to what is already in the register (see Figure 20-4). Note that the result is two lines, "Line 3" and "Line 2". (The `^J` in the register indicates end of line.)

```
Line 1
Line 2 ("Ayy here)
Line 3 ("ayy here)
Line 4
~
~
1) "ayy
Yank line into
register a
2) "Ayy
Append the line on to
register a
```

Figure 20-4: Appending text to a register.

Special Registers

Vim has a number of special registers. The first is the unnamed register, whose name is double quote (").

Others include the registers **1** through **9**. Register **1** contains the last text you deleted; register **2** the next to last, and so on.

(Back in the bad old days of Vi, these registers were a lifesaver. You see, Vi had only one level of undo. So if you deleted three lines by executing `dd` three times, you were out of luck if you wanted to undo the delete using the `u` command. Fortunately, the three lines were stored in registers **1**, **2**, and **3**, so you could put them back with "**1P"2P"3P**". You can also use the command "**1P..**" ("**1P** and two dots)).

Other special registers include the following:

Register	Description	Writeable
0	The last yanked text	Yes
-	The last small delete	No
.	The last inserted text	No
%	The name of the current file	No
#	The name of the alternate file	No
/	The last search string	No
:	The last ":" command	No
_	The black hole (more on this later)	Yes
=	An expression (see next page)	No
*	The text selected with the mouse	Yes

The Black Hole Register (_)

Placing text into the black hole register causes it to disappear. You can also "put" the black hole register, but this is pretty much useless because the black hole register always contains nothing. The black hole register is useful when you want to delete text without having it go into the **1** through **9** registers. For example, **dd** deletes a line and stores it in **1**. The command "**_dd**" deletes a line and leaves **1** alone.

The Expression Register (=)

The expression register (=) is designed so that you can enter expressions into text. When you enter a command beginning with an expression register specification, the *Vim* editor displays the prompt = at the end of the screen. This gives you an opportunity to type in an expression such as $38*56$, and you can then put the result into the text with the **p** command. For example " $=38*56<\text{Enter}>p$ " gives you 2128. Figure 20-5 shows this register in action.

The width is 38.
The height is 56.
So the area is
~
~
~=38*56

Enter the text, press **<Esc>** to enter normal mode.

Execute "**=**" to start expression mode.
The cursor jumps to here.

The width is 38.
The height is 56.
So the area is 2128

The **p** command pastes the answer in after the cursor.

Figure 20-5: The expression register.

An expression can contain all the usual arithmetic operators (*, +, -, /, and so on) as well as a ton of specialized *Vim* functions and operators. If you are doing more than simple arithmetic, you will want to check the full expression documentation.

You can specify the value of an environment variable, for example, by using the expression **\$NAME** (for instance, **\$HOME**). You can determine the value of a *Vim* variable by just specifying the variable (**LineSize**, for instance).

The Clipboard Register (*)

The clipboard register (*) enables you to read and write data to the system clipboard. This can be the X selection (UNIX) or the Microsoft Windows Clipboard. This enables you to cut and paste text between the *Vim* editor and other applications.

How to Edit All the Files That Contain a Given Word

If you are a UNIX user, you can use a combination of *Vim* and *grep* to edit all the files that contain a given word. This proves extremely useful if you are working on a program and want to view or edit all the files that contain a specified variable.

Suppose, for example, that you want to edit all the C program files that contain the word `frame_counter`. To do this, you use the following command:

```
$ vim `grep -l 'frame_counter' *.c`
```

Consider this command in detail. The *grep* command searches through a set of files for a given word. Because the `-l` option is specified, the command will list only the files containing the word and not print the line itself. The word it is searching for is `frame_counter`. Actually, this can be any regular expression. (Note that what *grep* uses for regular expressions is not as complete or complex as what *Vim* uses.)

The entire command is enclosed in backticks (`). This tells the UNIX shell to run this command and pretend that the results were typed on the command line. So what happens is that the *grep* command is run and produces a list of files; these files are put on the *Vim* command line. This results in *Vim* editing the file list that is the output of *grep*.

The Vim Tutorial and Reference

You might be asking, "Why show this here?" This is a feature of the UNIX shell (for example, *bash*), and is not part of *Vim*'s repertoire. The way to accomplish something similar within *Vim*, and which works on Win32 as well, is as follows:

```
:args `grep 1 'frame_counter' *.c`
```

(**:ar** is the short form of this command and you can use **:next `cmd`** and **:n `cmd`** as well.)

This command sets the argument list (for example, the files "on the command line," as it were).

Note: The *Vim* command **:vimgrep** can perform a similar function.

Editing a Specific File

To edit a specific file in this list (file 2, for instance), you need the following command:

```
:argument 2
```

(**:argument** can be abbreviated as **:argu**.)

This command enables you to specify a file by its position in the argument list. Suppose, for instance, that you start *Vim* with this command:

```
$ gvim one.c two.c three.c four.c five.c six.c seven.c
```

The following command causes you to be thrown into the file *four.c*.

```
:argument 4
```

Changing the File List

The file list is initially set to the list of files you specify on the command line. You can change this list by specifying a new list to the **:args** command. For example:

```
:args alpha.c beta.c gamma.c
```

After executing this command, you start editing *alpha.c*; the next file is *beta.c* and so on. (The previous file list is lost.)

Note: The **:next file-list** and **:n file-list** commands will do the same thing.

The +cmd Argument

Suppose that you want to start editing a file at line 97. You can start *Vim* and execute a **97G**, or you can tell *Vim* to start editing with the cursor on line 97. You can do this by using the option **+linenumber** on the command line. For example:

```
$ gvim +97 file.c
```

You can also use the **+cmd** to search for a string by using **+/string** on the command line. To start editing a file with the cursor positioned on the first line containing `#include`, for instance, use this command:

```
$ gvim +/#include file.c
```

Finally, you can put any command-mode command after the plus sign (+). You can specify the **+cmd** argument in a number of commands. For example, the general form of the **:vi** (:visual) command is as follows:

```
:vi [+cmd] {file}
```

These other commands can take a **+cmd**:

```
:next [+cmd]
:n [+cmd]
:wnext [+cmd]
:wn [+cmd]
:previous [+cmd]
:prev [+cmd]
:N [+cmd]
:wprevious [+cmd]
:wp [+cmd]
:wNext [+cmd]
:wN [+cmd]
:rewind [+cmd]
:rew [+cmd]
:last [+cmd]
:la [+cmd]
:first [+cmd]
:fir [+cmd]
```

Defining the file list (arguments)

To set the list of files being edited use the **:args** command:

```
:args {++opt} [+cmd] {file-list}
```

The Vim Tutorial and Reference

The `{++opt}` argument lets you specify some options to be used for each file. These include:

Option	Meaning
<code>++bad={value}</code>	If <code>{value}</code> is a single character, replace all bad characters with that character. The keyword drop causes the bad characters to be dropped. A value of keeps retains the bad characters.
<code>++bin</code>	Turns on the ' binary ' option.
<code>++binary</code>	
<code>++edit</code>	This only works for :read commands. It specifies that the options for the file be the same as the file being currently edited.
<code>++enc={encoding}</code>	Specify the file encoding.
<code>++encoding={encoding}</code>	
<code>+ff={format}</code>	Specify the file format.
<code>++fileformat={format}</code>	
<code>++nobin</code>	Turn off the ' binary ' option.
<code>++nobinary</code>	

If you wish to just add files to the argument list (and keep the existing files, use the `:argadd (:arga)` command. The general form of this command is:

```
:[count] argadd {file}
```

If a `[count]` is given the file is added after the `[count]` argument. If no count is present, it is added to the end.

Suppose you want to edit the file you've just added. The `:argedit (:arge)` command is a combination of `:argadd` and `:edit`.

To delete arguments from the argument list use the `:argdelete (:argd)` command. It take an file name pattern as an argument, which means you can use wildcards. For example, if you wish to remove all the C header files from an argument list use the command:

```
:argdelete *.h
```

You can also delete arguments by position. Simply supply the `:argdelete` command with a range. For example, to delete the second and third arguments, use the command:

```
:2,3argdelete
```

Local and Global argument Lists

So far we've been using one argument list. *Vim* actually supports two different types of argument lists. The global argument list is the default for all windows. However a window can switch over and use a local argument list if it wants to.

The **:arglocal** (**:argl**) command switches to the local argument list. If the window doesn't have one yet, the global one is copied. The **:argglobal** (**:argg**) command switches back to the global list. You can also use the **:arglocal** and **:argglobal** commands to set the file list just like the **:args** command. (The **:args** command works on the current set of arguments, while **:arglocal** and **:argglobal** work on a specific list.)

The **:drop** (**:dr**) also sets the argument list. The form of this command is:

```
:drop {file} [file] [file] ...
```

Not only does it set the argument list, but it starts editing the first file. If a the file is already being displayed in a window it will go to it. Otherwise it switch the current window to the file. But if the file in the window hash changes in it and can not be abandoned then it will split the window and to go the first file.

Basically the command is designed for interfacing with an interactive debugger and "does the right thing" when the debugger needs to switch files.

Global Marks

The marks **a-z** are local to the file. In other words, you can place a mark **a** in file *one.c* and another mark **a** in file *two.c*. These marks are separate and have nothing to do with each other. If you execute a go-to-mark command, such as '**a**', you will jump within that file to the given mark.

The uppercase marks (**A-z**) differ. They are global. They mark not only the location within the file, but also the file itself.

Take a look at an example. You are editing the file *one.c* and place the mark **A** in it. You then go on to edit file *two.c*. When you execute the jump-to-mark-**A** command ('**A**'), the *Vim* editor will switch you from file *two.c* to file *one.c* and position the cursor on the mark.

For example, you are editing a bunch of C files named *alpha.c*, *beta.c*, and *gamma.c*. You execute the following commands:

1. **/#include** Find the first #include (in *alpha.c*).

The Vim Tutorial and Reference

2. **mi** Mark it with the mark **i**.

```
/* alpha.c */
#include <stdio.h>
int alph(void)
{
    printf("In alpha\n");
}
```

mi
(mark location
with id "i")

3. **:next** Go to file *beta.c*.

4. **n** Find the first #include.

5. **mi** Mark it with the mark **i**.

6. **/magic_function** Find the magic function.

7. **mF** Mark it with the mark **F**.

```
/* beta.c */
#include <stdio.h>
int magic_function(void)
{
    printf("In beta\n");
}
"beta.c" 8L, 88C
```

mi

mF

8. **:next** Go to file *gamma.c*.

9. **/#include** Find the first #include.

10. **mi** Mark it with the mark **i**.

```
/* gamma.c */
#include <stdio.h>
int gamma(void)
{
    printf("In gamma\n");
}
"gamma.c" 8L, 81C
```

mi

After executing these commands, you have three local marks, all named **i**. If you execute the command '**i**', you jump to the mark in your buffer. The mark **F** is global because it is uppercase.

The Vim Tutorial and Reference

Currently you are in file *gamma.c*. When you execute the command to go to mark **F** ('**F**'), you switch files to *beta.c*. Now you use the following command to go to *alpha.c*.

```
:rewind
```

Place the **F** mark there using the **mF** command. Because this is a global mark (you can put it in only one place), the mark named **F** in the file *beta.c* disappears.

Advanced Text Entry

When you are entering text in insert mode, you can execute a number of different commands. For example, the **<BS>** command erases the character just before the cursor. **CTRL-U** erases the entire line (or at least the part you just inserted). **CTRL-W** deletes the word before the cursor.

Movement

Even though you are in insert mode, you can still move the cursor. You cannot do this with the traditional *Vim* keys **h**, **j**, **k**, and **l**, because these would just be inserted. But you can use the arrow keys **<Left>**, **<Right>**, **<Up>**, and **<Down>**. If you hold down the Control key, you can move forward and backward words. In other words, execute **<C-Left>** to go backward one word, and **<C-Right>** forward.

The **<Home>** command moves the cursor to the beginning of a line, and **<End>** moves to the end. The key **<C-Home>** moves to the beginning of the file, and **<C-End>** moves to the end.

The **<PageUp>** moves one screen backward, and **<PageDown>** a screen forward.

Inserting Text

If you type **CTRL-A**, the editor inserts the text you typed the last time you were in insert mode. Assume, for example, that you have a file that begins with the following:

```
"file.h"  
/* Main program begins */
```

You edit this file by inserting **#include** at the beginning of the first line:

```
#include "file.h"
```

The Vim Tutorial and Reference

```
/* Main program begins */
```

You go down to the beginning of the next line using the commands **j^**. You now start to insert a new line that contains a new include line. So you type **i**. The result is as follows:

```
#include "file.h"  
#include /* Main program begins */
```

The **#include** was inserted because **CTRL-A** inserts the contents of the previous insert. Now you type "**main.h**"<**Enter**> to finish the line:

```
#include "file.h"  
#include "main.h"  
/* Main program begins */
```

The **CTRL-Q** command does a **CTRL-A** and then exits insert mode.

The **CTRL-V** command is used to quote the next character. In other words, any special meaning the character has, it will be ignored. For example, **CTRL-V<Esc>** inserts an escape. You can also use the command **CTRL-V*digits*** to insert the character number digits. For example, the character number 64 is **@**. So **CTRL-V64** inserts **@**. The **CTRL-V*digits*** uses "decimal" digits by default, but you can also insert the hex digits.

For example,

```
CTRL-V123
```

and

```
CTRL-Vx7b
```

both insert the **{** character.

The **CTRL-Y** command inserts the character above the cursor. This is useful when you are duplicating a previous line.

One of my favorite tricks is to use ASCII art to explain complex things such as regular expressions. For example:

```
[0-9]*[a-z]*  
||||| | | | | +---- Repeat 0 or more times  
||||| | +++++----- Any lower case letter  
||||| +----- Repeat 0 or more times ++++  
+----- Any digit
```

Take a look at how you can use **CTRL-Y** to create this file. You start by entering the first two lines:

The Vim Tutorial and Reference

```
[0-9]*[a-z]*  
|||||+---- Repeat 0 or more times
```

Now you type **CTRL-Y** six times. This copies the | from the previous line down six times:

```
[0-9]*[a-z]*  
|||||+---- Repeat 0 or more times  
|||||
```

Now all you have to do is enter the rest of the comments.

The **CTRL-E** command acts like **CTRL-Y** except it inserts the character below the cursor.

Inserting a Register

The command **CTRL-R register** inserts the text in the register. If it contains characters such as <BS> or other special characters, they are interpreted as if they had been typed from the keyboard. If you do not want this to happen (you really want the <BS> to be inserted in the text), use the command **CTRL-R CTRL-R register**.

First you enter the following line:

```
All men^H^H^Hpeople are created equal
```

Note: To enter the backspace characters (which show up as ^H), you need to type **CTRL-V<BS>** or **CTRL-V CTRL-H**.

Now you dump this into register a with the command "**ayy**".

Next you enter insert mode and use **CTRL-Ra** to put the text into the file. The result is as follows:

```
All men^H^H^Hpeople are created equal (original line)  
All people are created equal (CTRL-Ra line)
```

Notice that *Vim* put the contents in as if you had typed them. In other words, the <BS> character (^H) deletes the previous character.

Now if you want to put the contents of the register in the file without interpretation, you could use **CTRL-R CTRL-R a**. This results in the following:

```
All men^H^H^Hpeople are created equal (original line)  
All people are created equal (CTRL-Ra line)
```

The Vim Tutorial and Reference

```
All men^H^H^Hpeople are created equal (CTRL-R CTRL-R a)
```

Leaving Insert Mode

The command **CTRL-\ CTRL-N** ends insert mode and goes to normal mode. In other words, it acts like **<Esc>**. The only advantage this has over **<Esc>** is that it works in all modes.

Finally, **CTRL-O** executes a single normal-mode command and goes back to insert mode. If you are in insert mode, for instance, and type **CTRL-O dw**, the *Vim* editor goes into normal mode, deletes a word (**dw**), and then returns to insert mode.

The .viminfo File

The problem with global marks is that they disappear when you exit *Vim*. It would be nice if they stuck around. The *.viminfo* file is designed to store information on marks as well as the following:

- Command-line history
- Search-string history
- Input-line history
- Registers
- Marks
- Buffer list
- Global variables

The trick is that you have to enable it. This is done through the following command:

```
:set viminfo={string}
```

The *{string}* specifies what to save. (The '**viminfo**' option can be abbreviated as '**vi**'.)

The syntax of this string is an option character followed by an argument. The option/argument pairs are separated by commas.

Let's take a look at how you can build up your own '**viminfo**' string.

The Vim Tutorial and Reference

First, the ' (single quote) option is used to specify how many files for which you save local marks (**a-z**). Pick a nice round number for this option (1000, for instance). Your '**viminfo**' option now looks like this:

```
:set viminfo='1000
```

The **f** option controls whether global marks (**A-z 0-9**) are stored. If this option is **0**, none are stored. If it is **1** or you do not specify an **f** option, the marks are stored. You want this feature, so now you have this:

```
:set viminfo='1000,f1
```

The **r** option tells *Vim* about removable media. Marks for files on removable media are not stored. The idea here is that jump to mark is a difficult command to execute if the file is on a floppy disk that you have left in your top desk drawer at home. You can specify the **r** option multiple times; therefore, if you are on a Microsoft Windows system, you can tell *Vim* that floppy disks A and B are removable with the **r** option:

```
:set viminfo='1000,f1,rA:,rB:
```

UNIX has no standard naming convention for floppy disks. On my system, however, the floppy disk is named */mnt/floppy*; therefore, to exclude it, I use this option:

```
:set viminfo='1000,f1,r/mnt/floppy
```

Note: There is a 50-character limit on the names of the removable media.

The " option (which must be escaped (\") to let *Vim* know it's not a comment) controls how many lines are saved for each of the registers. By default, all the lines are saved. If **0**, nothing is saved. You like the default, so you will not be adding a \" specification to the '**viminfo**' line.

The : option controls the number of lines of : history to save. 100 is enough for us:

```
:set viminfo='1000,f1,r/mnt/floppy,:100,
```

The / option defines the size of the search history. Again 100 is plenty:

```
:set viminfo='1000,f1,r/mnt/floppy,:100,/100
```

Note that *Vim* will never store more lines than it remembered. This is set with the '**history**' ('**hi**') option.

The Vim Tutorial and Reference

Generally, when *Vim* starts, if you have the '**hlsearch**' ('**hi**') option set, the editor highlights the previous search string (left over from the previous editing sessions). To turn off this feature, put the **h** flag in your '**viminfo**' option list. (Or you can just start *Vim*, see the highlighting, and decide you do not like it and execute a **:nohlsearch** (**:noh**)).

The '**@**' option controls the number of items to save in the input-line history. (The input history records anything you type as result of an input function call.) For this example, let this default to the size of the input-line history.

If the '**%**' option is present, save and restore the buffer list. The buffer list is restored only if you do not specify a file to edit on the command line:

```
:set viminfo='1000,f1,r/mnt/floppy,:100,/100,%
```

The **!** option saves and restores global variables. (These are variables whose names are all uppercase.)

```
:set viminfo='1000,f1,r/mnt/floppy,:100,/100,%,!
```

Finally, the **n** option specifies the name of the '**viminfo**' file. By default, this is **\$HOME/.viminfo** on UNIX. On Microsoft Windows, the file is as follows:

\$HOME\viminfo if **\$HOME** is set

\$VIM\viminfo if **\$VIM** is set otherwise

C:_viminfo

The '**n**' option must be the last option parameter. Because we like the default filename, we leave this option off. Therefore, the full '**viminfo**' line is this:

```
:set viminfo='1000,f1,r/mnt/floppy,:100,/100,%,!
```

You can put this command and other initializations into a **.vimrc** initialization file. The **.viminfo** file is automatically written when the editor exits, and read upon initialization. But you may want to write and read it explicitly. The following command writes the **.viminfo** file:

```
:wviminfo[!] [file]
```

(**:wv** is the short version of this command.)

If a file is specified, the information is written to that file. Similarly, you can read the **.viminfo** file using this command:

```
:rviminfo [file]
```

(**:rv** for short.)

This reads all the settings from file. If any settings conflict with currently existing settings, however, the file settings will not be used. If you want the information in the `.viminfo` file to override the current settings, use the following command:

```
:rviminfo! [file]
```

Dealing with Long Lines

Sometimes you will be editing a file that is wider than the number of columns in the window. When that occurs, *Vim* wraps the lines so that everything fits on the screen (see Figure 20-6).

A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters.

Figure 20-6: Text wrapping.

If you set the '**nowrap**' option, each line in the file shows up as one line on the screen. Then the ends of the long lines disappear off the screen to the right (see Figure 20-7).

A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters.

Figure 20-7: `:set nowrap`.

By default, *Vim* does not display a horizontal scrollbar on the GUI. If you want to enable one, as shown in Figure 20-8, use the following command:

```
:set guioptions+=b
```

('guioptions' can be abbreviated as 'go').

The Vim Tutorial and Reference

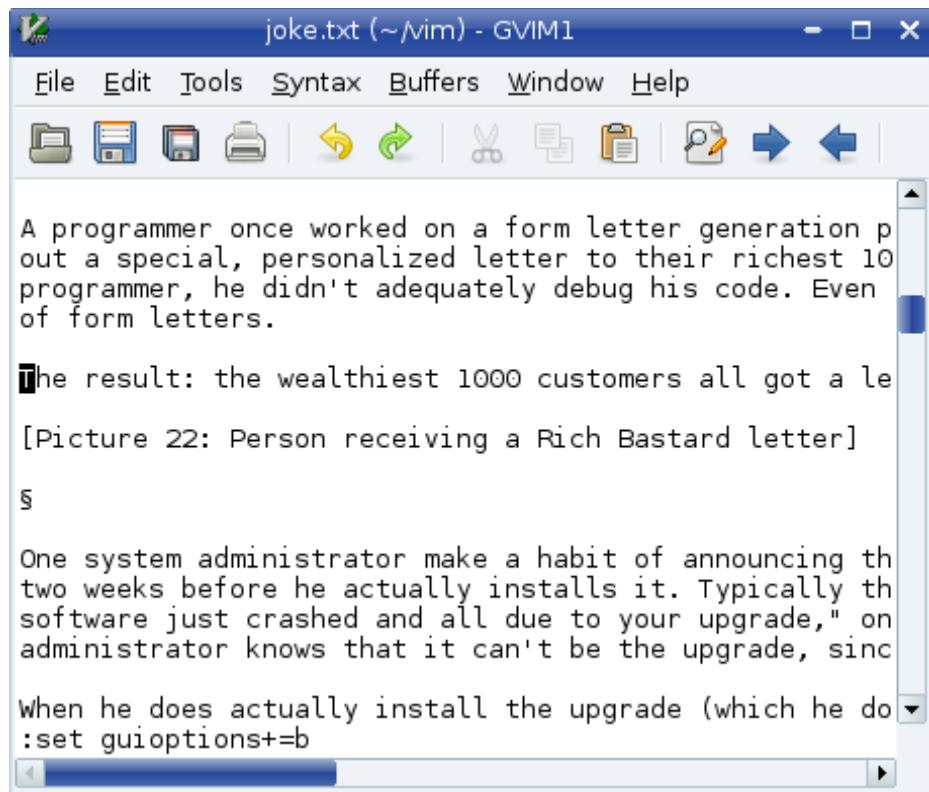


Figure 20-8: Horizontal scrollbar.

This window can be scrolled horizontally. All you have to do is position the cursor on a long line and move to the right using the **1** or **\$** command. Figure 20-9 shows what happens when you do a little horizontal scrolling.

The Vim Tutorial and Reference

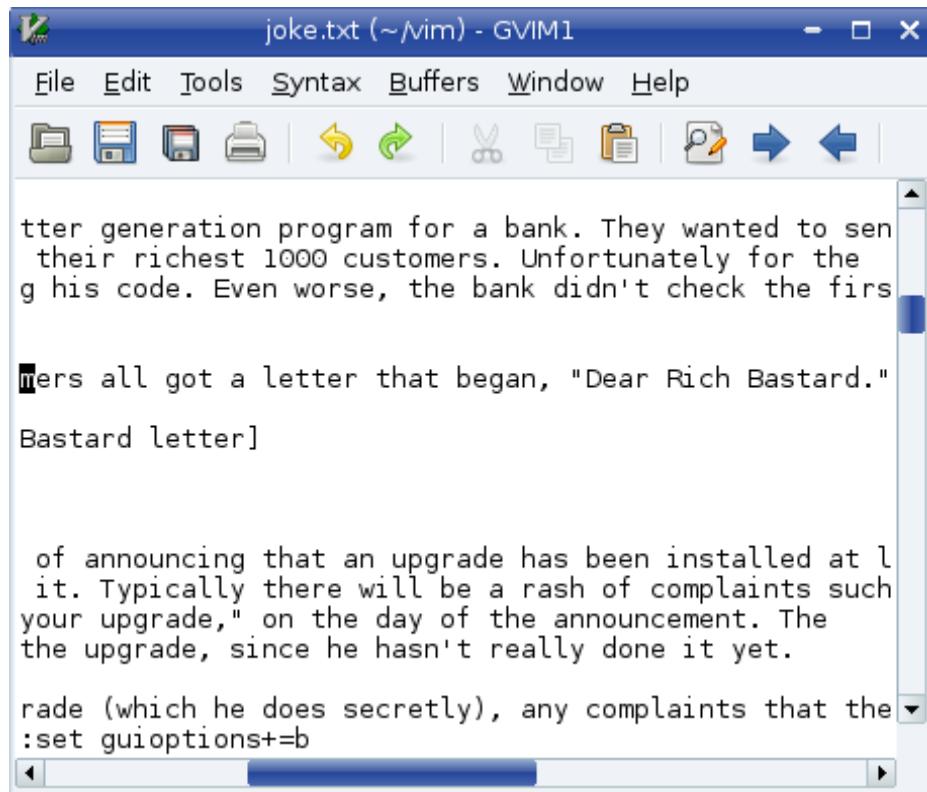


Figure 20-9: Horizontal scrolling.

The `^` command moves to the first non-blank character of the line. The `g^` command moves to the first non-blank character on the screen. If there is text to the left of the window, it is ignored.

There are a number of similar `g`-type commands:

Command	Command	Meaning (When nowrap Set)
<code>^</code>	<code>g^</code>	Leftmost non-blank character on the screen.
<code><Home></code>	<code>g<Home></code>	
<code>0</code>	<code>g0</code>	Leftmost character on the screen.
<code><End></code>	<code>g<End></code>	
<code>\$</code>	<code>g\$</code>	Rightmost character on the screen.
	<code>gm</code>	Move to the middle of the screen

Figure 20-10 shows how these commands work.

The Vim Tutorial and Reference

A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters. The result: the wealthiest 1000 customers all got a letter that began, "Dear Rich Bastard."



Figure 20-10: Line-movement commands.

The **[count] |** command goes to the count column on the screen.

The **[count] zh** command scrolls the screen **count** characters left while the **zL** command does the same thing to the right.

The **zL** command scrolls half a screen to the left and the **zR** command scrolls half screen to the right.

The **j** or **<Down>** command moves down a line. These commands move down lines in the file. Take a look at Figure 20-11.

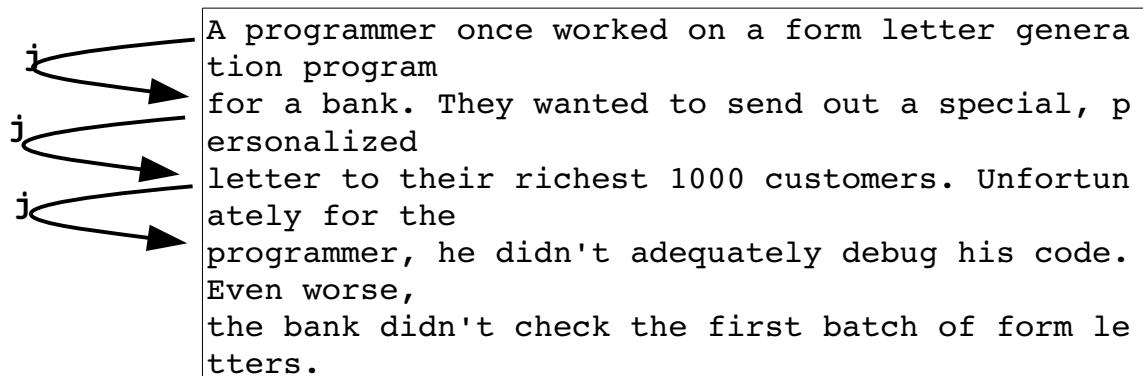
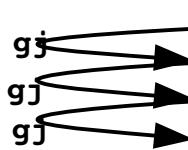


Figure 20-11: The **j** (down) command.

In this case, line 3 has wrapped. Now you start with the cursor on line 2. Executing a **j** command moves you to the beginning of line 3. Another **j** and you are down to the beginning of line 4. Note that although you have moved down a line in text space, you have moved down two lines in screen space.

Typing a **gj** or **g<Down>** command moves one line down in screen space. Therefore, if you start at the beginning of line 3 and type **gj**, you wind up one line down in screen space (see Figure 20-12). This is halfway between line 3 and line 4. (In file space, you are on the middle of line 3.) The **gk** and **g<Up>** commands do the same thing going up.

The Vim Tutorial and Reference



A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters.

Figure 20-12: The **gj** (down screen line) command.

Wrapping

By default, the *Vim* editor wraps long lines. It does this by putting as much of the line as possible on the first screen line, and then to breaking it, and putting the rest on the next line. You can turn this off by setting the following option:

```
:set nowrap
```

With this option set, long lines just disappear off the right side of the screen. When you move the cursor along them, the screen scrolls horizontally and you can see what you are doing. You can customize wrapping by setting some *Vim* options. First of all, you can tell *Vim* to break lines at nice places by setting the option:

```
:set linebreak      or  
:set lbr
```

Figure 20-13 shows how this option affects the screen.

A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters.

```
:set nolinebreak
```

A programmer once worked on a form letter generation program for a bank. They wanted to send out a special, personalized letter to their richest 1000 customers. Unfortunately for the programmer, he didn't adequately debug his code. Even worse, the bank didn't check the first batch of form letters.

```
:set linebreak
```

Figure 20-13: The '**linebreak**' option.

The Vim Tutorial and Reference

But what defines a "nice" place on the line. The answer is the characters in the '**breakat**' ('**brk**') option. By default, these are ^ I ! @ * - + ; : , . / ?. Now suppose you do not want to break words with _ in them. You need to remove _ from the list of '**breakat**' characters, so you execute the following command:

```
:set breakat -=_
```

Usually when lines are broken, nothing is put at the beginning of the continuation lines. You can change this, however, by defining the '**showbreak**' ('**sbr**') option. For example:

```
:set showbreak="---->"
```

Finally, there is the question of what to do if you need to break a line at the end of the screen. You have two choices: First, you can refuse to display half of a line. The *Vim* editor will display an @ at the bottom of the screen to indicate "there is a long line here that we cannot fit it on the screen. Second, you can display half the line. The *Vim* default is method one. If you want to use method two, execute this command:

```
:set display=lastline
```

('dy' is short for '**display**'.)

Spelling Dictionaries

Vim lets you create word lists in a variety of formats. The simplest is a straight word list. To turn a word list into a dictionary it needs to be compiled. This is done using the **:mkspell** (**:mksp**) command.

The general form of this command is:

```
:mkspell {out-file} {in-file}
```

There are a lot of rules concerning the name of the output file. First if the name ends in *.spl*, the **{out-file}** name is the actual name of the output file. If it does not, then it is the base name to be used. *Vim* will add the current encoding to the name as well as the extension *.spl*.

The **:mkspell** command takes an option (**-ascii**) which tells it to skip all non-ASCII characters.

Vim accepts word lists in a variety of formats. One of the more common dictionary layouts is the MySpell format. One source for dictionary files is http://lingucomponent.openoffice.org/spell_dic.html

The Vim Tutorial and Reference

Vim supports both simple word lists, and a very complex word lists. The latter specify things language and region, as well as indicating word parts (prefixes, suffixes), rare words, and misspelled words. Since this feature is of interest only to the handful users who will actually make word lists, a full description has been omitted.

The `:mkspell` command can take multiple input files and produce a single word list. The naming of the input files must follow the convention `{name}_{encoding}`. The following command creates a English language dictionary for three regions.

```
:mkspell en en_US en_CA en_AU
```

Finally if no output name is specified, the `:mkspell` command will produce one based on the input name.

Dictionary construction can be fine tuned through the '`mkspellmem`' ('`msm`') option. This option consists of three memory limits. The first tells `:mkspell` how much memory to use before starting compression, the second specifies the amount of memory that can be allocated before another compression is done and the last is the upper limit on memory. When memory reaches this limit aggressive (and time consuming) compression is started.

Dumping dictionaries

If you need to see a list of the words in the current dictionary use the `:spelldump` (`:spelld`) command. This opens a new window with the list of words in it.

What's the format of this list? It is not documented.

If you use the override (!) option on this command, the words along with a word count is dumped.

Documentation does not match results.

Customizing the spelling system

The '`spellcapcheck`' ('`spc`') option is used by `vim` to tell when a sentence ends. It contains a regular expression that is executed to detect periods and other sentence ending text for the spelling system's capitalization check.

The '`spellsuggest`' ('`sps`') is a option which contains information that tells *Vim* how to find suggested corrections for a word. It is a set of keywords separated by commas. The list of keywords includes:

The Vim Tutorial and Reference

best	Use the suggesting generator named “best”. This is the generator that works best for English.
double	This suggestion generator first tries the “best” method, then depending on the results, then tries the “fast” method and mixes the results. It can be somewhat slow and not as good as the other methods.
fast	This suggestion generator checks for dropped, added, or moved letters. (In other words typos.)
{number}	Maximum number of suggestions to generate.
file:{filename}	Specify a suggestion file. Each line in the file contains a misspelled word a slash and the correct version.
expr:{expression}	Evaluate the {expression} to get a list of suggestions and their scores. (Note: Lower scores are better.)

Chapter 21: All About Windows, Tabs, and Sessions

In *Chapter 5: Windows and Tabs* you learned the basic commands for using windows. But there are a lot more window-related commands. This chapter discusses many different commands for selecting and arranging windows. You will also learn how to customize the appearance of the windows.

Finally, this chapter discusses session files. These files enable you to save and restore all your editing and window settings so that you can return to editing where you left off.

The topics covered in this chapter include the following:

- Moving between windows
- Moving windows up and down
- Performing operations on all windows
- Editing the "alternate" file
- Split searches
- Shorthand operators
- Advanced buffer commands
- Session files

Moving Between Windows

As previously discussed, **CTRL-Wj** (**CTRL-W CTRL-J**, **CTRL-W<Down>**) goes to the window below and **CTRL-Wk** (**CTRL-W CTRL-K**, **CRL-W<Up>**) goes to the window above. The following commands also change windows.

CTRL-Wt Go to the top window.

CTRL-W CTRL-T

CTRL-Wb Go to the bottom window.

CTRL-W CTRL-B

CTRL-Wp Go to the window you were in before you switched to this
CTRL-W CTRL-P one. (Go to the preceding window.)

The Vim Tutorial and Reference

```
/* File one.c */
~
~
one.c
/* File two.c */
~
two.c
/* File three.c */
#include <stdio.h>
int i;
three.c
```

CTRL-Ww *[count]*CTRL-Ww Go down a window. If at the bottom, wrap. If *count* is specified, go to the window number *count*.
1CTRL-WW [count]CTRL-W CTRL-W Go up a window. If at the top, wrap. If *count* is specified, go to the window number *count*.
CTRL-Wt
2CTRL-Ww
2CTRL-WW
3CTRL-Ww
3CTRL-WW
CTRL-Wb

Figure 21-1: Window selection commands.

[*count*]CTRL-Ww Go down a window. If at the bottom, wrap. If *count* is specified, go to the window number *count*.
[*count*]CTRL-W CTRL-W Go up a window. If at the top, wrap. If *count* is specified, go to the window number *count*.

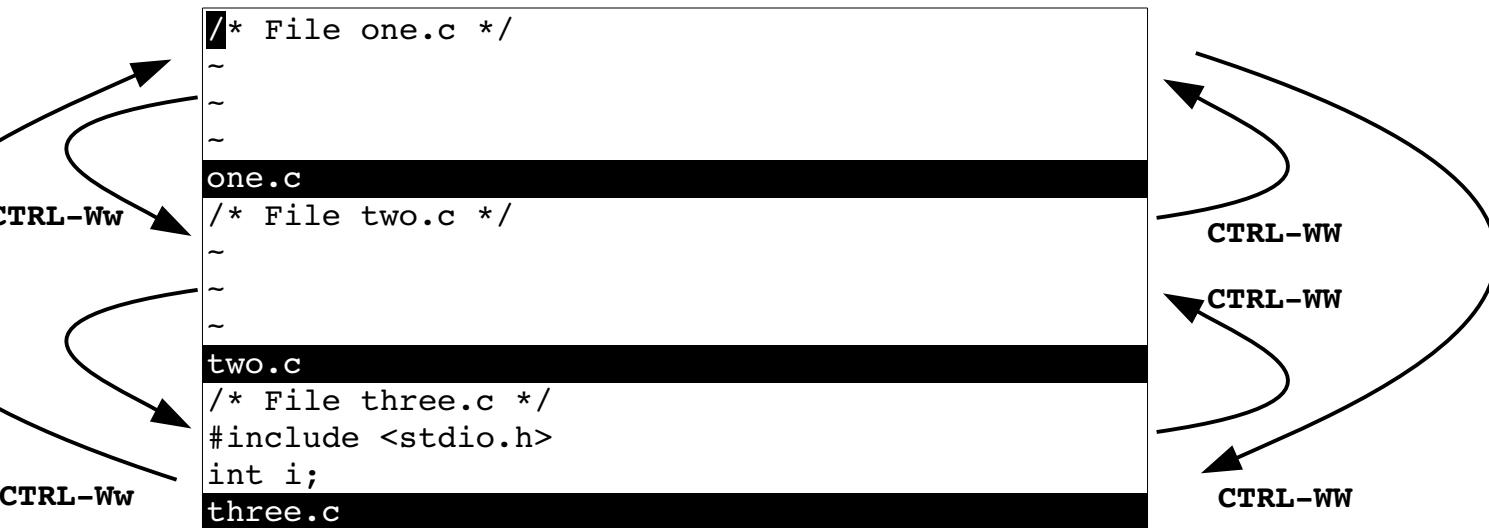


Figure 21-2: More window selection commands.

Moving Windows Up and Down

The **CTRL-Wr** (**CTRL-W CTRL-R**) command rotates the windows downward (see Figure 21-3).

The Vim Tutorial and Reference

The **CTRL-Wr** command takes a *count* argument, which is the number of times to perform the rotate down.

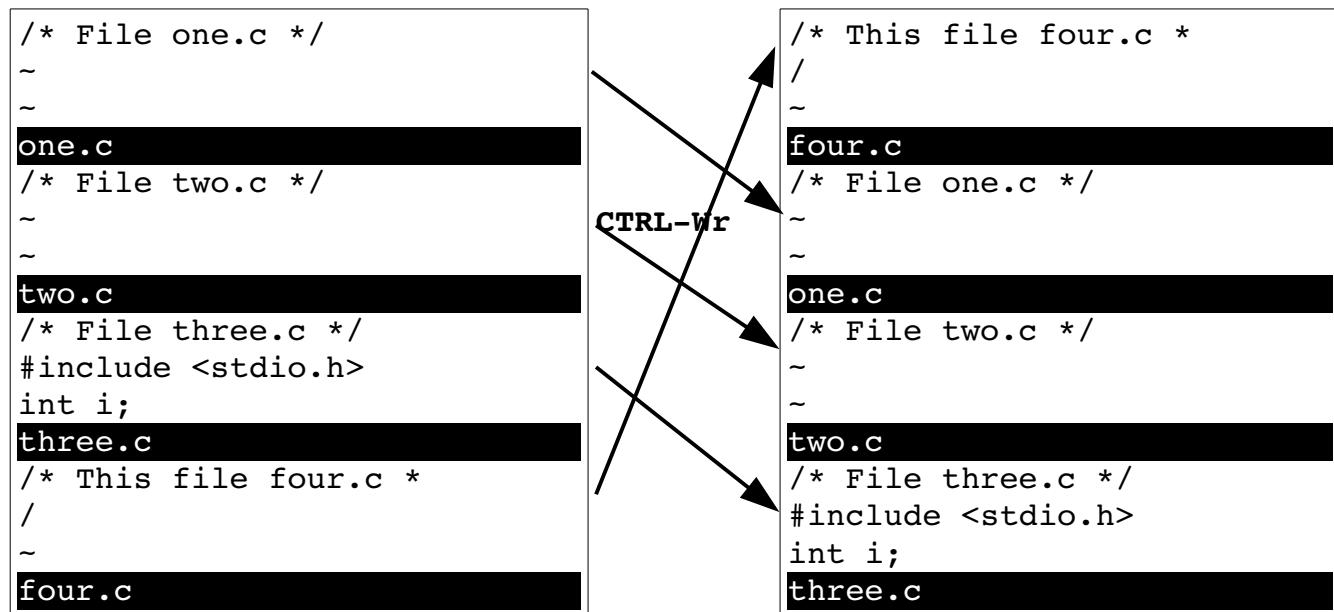


Figure 21-3: Rotating a window down.

The **CTRL-WR** command rotates the windows upward (see Figure 21-4).

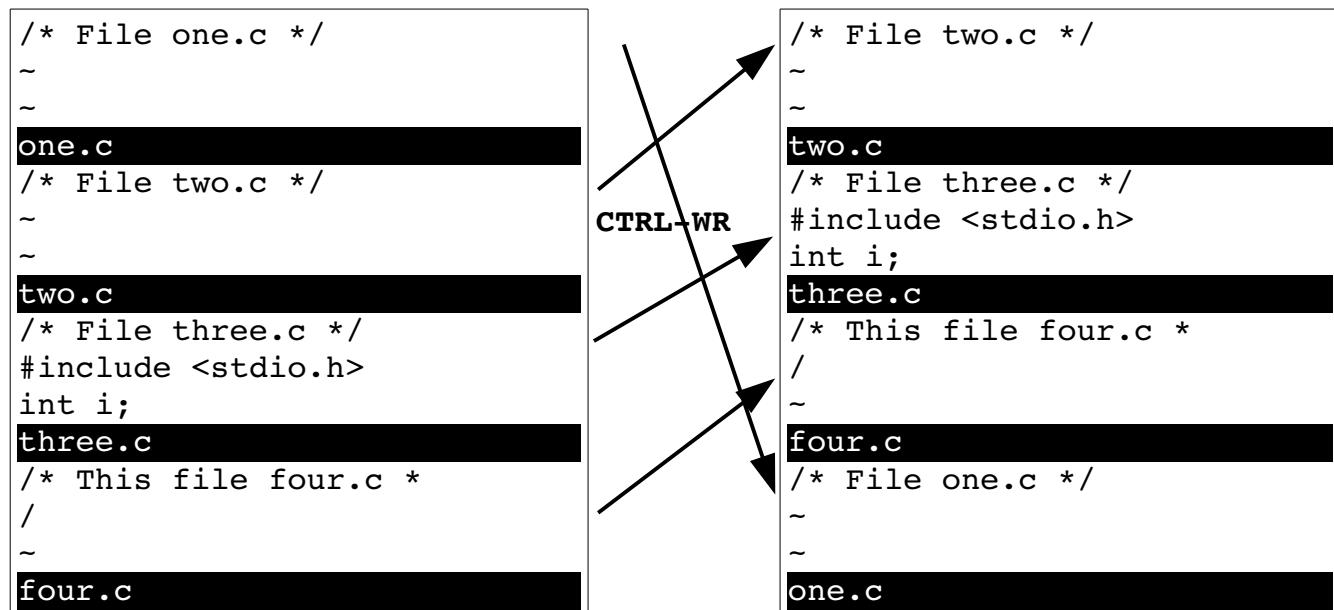


Figure 21-4: Rotating a window up.

The Vim Tutorial and Reference

The **CTRL-Wx (CTRL-W CTRL-X)** command exchanges the current window with the next one (see Figure 21-5). If the current window is the bottom window, there is no next window, so it exchanges the current window with the previous one.

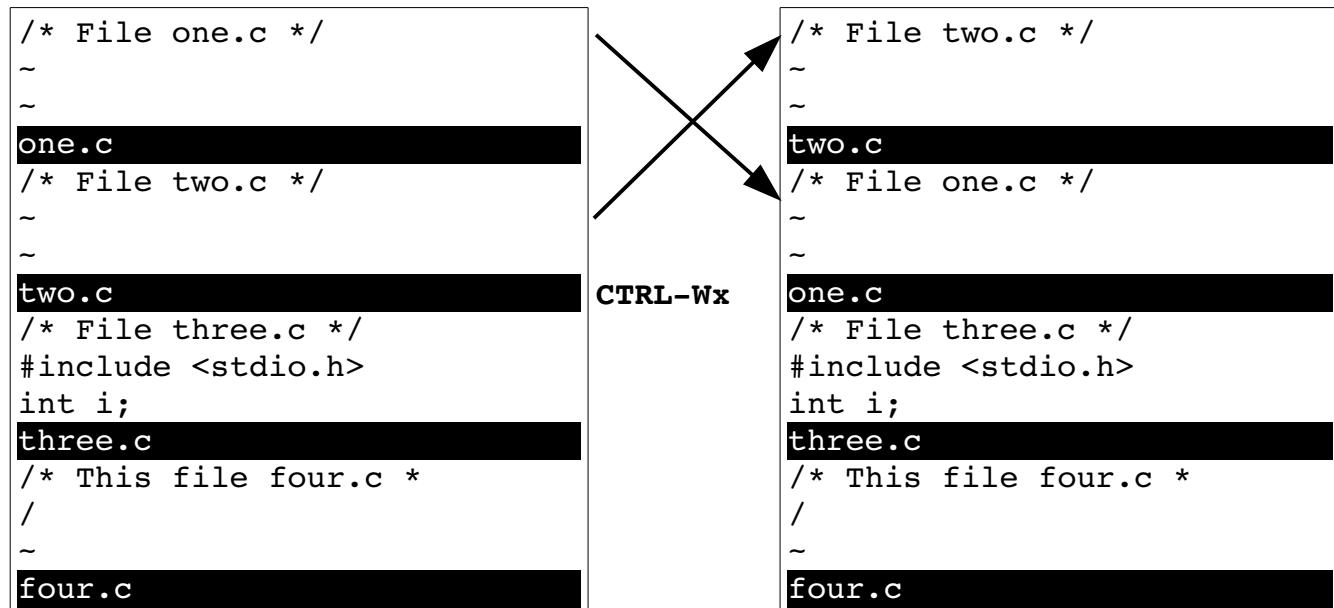


Figure 21-5: Exchanging a window.

Performing Operations on All Windows

The **:write (:w)** command writes out the current file. If you want to write all the files that have been modified (including hidden buffers), use the following command:

```
:wall
```

(**:wa** is the short form of **:wall**.)

The quit command **:quit (:q, CTRL-Wq, CTRL-W CTRL-Q)** closes the current window. (If this is the last window for a file, the file is closed.) If you have multiple windows up, you can quit them all using this command:

```
:qall
```

(**:qa** is the short form of **:qall**.)

If some of the files have been modified, and the changes have not been saved, the **:qall** command fails. If you want to abandon the changes you have made, use the force option (!), which gives you this command:

```
:qall!
```

The Vim Tutorial and Reference

(Use this with caution because you can easily discard work you wanted to keep.) If you want to perform a combination of `:wall` and `:qall`, use this command:

```
:wqall
```

(The aliases for `:wqall` are `:wqa`, `:xa`, `:xall`.)

Other Window Commands

The **CTRL-Wo** (**CTRL-W** **CTRL-O**, `:on`, `:only`) command makes the current window the only one on the screen. As Figure 21-6 shows, all the other windows are closed. (The system pretends that you did a `:quit` in each of them.)

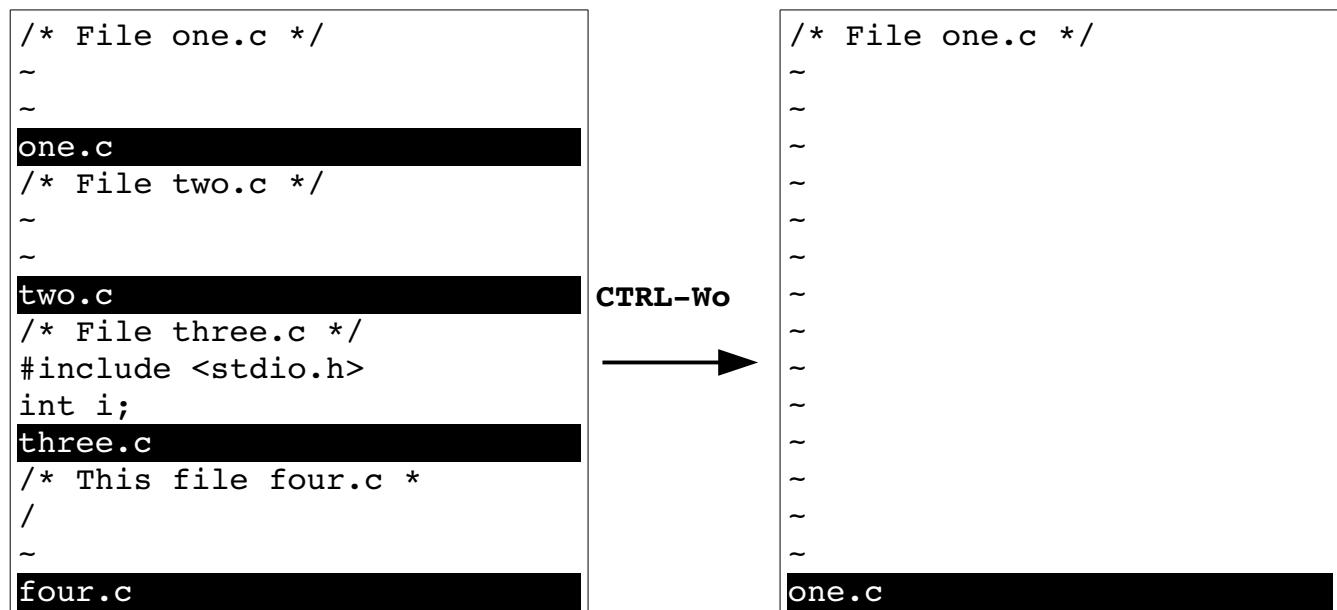


Figure 21-6: The **CTRL-Wo** command.

If you have specified multiple files on the command line or through the file-list command, the `:all` (`:al`, `:sal`, `:sal`) command opens up a window for each file (see Figure 21-7).

The Vim Tutorial and Reference

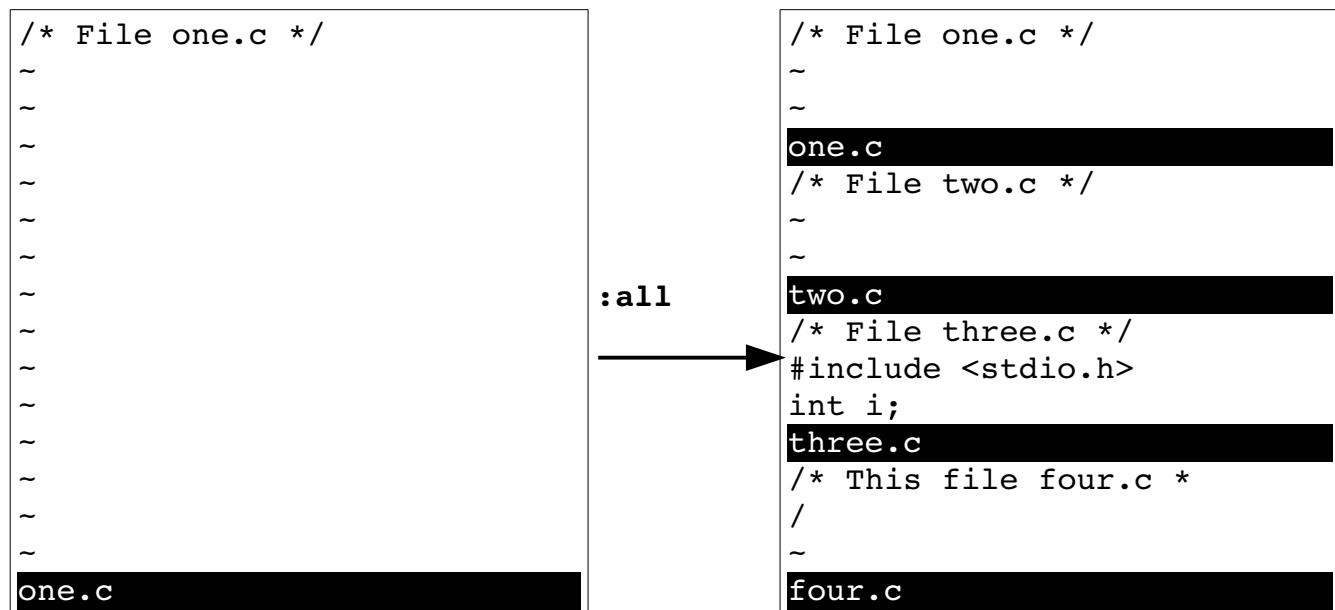


Figure 21-7: :all.

A variation of the `:all` command opens a new window for each hidden buffer:

:unhide

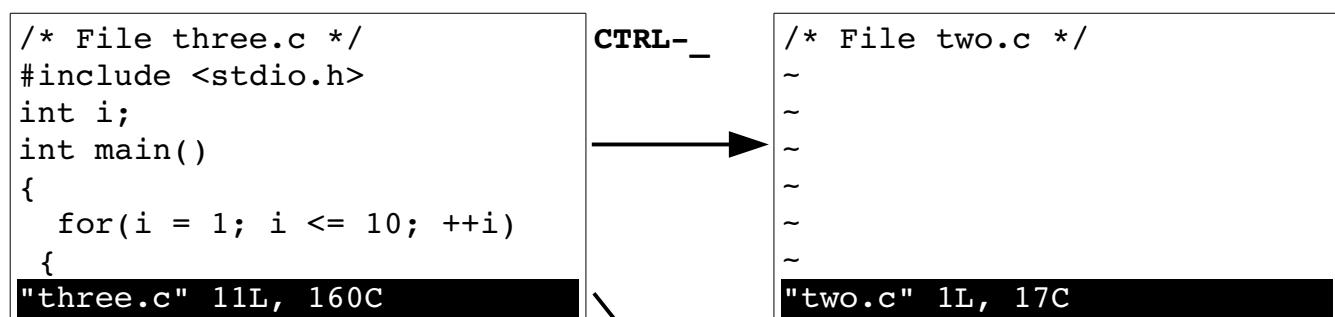
(Alternate formats :unh, :sunhide, :su.)

This command can take an argument that limits the number of windows that can be opened at one time. To unhide all the buffers but put no more than five windows on screen, for example, use the following command:

:unhide 5

Editing the Alternate File

You can split the window and edit the alternate file with the command **CTRL-W CTRL-^** (**CTRL-W^**). Figure 21-8 shows the results.



The Vim Tutorial and Reference

```
/* File two.c */
~
~
CTRL-W
CTRL-_
/* File three.c */
#include <stdio.h>
int i;
 "three.c" 11L, 160C |
```

Figure 21-8: **CTRL-W CTRL-^**.

Split Search

The **CTRL-W CTRL-I** command splits the window, and then searches for the first occurrence of the word under the cursor. This search goes through not only the current file, but also any `#include` files.

If you position the cursor on the `printf` on “Hello World” and press **CTRL-W CTRL-I**, you get a screen that looks like Figure 21-9.

```
extern int fprintf(...);

/* Write formatted output to stdout. */
extern int printf (...);
/* Write formatted output to s. */
extern int sprintf (...)

/usr/include/stdio.h [RO]
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return (0);
}
/tmp/hello.c
```

Figure 21-9: The **CTRL-W CTRL-I** command.

Shorthand Commands

The *Vim* editor contains some shorthand commands that do the work of multiple commands, including the following:

:{count}snext	:split followed by :countnext
:{count}sn	
:{count}sprevious	:split followed by :countprevious

The Vim Tutorial and Reference

:{count}spr	
:{count}sNext	:split followed by :countNext
:{count}sN	
:srewind	:split followed by :rewind
:sre	
:sfirst	
:sf	
:slast	:split followed by :last
:sl	
:sargument	:split followed by :argument
:sa	
CTRL-W CTRL-D	:split followed by]CTRL-D
CTRL-Wd	
CTRL-W CTRL-F	:split followed by a :find
CTRL-Wf	
CTRL-Wg]	:split followed a CTRL-]

One nice thing about these commands is that they do not open a new window if they fail.

Other Window Commands

The :wincmd (:winc) command executes a command as you typed **CTRL-W**. Not that useful interactively but very nice when you are writing scripts.

The :windo (:wind) command will execute a command for each open window. For example, to change “vim” to “Vim” in all the windows use the command:

```
:windo :%s/vim/Vim/g
```

Advanced Buffers

The following sections discuss adding, deleting, and unloading buffers.

Adding a Buffer

The *Vim* editor maintains a list of buffers. Usually you put a file on the list by editing it. But you can explicitly add it with the following command:

```
:badd file
```

(This is a bad command. Let me rephrase that. The abbreviated version of this command is :bad.)

The Vim Tutorial and Reference

The named file is merely added to the list of buffers. The editing process will not start until you switch to the buffer. This command accepts an argument:

```
:badd +lnum file
```

When you open a window for the buffer, the cursor will be positioned on line *lnum*.

Deleting a Buffer

The

```
:bdelete
```

command deletes a buffer. (**:bd** is the short form.)

You can specify the buffer by name:

```
:bdelete file.c
```

or by number:

```
:bdelete 3  
:3 bdelete
```

You can also delete a whole range of buffers, as follows:

```
:1,3 bdelete
```

If you use the override (!) option, any changes to the buffer are discarded:

```
:bdelete! file.c
```

Unloading a Buffer

The command **:bunload** (**:bun**) unloads a buffer. The buffer is unloaded from memory and all windows for this buffer are closed. However, the file remains listed in the buffer list. The **:bunload** command uses the same syntax as the **:bdelete**.

The **:bwipeout** (**:bw**) command unloads the file and causes *Vim* to forget everything it knew about it. This command takes the same syntax as **:bdelete** and should not be used unless you know what you are doing.

Opening a Window for Each Buffer

The **:ball** (**:ba**, **:sball**, **:sba**) command opens a window for each buffer.

Windowing Options

The '**laststatus**' ('**ls**') option controls whether the last window has a status line. (See Figure 21-10.) The three values of this option are as follows:

- 0 The last window never has a status line.
- 1 If there is only one window on the screen, do not display the status line. If there are two or more, however, display a status line for the last window. (default).
- 2 Always display a status line even if there is only one window on screen.

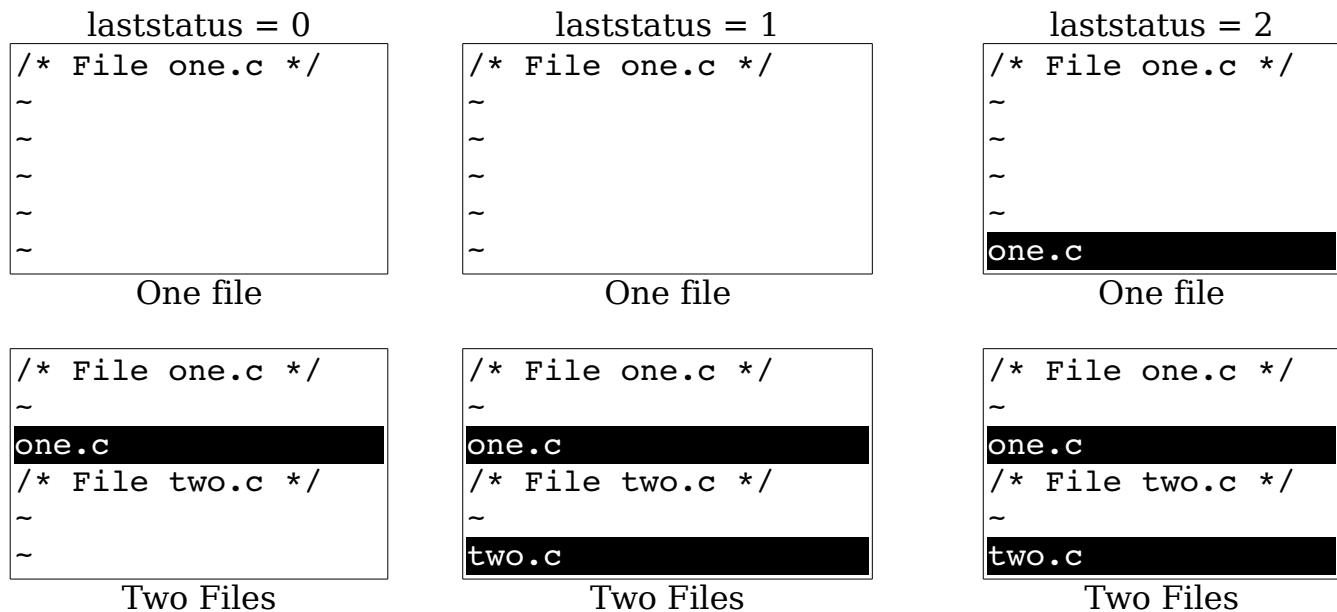


Figure 21-10: '**laststatus**' option.

The '**winheight**' ('**wh**') option sets the minimum number of lines for a window. This is not a hard limit; if things get too crowded, *Vim* will make smaller windows. The '**winwidth**' ('**wiw**') option does the same thing, only for width.

The '**winminheight**' ('**wmh**') option is the absolute minimum height of a window that's not the current window. The '**winminwidth**' ('**wmw**') does the same thing for the width of a window.

When the '**equalalways**' ('**ea**') option is enabled (the default), *Vim* will always split the screen into equal-size windows. When off, splits can result in windows of different sizes. Figure 21-11 shows the effect of this option.

The Vim Tutorial and Reference

```
gvim one.c
```

```
:split
```

```
:split
```

```
/* File one.c */  
~  
~  
~  
~  
one.c  
/* File one.c */  
~  
~  
~  
~  
one.c  
/* File one.c */  
~  
~  
~  
~  
one.c  
:split
```

```
:set equalalways
```

```
/* File one.c */
```

```
~
```

```
~
```

```
~
```

one.c

```
/* File one.c */
```

```
~
```

```
~o
```

one.c

```
/* File one.c */
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

one.c

```
:split
```

```
:set noequalalways
```

Figure 21-11: '**equalalways**' option.

By default '**equalalways**' works for horizontal and vertical splits. If you want this option to affect only horizontal windows set the '**eadiirection**' ('ead') option to **hor**. If you want only vertical windows to be affected set the '**eadiirection**' option to **ver**. The default (**both**) affects both directions.

If you split horizontally, and the '**equalalways**' option is set, any window with the '**wifixheight**' ('**wfh**') option set will not change size. For vertical splits the '**wifixwidth**' ('**wfw**') option keeps the size constant.

Controloing a split

Generally a **:split** command opens a window above the current window. The '**splitbelow**' ('**sb**') option causes a new window to appear below the current one.

The Vim Tutorial and Reference

A **:vsplit** command opens a window to the left of the current one. The '**splightright**' ('**spr**') option causes new to open to the right.

The **:aboveleft** (**:abo**, **:leftabove**, **:lefta**) command causes any command that splits a window to open the window above the current one or to the left regardless of the value of any options.

The **:rightbelow** (**:rightb**, **:belowright**, **:bel**) command does the same thing except the window will appear below or to the right of the current one.

The **:topleft** (**:to**) command will open a window at the top of the screen for horizontal splitting commands. The window will go across the entire screen. For vertical splitting commands, the window will occupy the entire left of the screen.

The **:botright** (**:bo**) command acts in a similar manner except the window will occupy the entire bottom or right of the screen.

Tabs

In this section we'll take a look at how to execute one command for multiple tabs as well as how to customize the tab system.

Executing a command for all tabs

The **:tabdo** (**:tabd**) command executes a given command for every tab. For example, to change the word "idiot" to "politician" for every tab, use the command:

```
:tabdo :s/idiot/politician/
```

Other tab commands

The **:tabs** command lists the tabs and the files being edited in them.

For example:

The Vim Tutorial and Reference

```
:tabs
Tab page 1
> + spell.add
Tab page 2
  new.txt
  tabpage.txt
Press ENTER or type command to continue
```

The `:tabmove` (`:tabm`) command moves the current tab to just after the given tab.

For example Figure 21-12 shows what happens when we have the tab *third.txt* open and execute the command:

```
:tabmove 1
```

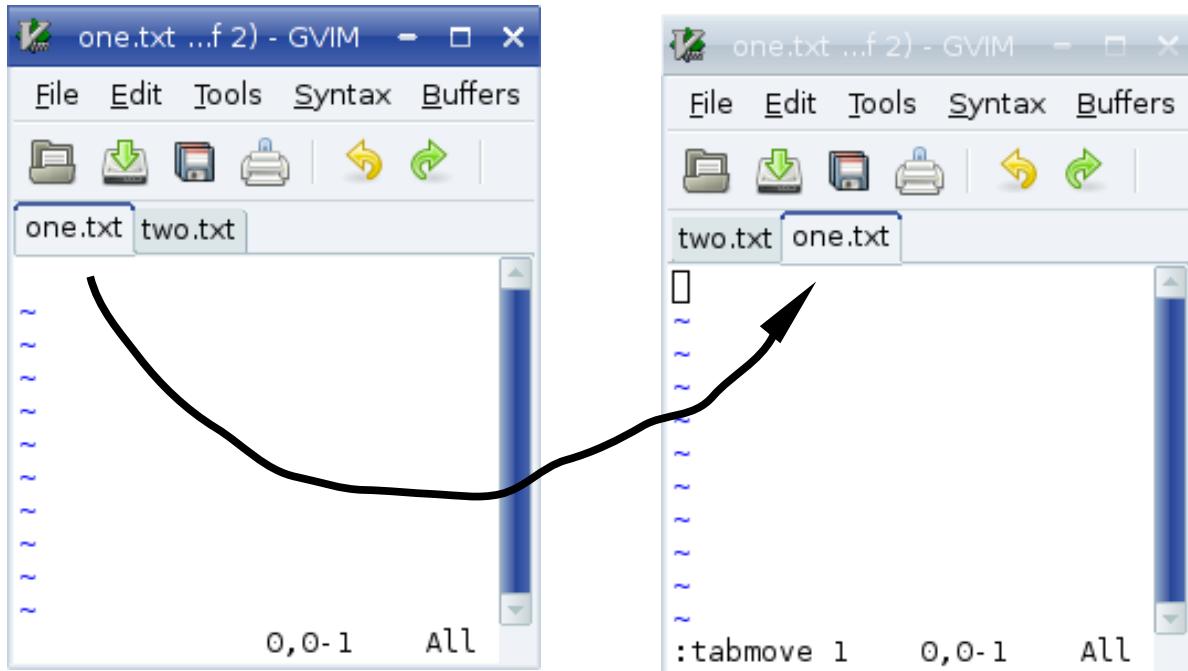


Figure 21-12: Result of `:tabmove 1`

Customizing tabs

The `'guitablabel'` (`'gtl'`) contains a string that defines the format of the label for each tab. The format is the same as the `'statusline'` option described on page 545.

The `'guitabtooltip'` (`'gtt'`) option is used to define a tooltip for each tab.

The Vim Tutorial and Reference

Both '**guitablabel**' and '**guttabtooltip**' are evaluated once per tab.

Tabs without the GUI

Tabs are still available if you are using console (non-GUI) mode. They don't look as nice and you can't click to move from tab to tab, but you do have tabs. (However, you may wish to consider using windows instead.)

The '**showtabline**' ('**stal**') tells *Vim* when to show the tab line. This option has the following values

0 -- Never show the tab line

1 -- Show the tab line only when required

2 -- Always show the tab line

The '**tabline**' ('**tal**') option defines the tabline to be displayed. Unlike '**guitablabel**' this option needs to define the entire tab line. For those of you really interested in this level of customization, there is an example of how to do this in the :**help tabs** help page.

Sessions

Suppose you are editing along, and it is the end of the day. You want to quit work and pick up where you left off the next day. You can do this by saving your editing session and restoring it the next day.

A *Vim* session contains all the information about what you are editing. This includes things such as the file list, windows, and other information. (Exactly what is controlled by the '**sessionoptions**' ('**ssop**') option is described later in the section "Specifying What Is Saved in a Session.")

The following command creates a session file:

```
:mksession file
```

(:**mks** is short for :**mksession**.)

For example:

```
:mksession vimbook.vim
```

Later if you want to restore this session, you can use this command:

```
:source vimbook.vim
```

The Vim Tutorial and Reference

(**:so** is short for **:source**.)

If you want to start *Vim* and restore a specific session, you can use the following command:

```
$ vim -c ":source vimbook.vim"
```

(This tells *Vim* to execute a specific command on startup (**-c**). The command is **:source vimbook.vim**, which loads the session *vimbook.vim*.)

Specifying What Is Saved in a Session

The '**sessionoption**' option controls what is saved in a session file. It is a string of keywords separated by commas. For example, the default '**sessionoptions**' setting is as follows:

```
:set sessionoptions=buffers,winsize,options,help,blank
```

The various keywords are

buffers	Saves all buffers. This includes the ones on the screen as well as the hidden and unloaded buffers.
globals	Saves the global variables that start with an uppercase letter and contain at least one lowercase letter.
help	The help window.
blank	Any blank windows on the screen.
options	All options and keyboard mapping.
winpos	Position of the GUI <i>Vim</i> window.
resize	Size of the screen.
winsize	Window sizes (where possible).
slash	Replace backslashes in filenames with forward slashes. This option is useful if you share session files between UNIX and Microsoft Windows. (You should set UNIX as well.)
unix	Write out the file using the UNIX end-of-line format. This makes the session portable between UNIX and Microsoft Windows.

Note: If you enable both the **slash** and **unix** options, the session files are written out in a format designed for UNIX. The Microsoft Windows version of *Vim* is smart enough to read these files.

Unfortunately, the UNIX version of *Vim* is not smart enough to read Microsoft Windows format session files. Therefore, if you want to have portable sessions, you need to force *Vim* to use the UNIX format.

Views

Let's say you've got a lot of windows and tabs open. You've gone to a lot of trouble to make things look just right. Each window is just in the right place, and the options have been finely tuned to make things just perfect.

Things are great as long as you stay within Vim. But once Vim exists, all these settings are lost.

That's where the **:mkview** (**:mkvie**) command comes in. It creates a script that restores all your windows and settings to their current values. (It's not perfect, but it does a pretty good job.)

To save you view use the command:

```
:mkview view.vim
```

(If the override (!) option is present, **:mkview** will overwrite an existing file.)

The script is called *view.vim*. You can load it with the command:

```
:source view.vim
```

You can also **:mkview** without a file name. In this case the file is stored in the directory specified by the '**viewdir**' ('**vdir**') option. If a number is used as an argument, then a numbered view is stored in this directory.

In any case, these views can be loaded by using the **:loadview** (**:lo**) command.

The '**viewoptions**' ('**vop**') option controls what's stored in the view. It is a comma separated list of keywords:

Keyword	Meaing
cursor	Save the cursor position
folds	Save the current state of the folds
options	Save the current options including the ones local to the buffer or window.
slash	Save file names with backslashed (\) replaced by forward slashes (/).
unix	Save in UNIX file format.

Chapter 22: Advanced Visual Mode

In *Chapter 6: Basic Visual Mode* you learned how to perform visual commands. Now you can take a look at many of the other visual-related commands. Many of these commands have a limited audience; but read on, that audience may include you. In this chapter, you learn about the following:

- Using visual mode with text registers
- Using the `$` selection command
- Reselecting text
- Additional highlighting commands
- Miscellaneous editing commands
- Select mode

Visual Mode and Registers

Chapter 4: Text Blocks and Multiple Files showed you how to use the `yank`, `put`, and `delete` commands with registers. You can do similar things with the visual-mode commands. To delete a block of text, for instance, highlight in visual mode and then use the `d (<Delete>, x)` command. To delete the text into a register, use the command "`{register}d`".

To yank the text into a register, use the `y` command. The `D` and the `Y` commands act like their lowercase counterparts, except they work on entire lines, whereas `d` and `y` work on just the highlighted section.

The `$` Command

In block visual mode, the `$` command causes the selection to be extended to the end of all the lines in the selection. Moving the cursor up or down extends the select text to the end of the line. This extension occurs even if the new lines are longer than the current ones. Figure 22-1 shows what happens when you don't use the `$` command, and Figure 22-2 shows what happens when this command is used.

The Vim Tutorial and Reference

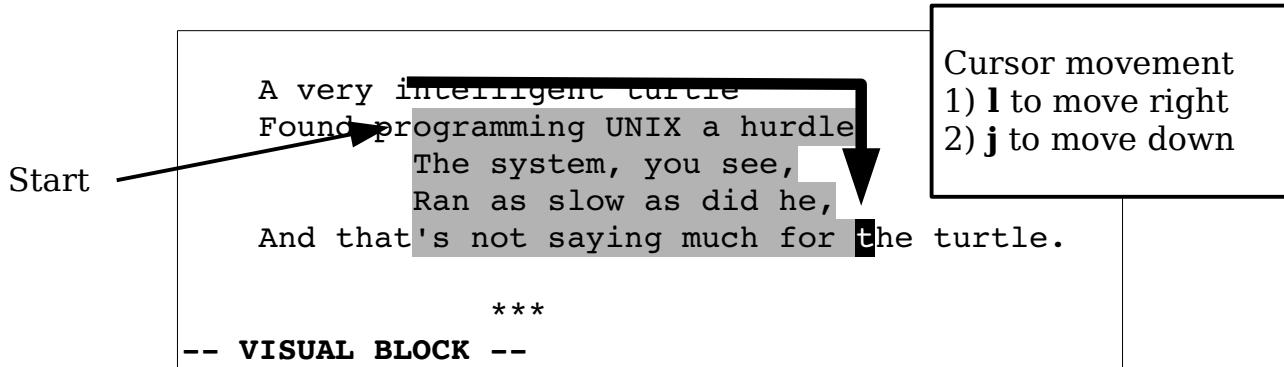


Figure 22-1: Block visual mode without `$` command.

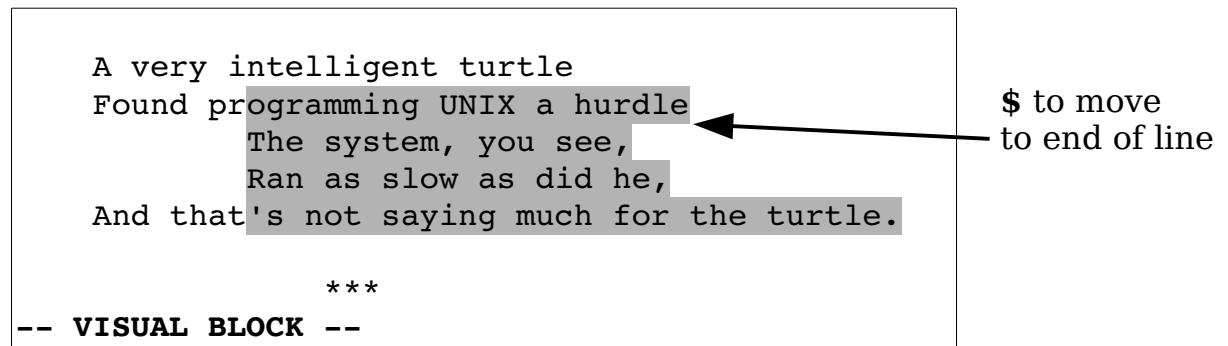


Figure 22-2: Block visual mode with the `$` command.

Repeating a Visual Selection

The **gv** command repeats the preceding visual mode selection. If you are already in visual mode, it selects the preceding selection. Repeated **gv** commands toggle between the current and preceding selection. Figure 22-3 shows the effects of these commands. The steps are as follows:

1. First visual selection.
2. Finished with visual.
3. **gv** reselects the old visual.
4. Define new visual.

The Vim Tutorial and Reference

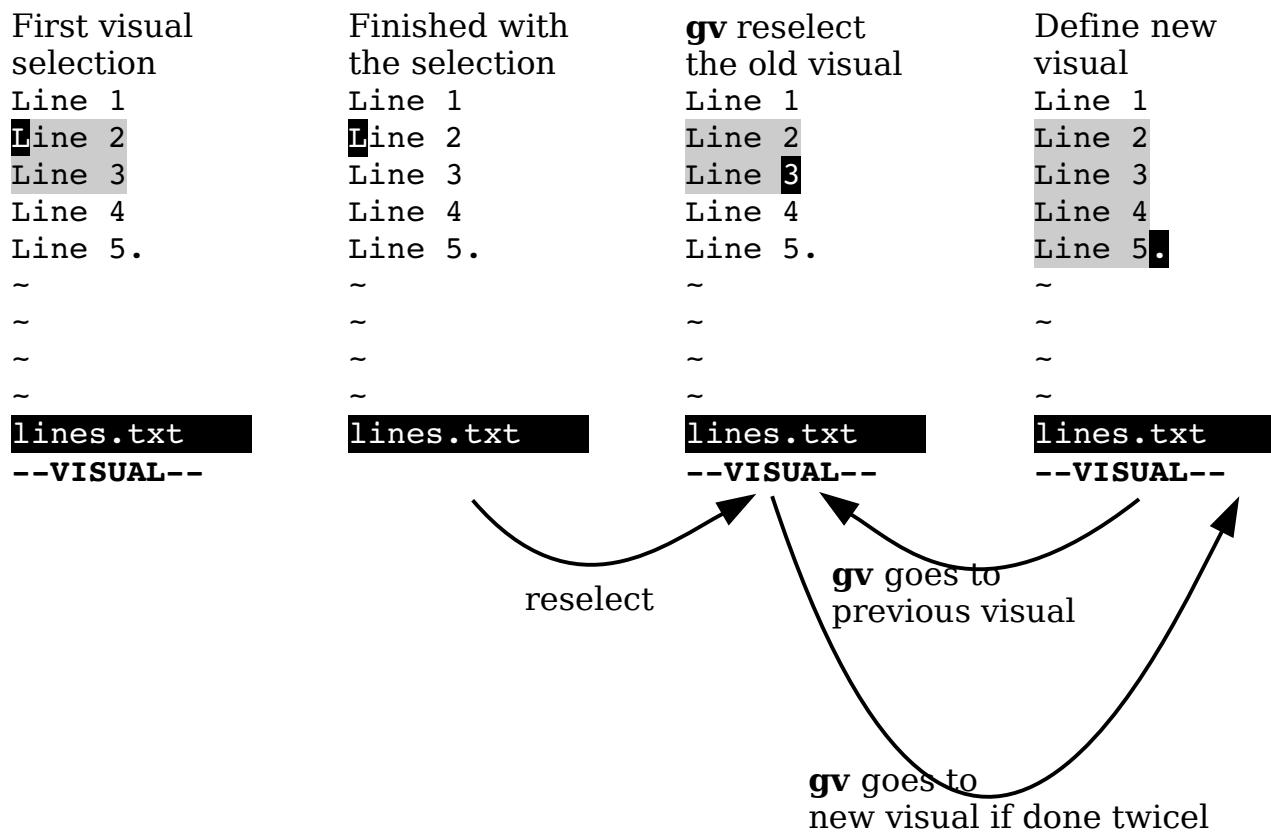


Figure 22-3: The **gv** command.

Selecting Objects

A number of commands in visual mode are designed to help you highlight the text you want.

The **aw** command, for example, highlights the next word. Actually it highlights not only the word, but also the white space after it. At first this may seem a bit useless. After all, the **w** command moves you forward one word, so why not just use it?

That is because when you perform a selection, the text selected is from the old cursor location to the new one inclusive. Now if you use the **w** command to move, the result is that the cursor is placed on the first character of the next word. Therefore if you delete the text, you not only get the words you selected, but the first character of the next word.

The Vim Tutorial and Reference

The **aw** command leaves the cursor positioned just before the first character of the next word. In other words, it selects the word and the spaces beyond it, but not the next word.

Another reason to use **aw** rather than **w** is that **aw** selects the whole word, no matter which part of the word the cursor is on, whereas **w** just selects from the current location to the end of the word.

If you want to just select the word, and nothing but the word, use the **iw** (inner word) command. Figure 22-4 shows how **iw** and **aw** work.

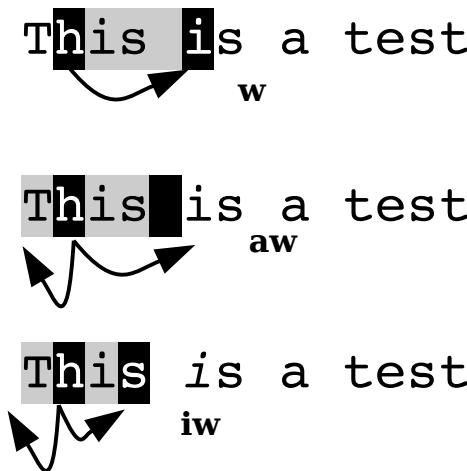


Figure 22-4: **iw** and **aw** commands.

You can use the following commands to select text:

{count}aw	Select a word and the space after it.
{count}iw	Select a word only (inner word).
{count}aW	Select a WORD and the space after it.
{count}iW	Select inner WORD (the word only)
{count}as	Select a sentence (and spaces after it.)
{count}is	Select the sentence only.
{count}ap	Select a paragraph and the following space.
{count}ip	Select a paragraph only.
{count}a(From within text enclosed in (), select the text up to and including the () .
{count}a)	
{count}ab	
{count}i(Like a(), except the () characters are not selected.
{count}i)	
{count}ia	
{count}a<	Select matching <> pair, include the <>.
{count}a>	

The Vim Tutorial and Reference

{count}i<	Select matching <> pair, excluding the <>.
{count}i>	Select matching [] pair, including the [].
{count}a[Select matching [] pair, including the [].
{count}a]	Select matching [] pair, excluding the [].
{count}i[Select matching [] pair, including the [].
{count}i]	Select matching {} pair, including the {}.
{count}a{	Select matching {} pair, including the {}.
{count}a}	Select matching {} pair, excluding the {}.
{count}i{	Select matching {} pair, excluding the {}.
{count}i}	Select enclosing "", including the "".
{count}a"	Select enclosing "", excluding the "".
{count}i"	Select enclosing ' ', including the ' '.
{count}a'	Select enclosing ' ', excluding the ' '.
{count}i'	Select enclosing ' ', excluding the ' '.
{count}a`	Select enclosing ` `, including the ` `.
{count}i`	Select enclosing ` `, excluding the ` `.
{count}at	Select the enclosing XML tag block (<foo> ... </foo>) including the tags.
{count}it	Select the enclosing XML tag block (<foo> ... </foo>) excluding the tags.

```
for (i = 0; i < 100; ++1)    for (i = 0; i < 100; ++1)
```

a(

i(

Figure 22-5: **a(** and **i(** commands.

Moving to the Other End of a Selection

The **o** command moves the cursor to the other end of a selection (see Figure 22-6). You can then move back to the other end (where you came from) with another **o**.

The Vim Tutorial and Reference

Found near the computer room of an American company:

ACHTUNG! ALLES LOOKENSPEEPERS!

o
Go to
other end

Das computermachine ist nicht fuer gefingerpoken und mittengrabben. Ist easy schnappen der springenwerk, blowenfusen und poppencorken mit spitzensporken. Ist nicht fuer gewerken bei das dumpkopfen. Das rubbernecken sichtseeren keepen das cotten-pickenen hans in das pockets muss; relaxen und watchen das blinkenlichten.

Figure 22-6: The **o** command.

The **o** command moves the cursor to the other corner of the selection in block visual mode (see Figure 22-7). In other words, the **o** command moves to the other end of the selection on the same line.

o
Other side

The last sign in pseudo-German has made its way around the world. It even made its way to Germany where they translated it into pseudo-English.

ATTENTION

This room is fullfilled mit special electronische equippmant. Fingergrabbing and pressing the cnoeppkes from the computers is allowed for die experts only! So all the "lefthanders" stay away and do not disturben the brainstorming von here working intelligencies. Otherwise you will be out thrown and kicked anderswhere! Also: please keep still and only watchen astaunished the blinkeenlights.

-- VISUAL BLOCK --

Figure 22-7: The **o** command.

Case Changes

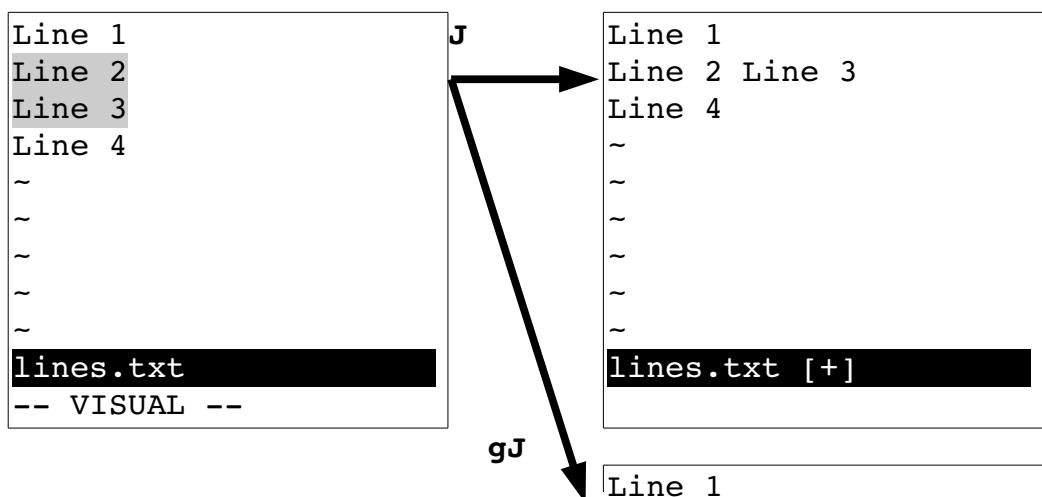
The `~` command inverts the case of the selection. The `U` command makes the text uppercase and the `u` command turns the text into lowercase. Figure 22-8 illustrates how the various case-changing commands work. The figures show initial selection, `~`, `U`, and `u`, respectively.

Line 1	Line 1	Line 1	Line 1
Line 2	LINE 2	LINE 2	line 2
Line 3	lINE 3	LINE 3	line 3
Line 4	Line 4	Line 4	Line 4
Line 5.	Line 5.	Line 5.	Line 5.
~	~	~	~
~	~	~	~
~	~	~	~
~	~	~	~
lines.txt	lines.txt	lines.txt	lines.txt
--VISUAL--		U	u
Initial Selection			

Figure 22-8: Case-changing commands.

Joining Lines

The `J` command joins all the highlighted lines into one long line. Spaces are used to separate the lines. If you want to join the lines without adding spaces, use the `gJ` command. Figure 22-9 shows how the `J` and `gJ` commands work.



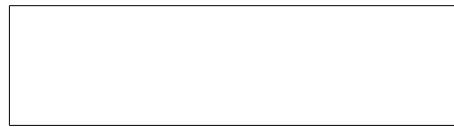


Figure 22-9: **J** and **gJ** commands.

Formatting a Block

The **gq** command formats the text (see Figure 22-10).

Select lines

```
Oh woe to Mertle the turtle  
who found web surfing quite a hurtle.  
The system you see  
was slower than he.  
And that's not saying much for the turtle.  
~  
poem.txt  
-- VISUAL --
```

gq

```
Oh woe to Mertle the turtle who found web  
surfing quite a hurtle. The system you  
see was slower than he. And that's not  
saying much for the turtle.  
~  
~  
poem.txt  
-- VISUAL --
```

Figure 22-10: The **gq** command.

The Encode (**g?**) Command

?The **g?** command encodes or decodes the highlighted text using the *rot13* encoding. (This primitive encoding scheme is frequently used to obscure potentially offensive Usenet news postings.)

With *rot13*, if you encode something twice, you decode it. Therefore if the text is encoded, **g?** decodes it. If it is in plain text, **g?** encodes it. Figure 22-11 shows how this encryption works.

The Vim Tutorial and Reference

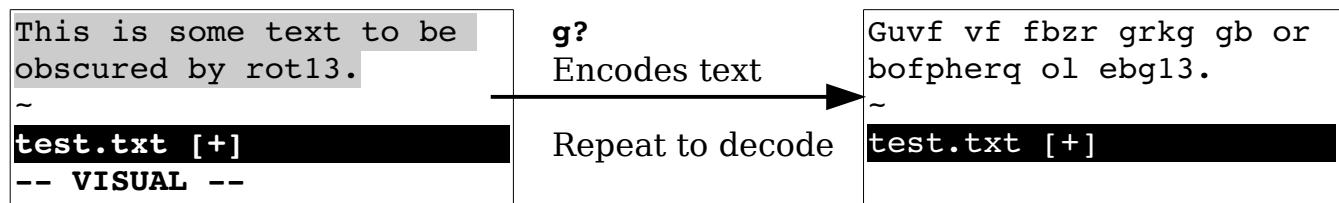


Figure 22-11: The `g?` command.

The Colon (:) Commands

The `:` command starts a command-mode command with a range already specified. If you want to write a block to a file, for example, select the text using visual mode, and then execute the following command:

```
:write block.txt
```

This writes the text to the file `block.txt`.

Note: The `:` command only works on whole lines.

Pipe (!) Command

The `!` command pipes a region of text through an external program. For example, the `!sort` pipes the selection through the UNIX `sort` program. Figure 22-12 shows the visual `!` command used to `sort` a range of lines.

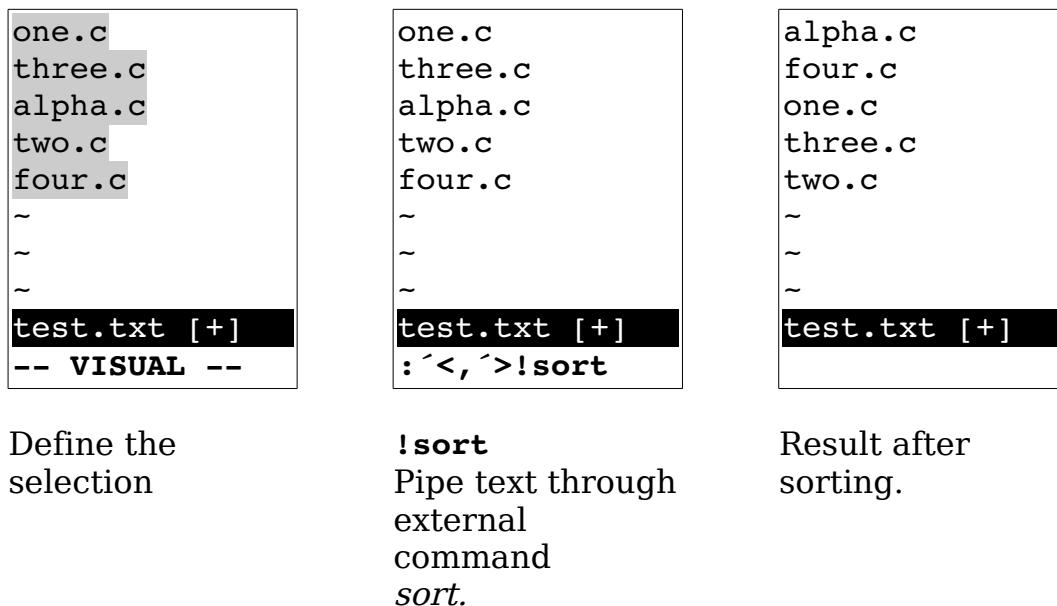


Figure 22-12: The `!` (pipe) command.

The Vim Tutorial and Reference

Note: The ! command always works on lines even if you are in character visual mode or visual block mode.

Select Mode

Select mode is yet another visual mode that allows for quick deletion or replacement of the selected text. The way you use select mode is simple. You highlight the text and then type <BS> to delete it. Or you can highlight the text, and then replace it by just typing the replacement.

How does select mode compare with visual mode? With visual mode, you highlight the text and then perform an operation. In other words, you need to end the visual mode operation with a command. With select mode, the commands are limited to <BS> (for delete) and printable characters (for replacement). This makes things faster because you do not need to enter a command, but it is much more limited than visual mode.

(**CTRL-H** is equivalent to <BS>.)

You can choose from three select-mode flavors. The commands to start the various flavors of the select mode are as follows:

- gh** Start characterwise selection.
- gH** Start linewise selection.
- gCTRL-H** Start block selection.

Moving the cursor in select mode is a little more difficult than moving it in normal visual mode because if you type any printable character, you delete the selected text and start inserting. Therefore, to select text, you must use the arrow, CTRL, and function keys.

You can also use the mouse to select text if you set the '**selectmode**' ('**s1m**') option to mouse, as follows:

```
:set selectmode=mouse
```

(Without this option, the mouse performs a visual selection rather than a select-mode selection.)

You can also use the '**selectmode**' option to let the shifted cursor keys enter select mode.

Deleting the Selection

The backspace command (<BS> or **CTRL-H**) deletes the selected text (see Figure 22-13).

The Vim Tutorial and Reference

```
Oh woe to Mertle the turtle
who found web surfing quite a hurtle.
    The system you see
        was slower than he.
And that's not saying much for the turtle.
-- SELECT --
```

gh (start select), then **<Right><Right>** to highlight “the”.

<bs> deletes text

```
Oh woe to Mertle  turtle
who found web surfing quite a hurtle.
    The system you see
        was slower than he.
And that's not saying much for the turtle.
```

Figure 22-13: Deleting text in select mode.

Replacing Text

Typing any printable character causes the selected text to be deleted and throws *Vim* into insert mode (see Figure 22-14).

The Vim Tutorial and Reference

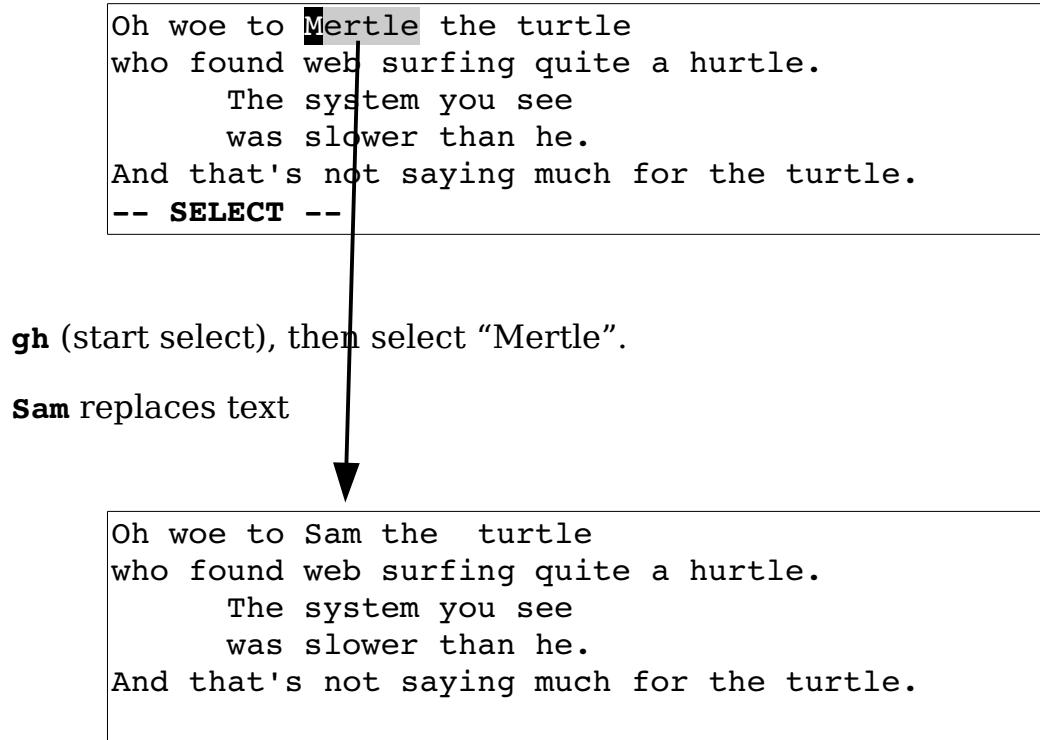


Figure 22-14: Replacing text in select mode.

Switching Modes

The **CTRL-O** command switches from selection mode to visual mode for one command. The **CTRL-G** command switches to visual mode without returning. To switch from visual mode to select mode, use the **CTRL-G** command.

Avoiding Automatic Reselection

Usually when you select text, the text remains selected. Even if you execute a command, the selection remains. The **gv** command causes the selection to disappear after the command is executed. This proves extremely useful for macros that make a selection, do something with it, and then want it to disappear.

Chapter 23: Advanced Commands for Programmers

The *Vim* editor was written by programmers who wanted a good text editor.

Because of that, *Vim* includes a lot of commands you can use to customize and enhance it to make editing programs easier.

Consider, for example, the problem of the `<Tab>` character. You can deal with this character in many different ways. You can set the tab stops to the indentation size, leave them at the default eight characters, or eliminate them altogether (force everyone to use spaces). The *Vim* editor supports all these types of editing. This chapter shows you how to use each of them.

Previously, you saw how to turn on C indent mode. This chapter describes, in detail, how to customize this mode.

You have learned how to turn syntax highlighting on as well. This chapter takes you a step further, showing you how to customize it. This chapter discusses the following:

- Removing **autoindents**
- Inserting registers and indent
- Indentation program options
- Tabbing options
- Customizing C indentation
- Comparing two files
- Using the preview window
- Matching options
- Additional motion commands for programmers
- Commands for editing files in other directories
- Advanced **:make** options
- Customizing the syntax highlighting

Removing an Automatic Indentation

Suppose you are editing a program. You have '**autoindent**' ('**ai**') set and are currently indenting in about three levels. You now want to put in a comment block. This is a big block, and you want to put it in column 1, so you need to undo all the automatic indents. One way to do this is to type **CTRL-D** a number of times.

Or you can use **OCTRL-D**. The **OCTRL-D** command in insert mode removes all the automatic indentation and puts the cursor in column 1. (Note that when you type the **O**, it appears on the screen--at this point, *Vim* thinks you are trying to insert a **O** into the text. When you type in the **CTRL-D**, it realizes you are executing a **OCTRL-D** command and the **O** disappears.)

When you use **OCTRL-D**, the cursor returns to column 1 (see Figure 23-1). The next line also starts in column 1 (normal '**autoindent**' behavior).

The diagram shows a code block in a box. The code is:

```

if (flag) {
    go to skip_it;
}
do something();
/* This is the start of comment block */
/* This is the second line of the block */

```

Two arrows point to the code from the left. The top arrow is labeled "OCTRL-D un-inds this line". The bottom arrow is labeled "This line is not automatically indented".

Figure 23-1: The **OCTRL-D** command.

Suppose, however, that you are typing in a label or an **#ifdef** directive and want to go to column 1 for one line only. In this case, you want the **^CTRL-D** command. This places you in column 1 for the current line only. When you enter the next line, the indent is automatically restored (see Figure 23-2).

The diagram shows a code block in a box. The code is:

```

if (flag) {
    go to skip_it;
}
#endif FLAG
if (flag2) {

```

Two arrows point to the code from the left. The top arrow is labeled "^CTRL-D un-inds this line". The bottom arrow is labeled "Automatically indented".

Figure 23-2: The **^CTRL-D** command.

Inserting Indent

The **CTRL-T** command is like a **<Tab>**, except that it inserts an indent the size of the '**shiftwidth**' option. If you use a '**shiftwidth**' ('**sw**') of 4, for instance, pressing **<Tab>** moves you to the next 8-column boundary (a '**tabstop**' ('**ts**')), assuming that you have the default setting of '**tabstop=8**'). But pressing **CTRL-T** moves us you to the next 4-column boundary.

The **CTRL-T** and **CTRL-D** commands work at any point on the line (not just the beginning). Therefore, you can type some text and then use **CTRL-T** and **CTRL-D** to adjust the indentation.

Inserting Registers

Generally when you use **CTRL-R** to insert the contents of a register, the contents are autoindented. If you do not want this to happen, use the command **CTRL-R CTRL-O register**. On the other hand, if you want to insert a register and have *Vim* "do the right thing," use the **CTRL-R CTRL-P** register command.

Take a look at how this works. Assume that you are editing a file that contains the following:

```
1 int main()
2 {
3     if (x)
4     {
5         y();
6     }
```

The following settings have been made:

```
:set number
:set cindent
:set shiftwidth=4
```

You start on line 3 and do a **v** to enter line visual mode. Going down to line 6, you highlight the entire if block. You dump this in register **a** with the command "**ay**". Next you add two lines at the end to start another if. Your text now looks like this:

The Vim Tutorial and Reference

```
1 int main()
2 {
3     if (x)
4     {
5         y();
6     }
7     if (z)
8     {
```

Register **a** contains the following:

```
if (x)
{
    y();
}
```

Next you go into insert mode and insert the contents of register **a** using the command **CTRL-R a**. The result is ugly:

```
if (x)
{
    y();
}
```

So what happened? Register **a** contains indented lines. But *Vim* has indenting turned on. Because you inserted the register indent and all, you wound up with double indentation. That was not what you wanted. Go back to where you were (**CTRL-Ou**, undo) and execute a **CTRL-R CTRL-O a**. The result is as follows:

```
1 int main()
2 {
3     if (x)
4     {
5         y();
6     }
7     if (z)
8     {
9         if (x)
10        {
11            y();
12        }
```

This is better. You do not have the double indents. Trouble is, you still do not have the right indent. The problem is that *Vim* kept the old indent from the original text. Because this line is under the `if (z)` statement, however, it should be indented an extra level. So you go back and try **CTRL-R CTRL-P a**.

The Vim Tutorial and Reference

The result is as follows:

```
1 int main()
2 {
3     if (x)
4     {
5         y();
6     }
7     if (z)
8     {
9         if (x)
10        {
11            y();
12        }
```

Now *Vim* correctly indented the text by recalculating the indent of each line as it was put in.

In normal mode, the "**{register}p**" command inserts the text in the specified register into the buffer. The "**{register}]p ()<MiddleMouse>**" command does the same thing, except each line has its indent adjusted. Similarly, the "**{register}]P**" command acts like the "**{register}P**" command with indent adjustment.

To Tab or Not to Tab

Back in the early days, B.C. (before computers), there existed a communication device called a Teletype. Some models of Teletype could do tabs. Unfortunately, tab stops were set at every eight spaces. When computers came along, their first consoles were Teletypes. Later, when more modern devices (such as video screens) replaced Teletypes, the old tab size of eight spaces was kept for backward compatibility.

This decision has caused programmers no end of trouble. Studies have shown that the most readable indentation size is four spaces. Tab stops are normally eight spaces. How do we reconcile these two facts? People have chosen several ways. The three main ones are:

1. Use a combination of spaces and tabs in your program to enter code. If you need an indentation of 12, for example, use a tab (8) and four spaces (4).
2. Tell the machine that tab stops are only 4 spaces and use tabs everywhere. (This is one solution I personally frown upon, because I do not use the special setting and the text appears to be over-indented.)

The Vim Tutorial and Reference

3. Throw up your hands and say that tabs are the work of the devil and always use spaces.

The *Vim* editor, thank goodness, supports all three methods.

Spaces and Tabs

If you are using a combination of tabs and spaces, you just edit normally. The *Vim* defaults do a fine job of handling things. But you can make life a little easier by setting the '**softtabstop**' ('**sts**') option. This option tells *Vim* to make the Tab key look and feel as if tabs were set at the value of '**softtabstop**', but use a combination of tabs and spaces to fake things (see Figure 23-3). After you execute the following command, every time you press the Tab key the cursor moves to the next 4-column boundary:

```
:set softtabstop=4
```

The first time you press it, however, you get 4 spaces inserted in your text. The second time, *Vim* takes out the 4 spaces and puts in a tab (thus taking you to column 8).

What Was Typed

```
<Tab>x  
<Tab><Tab>x  
<Tab><Tab><Tab>x  
<Tab>x<Tab><Tab>x  
<Tab>x<Tab><Tab><Tab>x
```

Dots represent spaces
Arrows represent tabs

Result

```
11111111112222222223  
123456789012345678901234567890  
....x  
→ x  
→ ....x  
....x → ....x  
....x → x
```

Figure 23-3: Soft tabs.

Smart Tabs

Another related option is the '**smarttab**' ('**sta**') option. With this option on (`:set smarttab`), tabs inserted at the beginning of a line are treated like soft tabs. The tab size used in this case is defined by the '**shiftwidth**' option.

The Vim Tutorial and Reference

But tabs inserted elsewhere in the text act just like normal tabs. Note that you must have soft tabs off (`:set softtabstop=0`) for this option to work. Figure 23-4 shows sample results.

Smart indenting is a combination of soft tabs and normal tabs. When you execute the following command, *Vim* treats tabs at the beginning of a line differently:

```
:set smarttab
```

Suppose, for example, that you have the following settings:

```
:set shiftwidth=4  
:set tabstop=8  
:set smarttab
```

Tab stops are every eight spaces and the indentation size is four spaces. When you type `<Tab>` at the beginning of a line, the cursor will move over the indentation size (four spaces). Doing a double `<Tab>` moves over two indentation sizes (eight spaces [4*2]).

What Was Typed

```
<Tab>x  
<Tab><Tab>x  
<Tab><Tab><Tab>x  
<Tab>x<Tab><Tab>x  
<Tab>x<Tab><Tab><Tab>x
```

Dots represent spaces
Arrows represent tabs

Result

```
11111111112222222223  
123456789012345678901234567890  
....x  
----->x  
-----> ----->x  
....x -----> ----->x  
....x -----> -----> ----->x
```

Figure 23-4: Smart tabs.

The following table shows you what happens if you type certain things at the beginning of the line.

What's types	What's Inserted
<code><Tab></code>	Four spaces
<code><Tab><Tab></code>	One tab
<code><Tab><Tab><Tab></code>	One tab, four spaces
<code><Tab><Tab><Tab><Tab></code>	Two tabs

The Vim Tutorial and Reference

When you type **<Tab>** anywhere else in the line, however, it acts like a normal tab.

Using a Different Tab Stop

The following command changes the size of the tab stop to 4:

```
:set tabstop=4
```

('ts' is the short form of '**tabstop**'.)

You can actually change it to be any value you want. Figure 23-5 shows what happens when '**tabstop**' is set to 4.

What Was Typed

```
<Tab>x  
<Tab><Tab>x  
<Tab><Tab><Tab>x  
<Tab>x<Tab><Tab>x  
<Tab>x<Tab><Tab><Tab>x
```

Result

1111111112222222223
123456789012345678901234567890
→x
→ → → x
→ → → → x
→x → → → x
→x → → → → x

Dots represent spaces
Arrows represent tabs

Figure 23-5: *tabstop* set at 4.

Note: Just because you change it in *Vim* does not mean that it will change in your terminal window, that your printing program will not still use eight-character tab stops, or that other editors will use the new setting. Therefore, your type and print commands might require special options to handle things.

No Tabs

If you want absolutely no tabs in your file, you can set the '**expandtab**' ('**et**') option. When this option is set, the **<Tab>** key inserts a series of spaces. (Note that setting '**expandtab**' does not affect any existing tabs. In other words, any tabs in the document remain tabs. If you want to convert tabs to spaces, use the **:retab** (**:ret**) command, which is explained later.)

The Vim Tutorial and Reference

Note: If you really want to insert a tab when this option is on, type **CTRL-V<Tab>**. The **CTRL-V** command tells *Vim* that you really want to insert this **<Tab>** as a tab and not a bunch of spaces.

The 'copyindent' and 'preserveindent' Options

If the '**preserveindent**' ('**pi**') is set then then *Vim* will attempt keep whatever indent structure you already have. (As much as possible.) For example, if you 16 wide indent consists of 8 spaces and one **<tab>**, *Vim* will keep what you've got. Without this option it will use two tabs.

If the '**copyindent**' ('**ci**') option is set, when a new line is opened, *Vim* will attempt to copy the indentation from the previous line.

The :retab Command

The **:retab** command transforms text with tab stops at one setting to tab stops with another. You can use it to turn tabs into a series of spaces as well, or a series of spaces into tabs.

For example, suppose that you have a file that was created with tab stops of 4 (**:set tabstop=4**). This is a non-standard setting, and you want to change things so that the tab stops are 8 spaces. (You want the text to look the same, just with different tab stops.) To change the tap stop in the file from 4 to 8, first execute the command

```
:set tabstop=4
```

The text should appear on the screen correctly. Now execute the command

```
:%retab 8
```

This changes the tab stops to 8. The text will appear unmodified, because *Vim* has changed the white space to match the new value of '**tabstop**'.

For another example, suppose that you are required to produce files with no tabs in them. First, you set the '**expandtab**' option. This causes the **<Tab>** key to insert spaces on any new text you type. But the old text still has tabs in it. To replace these tabs with spaces, execute the command

```
:%retab
```

Because you didn't specify a new tabstop, the current value of '**tabstop**' is used. But because the option '**expandtab**' is set, all tabs will be replaced with spaces.

Modelines

One of the problems with all these tabbing options is that people use them. Therefore, if you work with files created by three different people, you can easily have to work with many different tab settings. One solution to this problem is to put a comment at the beginning or end of the file telling the reader what tab stops to use.

For example:

```
/* vim:tabstop=8:expandtab:shiftwidth=8 */
```

When you see this line, you can establish the appropriate *Vim* settings, if you want to. But *Vim* is a smart editor. It knows about comments like this and will configure the settings for you. A few restrictions apply. The comment must be formatted in this manner and it must appear in the first or last five lines of the program (unless you change the setting of '**modelines**').

This type of comment is called a *modeline*.

Shift Details

Suppose that you are typing with a shift width of 4 and you enter a line with 3 spaces in front of it. What should the **>>** command do? Should it add 4 spaces in front of the line or move it to the nearest shift width. The answer depends on the value of the '**shiftround**' ('**sr**') option.

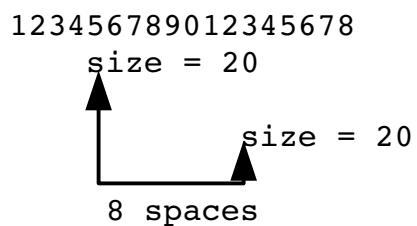
Usually this option is not set, so **>>** puts in 4 spaces. If you execute the following command, **>>** moves the indent to the next shift-width boundary:

```
:set shiftround
```

Figure 23-6 shows how this works.

The Vim Tutorial and Reference

With "noshiftround">>>moves the text over 4 spaces



With "shiftround">>>moves the text to the next "shiftwidth" boundary. (In this case, column 8.)

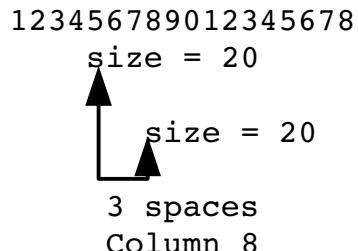


Figure 23-6: The '**shiftround**' option.

Specifying a Formatting Program

You can define the program *Vim* uses when executing the = command, by setting the '**equalprg**' option. If this option is not set (and you are not editing a lisp program), the *Vim* editor uses its own built-in indentation program that indents C or C++ programs. If you want to use the GNU indent program (available from www.gnu.org), for instance, execute this command:

```
:set equalprg=/usr/local/bin/indent
```

Formatting Comments

One of the great things about *Vim* is that it understands comments. You can ask *Vim* to format a comment and it will do the right thing.

Suppose, for example, that you have the following comment:

```
/*
 * This is a test.
 * Of the text formatting.
 */
```

You then ask *Vim* to format it using the following commands:

1. Position the cursor to the start of the comment.
2. Press **v** to start visual mode.

3. Go to the end of the comment with the % command.
4. Format the visual block with the command **gq**. The result is:

```
/*
 * This is a test. Of the text formatting.
 */
```

Note that *Vim* properly handled the beginning of each line.

(For deciding what is and is not a comment, *Vim* uses the '**comments**' ('**com**') option, described in the following section.)

The **gq{motion}** command accomplishes the same thing.

Defining a Comment

The '**comments**' option defines what is a comment. This option is a series of *flag:string* pairs.

The possible flags are as follows:

- b** Blank must follow. This means the character begins a comment only if followed by a blank or other whitespace.
- f** Only the first line has the comment string. Do not repeat the string on the next line, but preserve indentation.
- l** When used on part of a three-piece comment, make sure that the middle lines up with the beginning or end. This must be used with either the s or e flag.
- n** Indicates a nested comment.
- r** Same as l, only right-justify.
- x** Tells *Vim* that a three-part comment can be ended by typing just the last character under the following circumstances:
 1. You have already typed in the beginning of the comment.
 2. The comment has a middle.
 3. The first character of the end string is the first character on the line.

For three-part comments, the following flags apply:

- s** Start of three-piece comment.
- m** Middle of a three-piece comment.
- e** End of a three-piece comment.
- number** Add the number of spaces (can be negative) to the indentation of a middle part of a three-part comment.

The Vim Tutorial and Reference

A C comment starts with `/*`, has a middle of `*`, and ends with `*/`, as follows:

```
/*
 * This is a comment
 */
```

This results in the '**comments**' option specification of

```
set comments=s1:/*,mb:*,ex:*/
```

The **s1** indicates that this is the start of a three-part comment (**s**) and the other lines in the command need to be indented an extra space (**1**). The comment starts with the string `/*`.

The middle of the comment is defined by the **mb:*** part. The **m** indicates a middle piece, and the **b** says that a blank must follow anything that is inserted. The text that begins the comment is `*`.

The ending is specified by **ex:*/**. The **e** indicates the end, and the **x** indicates that you have only to type the last character of the ending to finish the comment. The end delimiter is `*/`.

Take a look at how this definition works. First, you need to set the following option :

```
:set formatoptions=qro
```

375('fo' is the short version of '**formatoptions**'.)

The following options prove useful for formatting text (see *Chapter 11: Dealing with Text Files* for complete details):

- q** Allow formatting of comments using **qq**.
- r** Automatically insert the middle of a comment after pressing **<Enter>**.
- o** Automatically insert the middle of a comment when a line inside a comment is opened with an **o** or **O** command.

Now start typing in comments. You start with a line containing the comment header, `/*`, as follows:

```
/*
```

When you type **<Enter>**, because **r** is in the format options, you get the following:

The Vim Tutorial and Reference

```
/*  
 *
```

The *Vim* editor automatically inserted the * surrounded by a space on each side to make the comment look good. Now enter a comment and a new line:

```
/*  
 * This is an example  
 *
```

Now you need to end the comment. But *Vim* has already typed a space after the asterisk. How can you enter */? The answer is that *Vim* is smart and will end the comment properly if you just type /. The cursor moves back, the slash is inserted, and you get the following:

```
/*  
 * This is an example  
 */
```

You can use a number of different formatting commands to format text or comments. For more information on these, see *Chapter 11: Dealing with Text Files* and *Chapter 20: Advanced Text Blocks and Multiple Files*.

Customizing the C Indentation

The C indentation process is controlled by the following options:

cinkeys	Defines the keys that trigger an indent event
cink	
cinoptions	Defines how much to indent
cino	
cinwords	Defines the C and C++ keywords
cinw	

The '**cinkeys**' option defines which keys cause a change to indentation. The option is actually a set of type-char key-char pairs. The type-chars are as follows:

- ! The following key is not inserted. This proves useful when you want to define a key that just causes the line to be re-indented. By default, **CTRL-F** is defined to effect re-indentation.
- * The line will be re-indented before the key is inserted.
- 0 The key causes an indentation change only if it is the first character typed on the line. (This does not mean that it is the first character on the line, because the line can be autoindented. It specifies the first typed character only.)

The key-chars are as follows:

The Vim Tutorial and Reference

<name>	The named key. See <i>Appendix B: The <> Key Names</i> for a list of names.
^x	Control character (that is, CTRL-x).
o	Tells <i>Vim</i> to indent the line when you use an o command to open a new line.
O	Line o, but for the o command.
e	Re-indent the line when you type the final e in else.
:	Re-indent the line when you type a colon after a label or case statement.
<^>, <>, <>>, <o>, <e>, <O>	The literal character inside the angle brackets.

The default value for this the '**cinkeys**' option is as follows:

```
:set cinkeys=0{,0},:,0#,!^F,o,O,e
```

Figure 23-7 shows how the '**cinkeys**' option works.

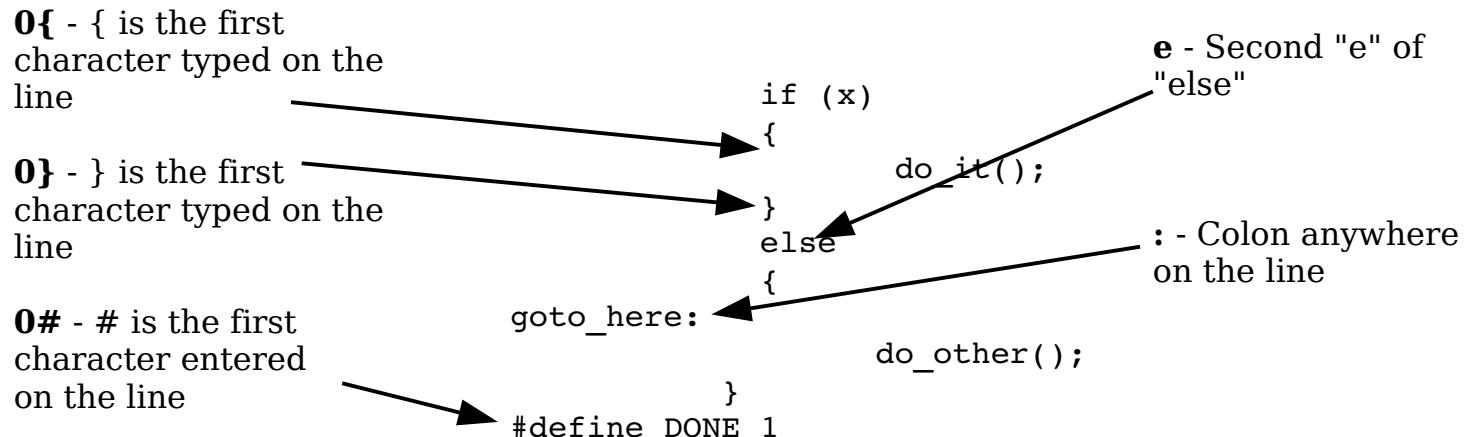


Figure 23-7: The '**cinkeys**' option.

The '**cinoptions**' Options

The '**cinoptions**' option controls how much *Vim* indents each line. This option consists of a series of key indent pairs. The key is a single letter that controls what part of the program is affected (see the following table and Error: Reference source not found and Error: Reference source not found). The indent tells the program how much indentation to use. This can be a number of spaces (for example, 8), or a negative number of spaces (-8). It can also be a multiple of the '**shiftwidth**' option that is specified as s. For example, **1s** is a shift width, **0.5s** is half a shift width, and **-1s** un-indents a shift width.

Key	Default	Description
>	s	Normal shift used for indents not covered by another letter.
e	0	Extra indent added to lines following a line with a curly brace that ends a line (or more technically, one that does not begin a line).
n	0	Extra indent added to single lines not inside curly braces after an if, while, and so on.
f	0	Extra indent added to a function body. Includes the outermost {} that define the function.
{	0	Spaces to be added to opening {.
}	0	Spaces to be added to closing }.
^	0	Spaces to be added to text inside a set of {} that start in column 1. For example, a value of -1s causes the body of a function to be pushed one shift width to the left.
:	s	Amount to indent a case inside a switch statement.
=	s	Extra indent for statements after the case statement.
l	0	If set to something other than zero, then align the code for a case with the end of the case.
b	0	Make the break statement match the preceding case inside a switch . (Please don't do this.)
g	s	Indentation for C++ protection keywords (public, private, protected).
h	s	Indentation for statements that follow a protection keyword.
p	s	Shift for K&R-style parameters.
t	s	Indent the type declaration for a function if it is on a separate line.
i	s	Indent for the initializers of a C++ class.
+	s	Indent for continuation lines (the 2-n lines of a statement).
c	3	Indent the middle of a multiline comment (if no middle * is present).
C	0	When non-zero indent lines as specified by the c option even if there is text in the comment which would cause a different indentation.
/	0	Indent comments lines this amount extra.
(2s	Indent for a line in the middle of an expression. Actually, the indent for a line that breaks in the middle of a set of () .
u	s	Indent for a line that breaks in the middle of a nested set of () (like (, but one level deeper).
U	0	When non-zero do not ignore u or (inside a set of () .

The Vim Tutorial and Reference

Key	Default	Description
w	0	If non-zero and using (0 or If non-zero and using u0 or If non-zero and using u0 and the unmatched (is the first non-blank character on the line then Align the line with the first non-blank character after the (. Basically, don't try to understand this, do some experimentation to see what looks nice.
w	0	Inside an unmatched (with (0 or u0 , the amount to indent.
m	0	If non-zero then align up the line with the ending) with the starting (.
M	0	When set a closing) aligns with the previous line.
j	0	Indent Java anonymous classes correctly.
)	20	Specify the number of lines to search for closing () .
*	30	Specify the number of lines to search for unclosed comment.
#	0	If non-zero recognize Perl comments.

The following examples show what happens with various indent options.
These examples assume a '**shiftwidth**' of 4. (Note dot represents space.)

cindentopt=""	cindentopt=">2"	cindentopt=">s2"
if (flag) {do_it(); }	if (flag) { ..do_it(); }	if (flag) {do_it(); }
cindentopt=""	cindentopt="e2"	cindentopt="es2"
if (flag) {do_it(); } else {do_other(); }	if (flag) { ..do_it(); } else {do_other(); }	if (flag) {do_it(); } else {do_other(); }
cindentopt=""	cindentopt="n2"	cindentopt="ns2"
if (flag)do_it();	if (flag) ..do_it();	if (flag)do_it();
cindentopt=""	cindentopt="f2"	cindentopt="fs2"

The Vim Tutorial and Reference

```
void do_it()  
{
```

```
cindentopt=""  
if (flag)  
{
```

```
cindentopt=""  
int foo()  
{  
....do_it();  
}
```

```
cindentopt=":0"  
switch (x)  
{  
....case 'x':  
.....break;  
}
```

```
cindentopt=""  
case 1:  
....flag = true;
```

```
cindentopt=""  
switch (x) {  
....case a:  
.....foo();  
.....break;  
    case b: {  
        foo(); .....foo();  
        break; .....break;  
    }  
}
```

```
cindentopt=""  
switch (x) {  
....case a:  
.....foo();  
.....break;  
}
```

```
cindentopt=""
```

```
void do_it()  
..{
```

```
cindentopt="{2"  
if (flag)  
..{
```

```
cindentopt="^2"  
int foo()  
{  
....do_it();  
}
```

```
cindentopt=":2"  
switch (x)  
{  
....case 'x':  
.....break;  
}
```

```
cindentopt="=2"  
case 1:  
..flag = true;
```

```
cindentopt="11"  
switch (x) {  
....case a:  
.....foo();  
.....break;  
    ....case b: {  
        .....}  
    }  
}
```

```
cindentopt="b1"  
switch (x) {  
....case a:  
.....foo();  
....break;  
}
```

```
cindentopt="g2"
```

```
void do_it()  
.....{
```

```
cindentopt="{s2"  
if (flag)  
.....{
```

```
cindentopt="^s2"  
if (flag)  
{  
.....do_it();  
}
```

```
cindentopt=":s2"  
switch (x)  
{  
.....case 'x':  
.....break;  
}
```

```
cindentopt=="s2"  
case 1:  
.....flag = true;
```

```
cindentopt="gs2"
```

The Vim Tutorial and Reference

class foo {public:int i; }	class foo { ..public:int i; }	class foo {public:int i; }
cindentopt="" class foo {public:int i; }	cindentopt="h2" class foo {public:int i; }	cindentopt="hs2" class foo {public:int i; }
cindentopt="" int foo(a,b)int a;int b;	cindentopt="p2" int foo(a,b) ..int a; ..int b;	cindentopt="ps2" int foo(a,b)int a;int b;
cindentopt=""int foo(a,b)	cindentopt="t2" ..int foo(a,b)	cindentopt="ts2"int foo(a,b)
cindentopt="" class foo:public class b	cindentopt="i2" class foo: ..public class b	cindentopt="is2" class foo:public class b
cindentopt="" a = b + c +d;	cindentopt="+2" a = b + c + ..d;	cindentopt="+s2" a = b + c +d;
cindentopt="" /* ...test */	cindentopt="c2" /* ..test */	cindentopt="cs2" /*test */
cindentopt="" ***** ..test *****/	cindentopt="C1,c2" ***** ..test *****/	cindentopt="C1,cs2" *****test *****/
cindentopt="" do_it(); /* Did it */	cindentopt="/2" do_it(); ../* Did it */	cindentopt="/s2" do_it();/* Did it */
cindentopt=""	cindentopt="(2"	cindentopt="(s2"

The Vim Tutorial and Reference

a = (b +c) c = ((c + d) * (.....e + f))	a = (b + ..c) c = ((c + d) * (...e + f) ..)	a = (b +c) c = ((c + d) * (.....e + f))
cindentopt="" a = (b+ (c +d + e))	cindentopt="u2" a = (b+ (c +d + e))	cindentopt="us2" a = (b+ (c +d + e))
cindentopt="" a = (b +(c +d + e))	cindentopt="U1,(2" a = (b + ..(c +d + e ..))	cindentopt="U1,(2s" a = (b +(c +d + e))
cindentopt="" a = (b +(c +d + e))	cindentopt="w1,(2" a = (b + ..(c + ...d + e ..))	cindentopt="w1,(2s" a = (b +(c +d + e))
cindentopt="(0" func(.....arg1,arg2); another_func(arg1,arg2);	cindentopt="W2,(0" func(..arg1, ..arg2); another_func(arg1,arg2);	cindentopt="W2s,(0" func(.....arg1,arg2); another_func(arg1,arg2);
cindentopt="" a = (b +) * (d +) * e +(f + (g *h) + i)	cindentopt="m1" a = (b +) * (d +) * e + (f + (g *h) + i)	
cindentopt=""	cindentopt="M1"	

The Vim Tutorial and Reference

```
if ( a || (           if ( a || (
.....b && c       .....b && c
.....) ||          .....)
.....d             .....d
...)               ....))
```



```
cindentopt=""
addListener(new EventListener() {
.....public void eventHandler(Event e) {
.....process(e);
....}
....});
cindentopt="j1"
addListener(new EventListener() {
....public void eventHandler(Event e) {
.....process(e);
....}
});
```

The 'cinwords' Option

The '**cinwords**' option defines what words cause the next C statement to be indented one level in the Smartindent and cindent mode. The default value of this option is as follows:

```
:set cinwords=if,else,while,do,for,switch
```

Advanced Diff Mode

Vim's diff mode is simple to use, yet very powerful. You can also do a great deal of customization with the editor. There are a number of ways you can start diff mode besides using the *gvimdiff* command from the command line.

The first is **:diffsplit {filename}** (**:diffs**). This splits the window and does a diff between the current file and new file you just specified. By default this splits the window horizontally so you may want to execute the following command to split things vertically:

```
:vertical diffsplit {file-name}
```

The Vim Tutorial and Reference

The other way is to execute the command **:diffthis (:diff)**. This makes the current file part of the set of files whose differences are being displayed. Naturally you have to execute this command for more than one file for the diffs to appear.

Next we have the **:diffpatch (:diffp)** command:

```
:vertical diffpatch {patch-file}
```

This runs the *patch*⁵ program on the selected file, and opens a new window with the results. Thus you can see what applying a patch to a file is going to do to you.

Vim knows how to run the standard GNU patch program, but if you have a different one you can set the '**patchexpr**' ('**pex**') to be anything you want. When that **:diffpatch** command is run, *Vim* will evaluate this option to perform the patch.

Vim attempts to be very good about making sure that the difference display is keep up to date even when you change a file. However, it's not perfect. If things get a little confused, you can always tell *Vim* to redo the difference highlighting with the **:diffupdate (:dif)** command.

Finally to take a window out of the difference set, use the **:diffoff** command. The command **:diffoff!** performs this operation for all windows in the current tab page.

Moving from difference to difference

The **]c** command jumps forward to the next change. A [count] can be given to jump forward multiple changes. The **[c** command does the same thing only backwards.

Moving Differences Around

As we've already discussed **do** obtains a difference from the other window, and **dp** puts the current difference to the other window. The **:diffget (:diffg)** and **:diffput (:diffpu)** commands do the same thing only they give you a little more control

First of all they take a range argument. All differences within that range will be moved. For example, to take all the differences in the first 100 lines from the current file and put them in the other file use the command:

⁵ The *patch* program is a standard Linux and UNIX tool. A version for Microsoft Windows is found in the Cygwin package.

```
:1,100 diffput
```

The other advantage of `:diffget` and `:diffput` is that they take an argument which specifies which buffer is to be considered the “other” file. This can be a buffer number, or enough of a file name so that Vim can identify the buffer.

For example, suppose we are doing a multi-way diff between the file *main.c*, *main.c.v1*, *main.c.v2*, *main.c.v3*. We are currently editing *main.c* and want to grab the diff from *main.c.v2*. For that we use the command:

```
:diffget v2
```

Customizing Diff

The '`diff`' option tells Vim whether or not this file is part of the diff set. If this option is set, the the file is part of the set, if not set (`nodiff`) then it is not.

The option '`diffexpr`' ('`dex`') contains the expression that's evaluated to perform the diff. The variables `v:fname_in`, `v:fname_new`, `v:fname_out` and `should diff v:fname_in and v:fname_new and store the results in v:fname_out`.

The '`diffopt`' ('`dip`') option let's customize difference mode. The values for this option include:

Option	Meaning
<code>filler</code>	Display filler lines when lines are added or deleted to keep things aligned.
<code>context:{number}</code>	Display <code>{number}</code> lines of context around a difference.
<code>icase</code>	Ignore case differences
<code>iwhite</code>	Ignore whitespace differences
<code>horizontal</code>	By default split the windows horizontally in diff mode.
<code>vertical</code>	By default split the windows vertically in diff mode.
<code>foldcolumn:{column}</code>	Set the number of columns to use for the folding indicator.

Comparing Two Files The Old Fashioned Way

Suppose you want to compare two files that differ by a just a few edits and for some reason diff mode does not appeal to you. You can do this, start by opening two windows, one for each edit. Next, execute the following command in each window:

```
:set scrollbind
```

('scb' is short for '`scrollbind`').)

The Vim Tutorial and Reference

Now when one window scrolls, so does the other. Figure 23-8 demonstrates how this command works. (Go down to line 14 in the first window; the second window scrolls.) As you scroll through both windows, you might encounter a place where you must move one window without moving the other. To do so, all you have to do is execute the following command in the window you want to move:

```
:set noscrollbind
```

After moving the text, then synchronize scrolling, by executing:

```
:set scrollbind
```

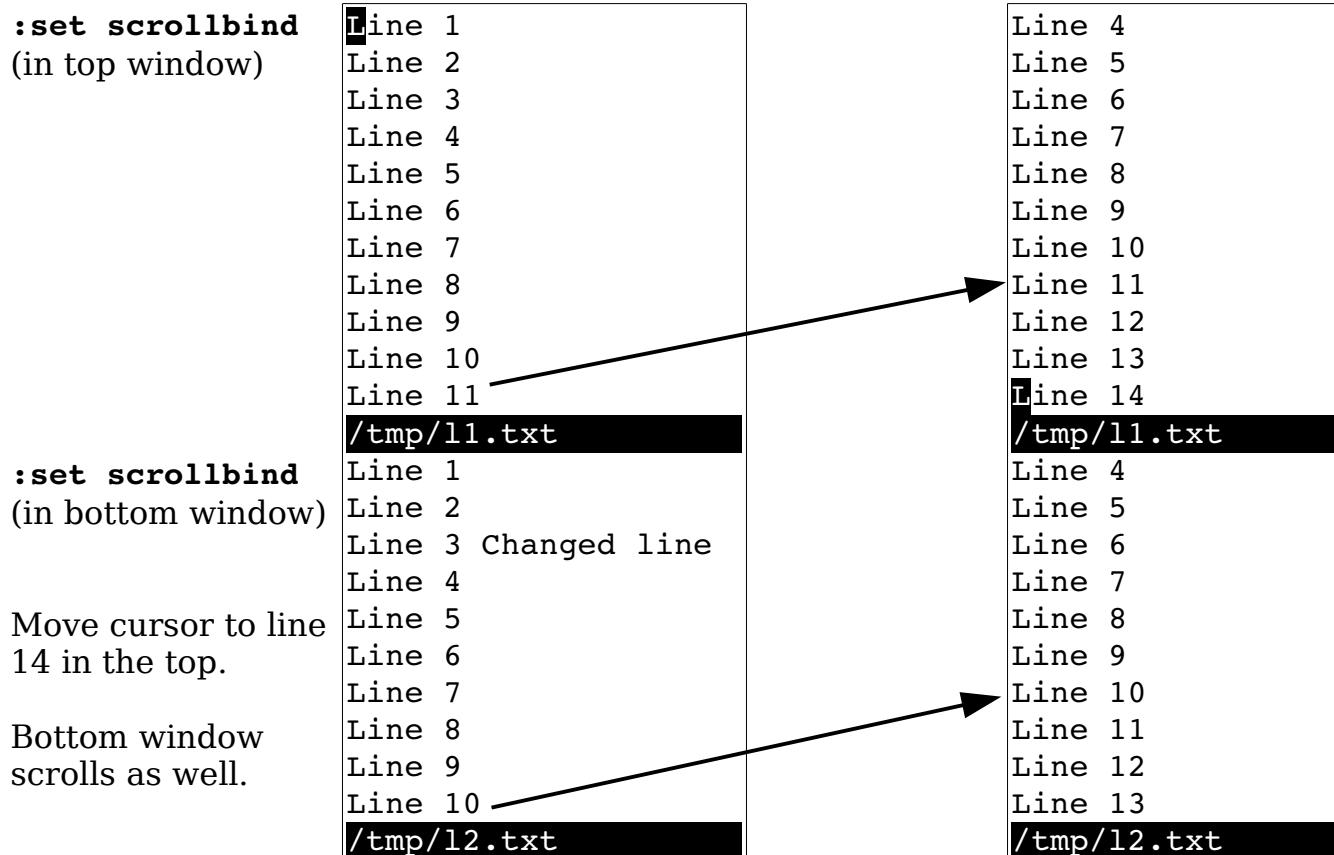


Figure 23-8: '**scrollbind**'.

The '**scrollopt**' ('**sbo**') option controls how '**scrollbind**' works. It is a set of the following keywords:

ver Vertical scrolling
hor Horizontal scrolling

The Vim Tutorial and Reference

jump When switching between windows, make sure the offsets are 0.

Finally, the following command synchronizes the two windows:

```
:syncbind
```

(**:sync** for short.)

Suppose, for example, that you have been looking at two versions of a file, moving both around. To look at some things, you turned off '**scrollbind**'. Now the files point at two different places. You want to go back to synchronized scrolling.

You could synchronize the windows by going to each and moving it to the right location; you let *Vim* do the work. In this case, you set '**scrollbind**' in both windows and then execute the following:

```
:syncbind
```

Vim then synchronizes the two files.

Advanced Folding

There are actually a number of different ways that you can cause folding to happen in your text. We've already discussed the manual mode where you manually decide to open and close folds. In *Chapter 7: Commands for Programmers* we also discussed the indent method where the indent level controls what's folded.

You can also put special markers in your text that tell *Vim* where to start and stop a fold. To make this work you need to set '**foldmethod**' ('**fdm**') to **marker**.

The '**foldmarker**' ('**fmr**') option consists of two strings, the first starts a fold, the second ends it. By default the value of this option is {{{{,}}}}. Let's take a look at a typical text

```
{{{  
This will be folded  
}}}  
This is normal  
{{{  
Another fold (to be hidden  
}}}
```

Vim will display this text as

The Vim Tutorial and Reference

```
+-- 3 lines: -----
This is normal
+-- 3 lines: -----
```

You can put a number after the marker to indicate the fold level. For example:

```
{{{1
Folded at level 1
{{{2
Folded at level 2
}}}2
Folded at level 1
}}}1
Not folded.
```

Note: An end marker ends all folds at that level and higher. So if you forget the `}}}}2`, the `}}}}1` will close the fold.

When in marker mode, the visual command `zf` command not only creates the fold, it adds the marker to the end of line. In Figure 23-9 we first highlight three lines in visual mode then fold them with `zf`. (Power users can use `zf{motion}` to do the same thing.)

```
A normal line
Fold me
Fold me
Fold me
Normal again
```

zf



```
A normal line
+-- 3 lines: Fold me-----
A normal line
```

`zo` to open the fold

```
A normal line
Fold me{{{
Fold me
Fold me}}}
Normal again
```

Figure 23-9: Creating a fold with `zf`

The Vim Tutorial and Reference

The **commentstring**' ('**cms**') option can be used to tell *Vim* to put the markers inside a comment. This option is a string with a %s in side which tells Vim where the comment should go. For example:

```
:set commentstring=\ Comment:%s
```

We can see the results of a **zf** with the '**commentstring**' set in Figure 23-10.

```
A normal line
Fold me Comment:{ {{
Fold me
Fold me Comment:}}}
Normal again
```

Figure 23-10: **zf** with '**commentstring**' set.

Similarly the **zd** command will delete the markers.

Additional fold commands

The **[count]zf** command will create a fold for **[count]** lines.

The **:fold (:fo)** command will fold a range of lines. For example to fold the first 10 lines of a file use the command:

```
:1,10 fold
```

The **zd** command deletes a fold, but will not delete nested folds. The **zD** command deletes all folds recursively for the fold under the cursor.

If you want to see all the text that in the window, the **zE** command will open all the folds visible on the screen when you execute this command.

The **zo** command opens a single fold. The **zo** command opens all the folds under the cursor.

A fold can also be opened with the **:foldopen (:foldo)** command. All folds within the line range specified are opened one level. If the override (!) option is used, all levels are opened.

The **zc** command closes one level of fold around where the cursor is located. The **zc** command closes them all.

A fold can also be closed with the **:foldclose (:foldc)** command. All folds within the line range specified are closed one level. If the override (!) option is used, all levels are closed.

Toggling folds

So **zo** opens a fold and **zc** closes. The **za** command will open a fold if it's closed and close it if it's open. This works on one level of folding. To toggle all the levels at once, use the **zA** command.

Enabling and disabling folding

The '**foldenable**' ('**fen**') option enables folding. If it is turned off ('**nofoldenable**') no folding can be done. The **zn** command turns this option off, **zN** turns it on, and **zi** (fold invert) toggles it.

Moving around folds

You can use the normal movement commands to go up and down the screen. If you want to jump from fold to fold, use the **[z** command to move to the start of the current fold (obviously the fold must be open). The **]z** command moves to the end of the fold. The **zj** command moves down to the start of the next fold and **zk** moves up to the end of the previous one.

Executing a command for all folds

The **:folddoopen** (**:folddd**) [fold do open] command executes a single command for every open fold. For example, to comment out all sections of code except the stuff in a closed fold execute the command:

```
:% folddoopen s/^/\//\//
```

Note that in this command we use a range (%) to tell the command to work on the entire file. If we wanted to just affect the lines in a closed fold we would use the command **:folddoclosed** (**:foldddoc**).

Customizing folds

The '**foldtext**' ('**fdt**') option controls how the text for a fold is displayed. The following variables are set during the execution of this function:

v:foldstart First line folded (this fold)

v:foldend Last line folded (this fold)

v:folddashes String to be put at the beginning of each fold (you can set this)

The Vim Tutorial and Reference

v:foldlevel The folding level

The '**foldcolumn**' ('**fdc**') option tells *Vim* what column to display the fold information in. Its value must be between 0 and 12.

The '**foldminlines**' ('**fml**') option controls the minimum number of lines that can be folded. If a fold is smaller than '**foldminlines**' it will not be displayed.

The '**foldnestmax**' ('**fdn**') option controls the maximum nesting level of the folds.

Controlling what opens and closes folds

The '**foldopen**' ('**fd0**') option is a set of keywords that define when a command will open a fold. It can be any set of the following keywords:

Keyword	Meaning
all	Any command
block	Any block movement command such as (, { , [, or [{ .
hor	Any horizontal movements such as l , w , or fx .
insert	Any command that starts an insert.
jump	Any command that performs a major jump such as G or gg .
mark	Jumping to a mark.
percent	The % command
quickfix	Any quick fix command such as :cc or :cn .
search	Any search such as / , ? , or n .
tag	Jumping to a tag with a CTRL-] or :tag command.
undo	The undo or redo command.

The '**foldclose**' ('**fcl**') option tells *Vim* when to automatically close a fold. It can be set to empty in which case any folds that are automatically opened are never closed, or to any in which case, all folds where are opened automatically are closed when the cursor leaves the fold.

Fold Methods

There are many different ways folds can be created. This is controlled by the '**foldmethod**' ('**fdm**') options.

The methods are:

The Vim Tutorial and Reference

diff	Folds are created as part of diff operation. (See the section <i>Diff Mode</i> in <i>Chapter 8: Basic Abbreviations, Keyboard Mapping, and Initialization Files.</i>)
expr	Folds are controlled by an expression.
indent	Folds are controlled by indentation level.
manual	Folds are created manually.
marker	Markers in the text tell <i>Vim</i> where to fold things.
syntax	The syntax highlighting rules tell <i>Vim</i> where to make the folds.

When the '**foldmethod**' is set to **syntax**, then any syntax elements which have the fold option on them are folded. For example in the C language to fold all the stuff between `#if 0` and `#endif`, *Vim* uses the following syntax rule:

```
:syn region cCppOut
\      start="^\\s*\\(%:\\|#\\)\\s*if\\s\\+0\\+\\>"
\\ end=".\\@=\\|$" contains=cCppOut2 fold
```

When '**foldmethod**' is set to **expr** then the expression in the '**foldexpr**' ('**fde**') is evaluated to get the fold level. Creating such a function is a challenging bit of Vim programming and is beyond the scope of this book.

The Preview Window

Suppose you are going through a program and find a function call that you do not understand. You could do a **CTRL-J** on the identifier and jump to the location represented by the tag. But there is a problem with this. The current file disappears because the file with the function definition replaces it.

A solution to this is to use a special window called the "preview" window. By executing the following command, you open a preview window and display the function definition:

```
:ptag function
```

(**:pt** is the short form of this command.)

(If you already have a preview window up, it is switched to the definition of the function.) Figure 23-11 shows a typical example. Assume that you have just executed the following command:

The Vim Tutorial and Reference

```
:ptag copy_p_data
```

After you have finished with the preview window, execute the following command:

```
:pclose
```

(**:pc**, **CTRL-W** **CTRL-Z**, **CTRL-Wz** accomplish the same thing.)

```
        assert((*the_data->n_data_ptr) <1000);
        return (result);
}
/* Create data from a db data record */
void copy_p_data(
    struct p_data *the_data,
    const datum db_data
){
    the_data->n_data_ptr = (int*)&the_data->raw_data[0];
    set_datum(the_data, db_data);
}
/mnt/sabina/sdo/tools/local/proto/p_data.c [Preview]

        copy_p_data(&cur_entry, cur_value);

pq.c
```

Figure 23-11: **:ptag** example.

A whole set of commands is designed to manipulate the file in the preview window. The commands are as follows:

:pedit <i>[!] [+opt] [+cmd] {file}</i>	Open a preview window and do an :edit (:e) command in it.
:ped <i>[!] [+opt] [+cmd] {file}</i>	:edit (:e) command in it.
:ppop	Do a :pop command in the preview window.
:pp	
:ptselect <i>identifier</i>	Open a preview window and do a :tselect .
:pts <i>identifier</i>	
:ptjump <i>identifier</i>	Open a preview window and do a :tjump .
:ptj <i>identifier</i>	
:[count] ptnext	Do a :[count] tnext in the preview window.
:[count] ptn	
:[count] ptprevious	Do a :[count] tprevious in the preview window.
:[count] ptp	
:[count] ptNext	
:[count] ptN	
:[count] ptrewind	Do a :[count] trewind in the

:[count] ptr	preview window.
:[count] ptfirst	
:[count] ptf	
:ptlast	Do a :tlast in the preview window.
:ptl	
CTRL-W}	Do a :ptag on the word under the cursor.
CTRL-Wg}	Do a :ptjump on the word under the cursor.
CTRL-W CTRL-G }	

Vim also sets the option '**previewwindow**' ('**pvw**') to indicate that the window is the preview window.

Match Options

The '**matchpairs**' ('**mps**') option controls what characters are matched by the % command. The default value of this option is as follows:

```
:set matchpairs = (:),{:,}[:]
```

This tells *Vim* to match pairs of (), [], and {}. To match <> (useful if you are editing HTML documents), for example, use the following command:

```
:set matchpairs=<:>
```

This matches just <> pairs. If you want to match <> in addition to the other characters, you need this command:

```
:set matchpairs=(:),{:,}[:],<:>
```

This is a little long and awkward to type. The += flavor of the :set command adds characters to an option. Therefore, to add <> to the match list, use the following command:

```
:set matchpairs+=<:>
```

Showing Matches

If you execute the following command, when you enter any type of bracket ((,), [,], {, }), *Vim* will cause the cursor to jump to the matching bracket briefly when entering:

```
:set showmatch
```

('showmatch' can be abbreviated as 'sm')

The Vim Tutorial and Reference

Generally this jump lasts only for a half second, but you can change it with the '**matchtime**' ('**mat**') option. If you want to make it 1.5 seconds, for instance, use the following command:

```
:set matchtime=15
```

The value of this option is 1/10 second.

Finding Unmatched Characters

The **[{** command finds the previous unmatched { (see Figure 23-12). The **] {** command finds the next unmatched {. Also, **[}** finds the next unmatched }, whereas **] }** finds the previous unmatched }.

```
int main()
{
    if (flag) [{ ←
        do_part2();
        do_part1();
    }
    return (0); }
```

Figure 23-12: The **[{** command.

The **])** command finds the next unmatched). The **[(** finds the previous unmatched (. The command **[#** finds the previous unmatched #if or #else (see Figure 23-13). The command **] #** finds the next unmatched conditional.

```
#ifdef FOO
#   define SIZE 1
#else /* FOO */
#   ifdef BAR
#       define SIZE 20
#   else /* BAR */
#       define SIZE 30
#   define WIDTH 10
#endif
```

Figure 23-13: The **[#** command.

These commands are not that reliable because matching by mechanical means is impossible. It is possible to tell that you have three { and two }, but Vim can only guess at which { is missing a }.

Method Location

The following commands move to the beginning or end of a Java method:

- | | |
|----|--|
| [m | Search backward for the start of a method. |
| [M | Search backward for the end of a method. |
|]m | Search forward for the start of a method. |
|]M | Search forward for the end of a method. |

Movement

Several movement commands are designed to help programmers navigate through their text. The first set finds the characters { and } in column 1. (This usually indicates the start of a procedure, structure, or class definition.) The four curly brace-related movement commands are as follows:

count [[Move backward to the preceding { in column 1.
count []	Move backward to the preceding } in column 1.
count]]	Move forward to the next { in column 1.
count][]	Move forward to the next } in column 1.

Figure 23-14 shows how these commands work.

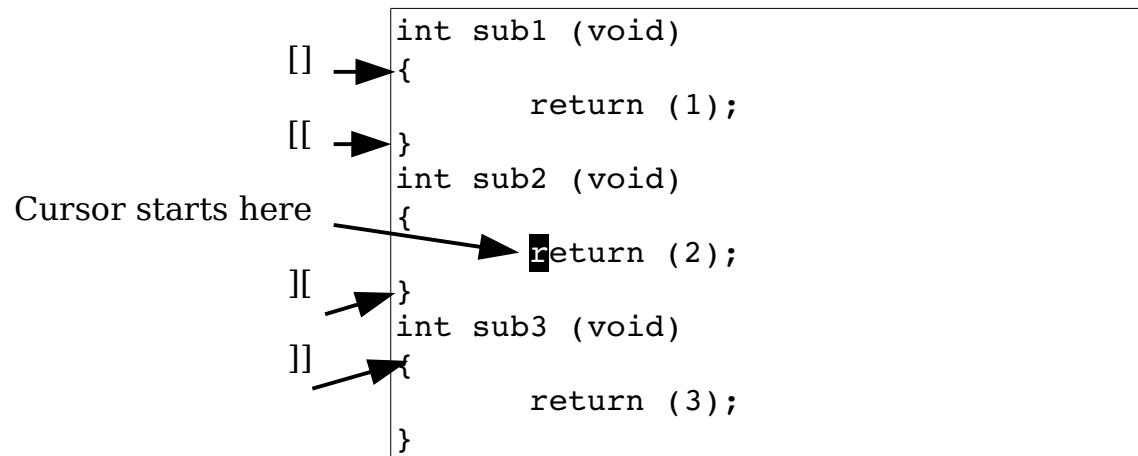


Figure 23-14: Curly brace movement commands.

Comment Moves

The commands `[/` and `[*` move you backward to the start of the first C comment it can find. The commands `] /` and `] *` move you forward to the end of the next C comment it can find. Figure 23-15 illustrates some simple comment motions.

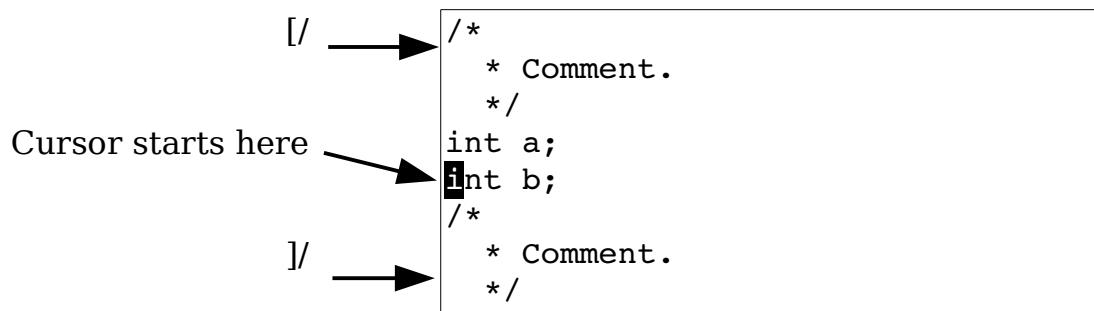


Figure 23-15: Comment motions.

Dealing with Multiple Directories

As programming projects grow larger and larger, you might find it convenient to organize things in different directories. Take a look at a small project. You have a *main* directory that contains *main.c* and *main.h*. The other directory is *lib* and it contains *lib.c* and *lib.h*. (This naming has no imagination, but it does make a good example.) Figure 23-16 shows this example organization.

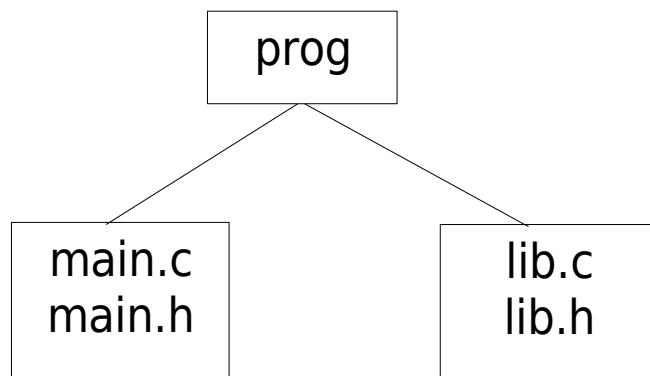


Figure 23-16: File layout.

The Vim Tutorial and Reference

You start editing in the directory *main*. The first thing you need to do is tell *Vim* about your new directory. You use the `:set ^=(:se ^=)` command to put the directory at the top of the search path with the following command:

```
:set path ^= ../lib
```

('path' can be abbreviated 'pa'.)

Suppose you are editing the file *main.c*. The file looks like this:

```
#include "main.h"  
#include "lib.h"  
  
int main(int argc, char *argv[])
```

Now you need to check out a subroutine declaration in *lib.h*. One way of going to that file is to execute the following command:

```
:vi ../lib/lib.h
```

This assumes you know where *lib.h* resides. But there is a better way. First, you position the cursor over the filename in the following line:

```
#include "lib.h"
```

Now you execute the command **gf**. The *Vim* editor tries to edit the filename which is under the cursor. The editor searches for the file in each directory in the path variable. In this case it will find the file even though it lives in another directory.

Suppose, however, that you want to edit the file *lib.c*. This name does not appear in the text, so you cannot use the **gf** command. Instead, you execute the following command:

```
:find lib.c
```

(**:fi** is the diminutive form of **:find**.⁶)

This acts just like the **:vi** command, except the editor searches for the file along the path. The following command does the same thing, but it splits the window and then does a **:find**:

```
:sfind lib.c
```

(**:sf** is the abbreviation for **:sfind**.)

⁶ I'm sorry but with all these abbreviations, I'm running out of ways to describe them.

The Vim Tutorial and Reference

The **gf** command acts like a **:find** command but use the word under the cursor as the name of the file to edit. If there is more than one file along the '**path**' that matches the given file name, then you can select which one *Vim* edits by giving the **gf** command a count.

In other words if you position the cursor on the name *param.h* and execute the command **2gf**, *Vim* will edit the second *param.h* file it finds looking through the directories specified by the '**path**' option.

The **Jf** and **[f** command are older, depreciated versions of the **gf** command.

The Java language is a little funny. The syntax of the **import** directive is designed to decouple the name of the imported class from the file system. So the directive:

```
import com.steve.database;
```

actually refers to the file *com/steve/database.java*. To handle this *Vim* uses two options. The first '**includeexpr**' ('**inex**') defines an expression that *Vim* uses to translate names in the text to file names. The second '**suffixesadd**' ('**sua**') defines the suffixes to add to a name in the text to convert it to a file name for the **gf** command.

The include Path

The '**path**' ('**pa**') option is used by *Vim* to tell it where to look for files that were included in the current file. The format of this option is as follows:

```
:set path=directory,directory,...
```

The parameter directory is a directory to search. For example:

```
:set path=/usr/include,/usr/X11R6/include
```

You can use wildcards (*) in any directory of the path specification:

```
:set path=/usr/include,/usr/include/*
```

There are a number of special directories:

** Match an entire tree. For example:

```
:set path=/usr/include/**
```

This command searches */usr/include* and all its subdirectories. The following path specification searches the files in any directory that starts with */home/oualline/progs* and ends with *include*:

The Vim Tutorial and Reference

```
:set path=/home/oualline/progs/**/include ""
```

<empty>

The empty string indicates the current directory. (For example, the middle directory of the trio **first,, last.**)

- **(dot)**

The directory in which the file being edited resides.

For example, the following command tells *Vim* to search */usr/include* and all its subdirectories, the directory in which the file resides (.), and the current directory (,,).

```
:set path=/usr/include/**,,,
```

Checking the Path

To make sure that you can find all the #include files, you can execute the following command:

```
:checkpath
```

(:**che** for short.)

This command works not only on the #include directives in the file you are editing, but also on any files that they #include and so on. The result is that all #include files are checked.

Figure 23-17 shows how this command works.

In this case, a number of files include the files *stddef.h* and *stdarg.h*. But *Vim* cannot find these files. If you want to tell *Vim* to search the Linux-specific include directory, you can execute the following command:

```
:set path+=/usr/include/linux
```

The Vim Tutorial and Reference

```
        get_rel_name(&cur_entry, i),
        get_full_name(&cur_entry, i));
    }
    if (gdbm_errno != 0) {
-- Included files not found in path --
/usr/include/stdio.h -
<stddef.h>
<stdarg.h>
/usr/include/bits/types.h --
<stddef.h>
/usr/include/libio.h --
/usr/include/_G_config.h --
<stddef.h>
<stdarg.h>
/usr/include/bits/stdio-lock.h --
/usr/include/pthread.h --
/usr/include/sched.h --
/usr/include/time.h --
<stddef.h>
/usr/include/stdlib.h --
<stddef.h>
/usr/include/sys/types.h --
<stddef.h>
/usr/include/alloca.h --
<stddef.h>
/usr/include/string.h --
<stddef.h>
Press RETURN or enter command to continue
```

Figure 23-17: The **:checkpath** command.

Now do another:

```
:checkpath
```

Figure 23-18 shows the results.

```
for (i = 0; i < *cur_entry.n_data_ptr; i++) {
    printf("\t%d %s (%s)\n",
           (int)get_flags(&cur_entry, i),
           get_rel_name(&cur_entry, i),
           get_full_name(&cur_entry, i));
}
if (gdbm_errno != 0) {
All included files were found
```

Figure 23-18: **:checkpath** with all files found.

The Vim Tutorial and Reference

This command lists only the files that cannot be found. If you want to list all `#include` files, use this command:

```
:checkpath!
```

Figure 23-19 shows the results.

```
Included files in path
<stdio.h>
/usr/include/stdio.h >
    <features.h>
        /usr/include/features.h >
            <sys/cdefs.h>
                /usr/include/sys/cdefs.h >
                    <features.h> (Already listed)
                    <gnu/stubs.h>
    <stddef.h>
    <stdarg.h>
    <bits/types.h>
    /usr/include/bits/types.h >
        <features.h> (Already listed)
        <stddef.h> (Already listed)
        <bits/pthreadtypes.h>
        /usr/include/bits/pthreadtypes.h >
            <bits/sched.h>
    <libio.h>
    /usr/include/libio.h >
More █
```

Figure 23-19: The `:checkpath!` command.

Defining a Definition

The *Vim* editor knows about C and C++ macro definitions. But what about other languages? The option '`define`' ('`def`') contains the regular expression that *Vim* uses when it looks for a definition. To have *Vim* look for macros that start with the string `function`, for instance, use the following command:

```
:set define=function
```

Locating `include` Files

The '`include`' ('`inc`') option defines what an include directive looks like. This option is used for the `]CTRL-I`, `[CTRL-I`, `]d`, and `[d` searches that look through `#include'd` files.

The Vim Tutorial and Reference

In Java you specify packages. The package names happen to look a lot like file names except that the slashes (/) have been changed to dots (.). In order to make transformation from Java package to actual file path, *Vim* uses the value of the '**includeexpr**' ('**inex**') to change the name in the program to a name that the file system understands.

This option is used for the **:checkpath** command as well. Like the '**define**' option, the value of this option is a regular expression.

The **[i** command searches for the first occurrence of the word under the cursor. Text inside comments is ignored.

The **]i** command searches for the next occurrence of the word under the cursor. Again, text inside comments is ignored.

The **[I** command lists all the lines which contain the keyword under the cursor. (Comments ignored.) The **]I** command does the same thing starting at the current cursor location.

Multiple Error Lists

The **:make** (**:mak**) command generates an error list. The *Vim* editor remembers the results of your preceding 10 **:make**, **:grep**, **:gr**, **:vimgrep** or **:vim** commands. To go to a previous error list, use the following command:

:colder

(or **:col.**)

To go to a newer one, use this command:

:cnewer

(or **:cnew.**)

Manipulating the quick fix list

The **:cfile** (**:cf**) command takes a file and loads it into the quick fix list. For example:

:cfile error-list.log

The **:caddfile** (**:caddf**) does the same thing only the file is added to the quick fix list.

The Vim Tutorial and Reference

There are a whole lot of other commands that change the contents of the quick fix list. These are:

Command	Add or Replace	Jump or No Jump	Description
<code>:cfile {file}</code> <code>:cf {file}</code>	Replace	Jump	Replace error list with file.
<code>:cgetfile {file}</code> <code>:cg {file}</code>	Replace	No Jump	Replace error list with file.
<code>:caddfile {file}</code> <code>:caddf {file}</code>	Adds	No Jump	Add file to error list.
<code>:cbuffer {buffer}</code> <code>:cb {buffer}</code>	Replace	Jump	Replace error list with buffer.
<code>:cgetbuffer {buffer}</code> <code>:cgetb {buffer}</code>	Replace	No Jump	Replace error list with buffer.
<code>:caddbuffer {file}</code> <code>:cad {biffer}</code>	Adds	No Jump	Add buffer to error list.
<code>:cexpr {expr}</code> <code>:cex {expr}</code>	Replace	Jump	Replace error list with the expression.
<code>:cgetexpr {expr}</code> <code>:cgetx {expr}</code>	Replace	No Jump	Replace error list with expression.
<code>:cadexpr {expr}</code> <code>:cadde {expr}</code>	Adds	No Jump	Add the expression to error list.

Local error lists

Vim actually maintains two different types of lists. So far we've been using the error list which is the same no matter what the current buffer is. There is also a local list which is local to a buffer. The two are very similar and all of the error list commands have local list equivalents. The following tables lists the various commands.

Error List Command	Location List Command	Description
<code>:cc</code>	<code>:1l</code>	Display current error.

The Vim Tutorial and Reference

:cn :cnext	:lne :lnext	Display next error.
:cN :cNext :cp :cprevious	:lN :lNext :lp :lprevious	Display previous error.
:cnf :cnfile	:lnf :lnfile	Display first error in the next file.
:cNf :cNfile :cpf :cpfile	:lNf :lNfile :lpf :lpfile	Display last error in the previous file.
:cr :crewind :cfir :cfirst	:lr :lrewind :lfir :lfirst	Display the first error.
:cla :clast	:lla :llast	Display the last error.
:cf :cfile	:lf :lfile	Read errors from a file. Jump to first.
:cg :cgetfile	:lg :lgetfile	Read errors from a file. No jump.
:caddf :caddfile	:laddf :laddfile	Add lines from a file to the error list
:cb :cbuffer	:lb :lbuffer	Read errors from a buffer. Jump to first
:cgetb :cgetbuffer	:lgetb :lgetbuffer	Read errors from a buffer. No jump.
:cad :caddbuffer	:laddb :laddbuffer	Add lines from a buffer to the error list.
:cex :cexpr	:lex :lexpr	Create error list from expression. Jump to first
:cgete :cgetexpr	:lgete :lgetexpr	Create error list from expression. No jump.
:cadde :caddeexpr	:lad :laddexpr	Add expression to the error list.

The Vim Tutorial and Reference

:cl :clist	:lli :llist	List errors
:cope :copen	:lop :lopen	Open a window containing the error list
:ccl :cclose	:lcl :lclose	Close the error list window
:cw :cwindow	:lw :lwindow	Open the quick fix window if needed.
:col :colder	:lol :lolder	Go to older error list
:cnew :cnewer	:lnew :lnewer	Go to newer error list
:mak :make	:lmak :lmake	Run the make program and capture the results.
:vim :vimgrep	:lv :lvimgrep	Generate results using the internal grep
:vimgrepa : :vimrepadd	: :lvimgrepa :lvimrepadd	Add grep results to the list
:gr :grep	:lgr :lgrep	Run the external <i>grep</i> command and capture the results
:grepa :grepadd	:lgrepa :lgrepadd	Run the external <i>grep</i> command and add the result to the quick fix list
:tag :ta	:ltag :lt	Jump to a given tag.

Customizing the :make Command

The name of the program to run when the **:make** command is executed is defined by the '**makeprg**' ('**mp**') option. Usually this is set to *make*, but Visual C++ users should set this to *nmake* by executing the following command:

```
:set makeprg=nmake
```

The **:make** command redirects the output of *make* to an error file. The name of this file is controlled by the '**makeef**' ('**mef**') option. If this option contains the characters **##**, the **##** will be replaced by a unique number. The default value for this option depends on the operating system you are on. The defaults are as follows: Amiga UNIX : Microsoft Windows and others

The Vim Tutorial and Reference

Amiga	<i>t:vim##.Err</i>
UNIX, Linux, FreeBSD	<i>/tmp/vim##.err</i>
Microsoft Windows and others	<i>vim##.err</i>

You can include special *Vim* keywords in the command specification. The % character expands to the name of the current file. So if you execute the command

```
:set makeprg=make\ %
```

and you do a

```
:make
```

it executes the following command:

```
$ make file.c
```

The parameter *file.c* is the name of the file you are editing. This is not too useful, so you will refine

the command a little and use the :r (root) modifier:

```
:set makeprg=make\ %:r.o
```

Now if you are editing *file.c*, the command executed is as follows:

```
$ make file.o
```

The Error Format

The option '**errorformat**' ('**efm**') controls how *Vim* parses the error file so that it knows the filename and line number where the error occurred. The format of this option is as follows:

```
:set errorformat=string,string,string
```

The string is a typical error message with the special character % used to indicate special operations (much like the standard C function **scanf**). The special characters are as follows:

%f	Filename
%l	Line number
%c	Column
%t	Error type (a single character)
%n	Error number
%m	Error message
%r	Matches the remainder of the line
%*char	Matches (and skips) any scanf conversion specified by char.
%%	The character %

The Vim Tutorial and Reference

When compiling a program, you might traverse several directories. The GNU *make* program prints a message when it enters and leaves a directory. A sample make log looks like this:

```
make[1]: Entering directory '/usr/src/linux-2.2.12'
make -C kernel fastdep
make[2]: Entering directory '/usr/src/linux-2.2.12/kernel'
/usr/src/linux/scripts/mkdep sysctl.c time.c > .depend
make[2]: Leaving directory '/usr/src/linux-2.2.12/kernel'
make -C drivers fastdep
make[2]: Entering directory '/usr/src/linux-2.2.12/drivers'
/usr/src/linux/scripts/mkdep > .depend
make[3]: Entering directory '/usr/src/linux-2.2.12/drivers'
make -C block fastdep
make[4]: Entering directory '/usr/src/linux-2.2.12/drivers/block'
/usr/src/linux/scripts/mkdep xd.c xd.h xor.c z2ram.c > .depend
make _sfdep_paride _FASTDEP_ALL_SUB_DIRS=" paride"
make[5]: Leaving directory '/usr/src/linux-2.2.12/drivers/block'
make[4]: Leaving directory '/usr/src/linux-2.2.12/drivers/'
```

To get the filename right *Vim* needs to be aware of this change. The following error format specifications are used to tell *Vim* about directory changes:

- %D Specifies a message printed on entering a directory. The %f in this string indicates the directory entered.
- %X Specifies the leave directory message. The %f in this string specifies the directory that make is done with.

Some compilers, such as the GNU GCC compiler, output very verbose error messages. The GCC error message for an undeclared variable is as follows:

```
tmp.c: In function 'main':
tmp.c:3: 'i' undeclared (first use in this function)
tmp.c:3: (Each undeclared identifier is reported only once
tmp.c:3: for each function it appears in.)
```

If you use the default *Vim* '**errorformat**' settings, this results in three error messages. This is really annoying. Fortunately, the *Vim* editor recognizes multiline error messages. The format codes for multiline error messages are as follows:

- %A Start of a multiline message (unspecified type)
- %E Start of a multiline error message
- %W Start of a multiline warning message
- %C Continuation of a multiline message
- %Z End of a multiline message
- %G Global; useful only in conjunction with + or -
- %O Single-line file message: overread the matched part
- %P Single-line file message: push file %f onto the stack

The Vim Tutorial and Reference

%Q Single-line file message: pop the last file from stack

A + or - can precede any of the letters. These signify the following

%-letter Do not include the matching line in any output.

%+letter Include the whole matching line in the **%m** error string.

Therefore, to define a format for your multiline error message, you begin by defining the start message. This matches the following:

```
tmp.c:3: 'i' undeclared (first use in this function)
```

The error message specification is:

```
%E%f:%l:\ %m\ undeclared\ (first\ use\ in\ this\ function)
```

Note the use of \ to tell *Vim* that the space is part of the string.

Now you have a problem. If you use this definition, the **%m** will match just 'i'. You want a longer error message. So you use + to make *Vim* put the entire line in the message:

```
%+E%f:%l:\ %m\ undeclared\ (first\ use\ in\ this\ function)
```

The middle matches this:

```
tmp.c:3: (Each undeclared identifier is reported only once)
```

This translates into the error string:

```
%-C%f:%l:\ (Each\ undeclared\ identifier\ is\ reported\ only\ once)
```

Note the use of the - modifier to keep this message out of the list of messages.

The end of the error is as follows:

```
tmp.c:3: for each function it appears in.)
```

which results in the string:

```
%-Z%f:%l:\ for\ each\ function\ it\ appears\ in.)
```

So you add these three lines to the error format:

```
%+E%f:%l:\ '%*\k*'\\ undeclared\\ (first\\ use\\ in\\ this\\ function),  
%-C%f:%l:\ (Each\\ undeclared\\ identifier\\ is\\ reported\\ only\\ once,  
%-Z%f:%l:\ for\\ each\\ function\\ it\\ appears\\ in.)
```

Now this works, but there is a slight problem. When the GNU compiler encounters the second undefined variable, it does not output the three-line message. Instead, it outputs just the first line. (It figures you have already seen the stuff in parenthesis, so why output it again.)

The Vim Tutorial and Reference

Unfortunately, your error specification tries to match all three lines. Therefore, you need a different approach. The solution is to globally tell *Vim* to forget about the second two lines:

```
%-G%f:%l:\ (Each\ undeclared\ identifier\ is\ reported\ only\ once  
%-G%f:%l:\ for\ each\ function\ it\ appears\ in.)
```

Now all you have to do is to add this option to your *.vimrc* file. You can just add them on to the '**errorformat**' option by using the following command:

```
" This will not work  
:set errorformat+=  
 \%-G%f:%l:\ (Each\ undeclared\ identifier\ is\ reported\ only\ once,  
 \%-G%f:%l:\ for\ each\ function\ it\ appears\ in.)
```

Note that in *Vim*, continuation lines start with a backslash (\). Also, you have added a comma at the end of the first error message to separate it from the second.

There is only one problem with this technique: It doesn't work. The problem is that *Vim* goes through the list of strings in '**errorformat**' in order, stopping on the first one that matches. We are appending our messages to the existing list of messages, and the error string for the GNU compiler (%f:%l:%m) is defined before us.

It is matched first, and therefore you never get to your two new error messages. You need to put the more specific matches (your two new messages) at the beginning. This is accomplished with the following command:

```
" This will work  
:set errorformat ^=  
 \%-G%f:%l:\ (Each\ undeclared\ identifier\ is\ reported\ only\ once,  
 \%-G%f:%l:\ for\ each\ function\ it\ appears\ in.)
```

Remember, the **:set ^=** command adds the string to the beginning of the list.

***The 'switchbuf* Option**

Normally when you do a **:make** and errors occur, *Vim* will display the offending file in the current window. If you set the '**switchbuf**' ('**swb**') option to **split**, then the editor will split the current window displaying the bad file in the new window. Note the '**switchbuf**' option can have the values: ''(nothing), '**split,useopen**' and '**split,useopen**'. For a description of the **useopen** argument see *Chapter 5: Windows and Tabs*.

The Vim Tutorial and Reference

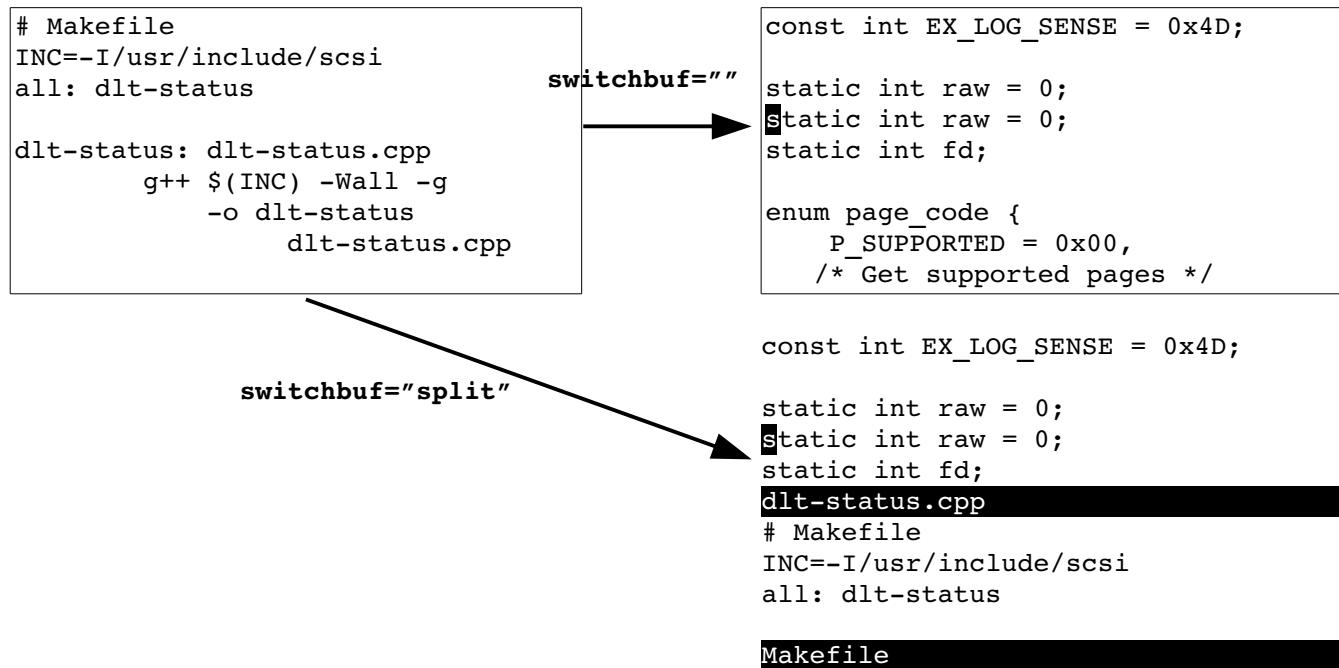


Figure 23-20: The '**switchbuf**' option.

Customizing :grep

The **:grep** (**:gr**) command runs the program specified by the '**grepprg**' ('**gp**') option. This option contains the command line to use. The '#' and '%' characters will be expanded to be the names of the current and alternate file.

Also the string '\$*' will be replaced by any arguments to the **:grep** command. Note that on UNIX, the '**grepprg**' defaults to *grep -n*. On Microsoft Windows, it defaults to *findstr/s*. The capabilities of these two programs differ vastly.

The **:grep** command uses the '**grepformat**' option to tell *Vim* how to parse the output of grep. (It uses the same format as the '**errorformat**' option.)

Defining How a Tag Search Is Done

Usually *Vim* does a binary search for a given tag name. This makes things quick if the tag file is sorted. Otherwise, a linear search is performed. To force a linear search, use this command:

```
:set notagbsearch
```

(The sort form of the '**tagbsearch**' option is '**tagb**').

The Vim Tutorial and Reference

The '**notagbsearch**' option is useful if your tag file is not sorted.

Some systems limit the number of characters you can have in a function name. If you want this limit to be reflected in *Vim*, you can set the '**taglength**' ('**tl**') option to the maximum length of your function names.

You specify the name of the tags file with the '**tags**' ('**tag**') option. This can be made to point to a file in another directory. For example:

```
:set tags+=/home/oualline/tools/vim/tags
```

But this causes a little confusion. Did you start in the current directory and tell *ctags* to put the tag file in the directory */home/oualline/tools/vim* or did you execute the *ctags* command in this directory?

The *Vim* editor solves this problem with yet another option. If you set the following, all tags are relative to the directory that contains the tag file:

```
:set tagrelative
```

otherwise, they are relative to the current directory. ('**tr**' is short for '**tagrelative**'.)

With the '**tagstack**' option set, the **:tag** and **:tjump** commands build a tag stack. Otherwise, no stack is kept.

Customizing the Syntax Highlighting

The *Vim* editor enables you to customize the colors used for syntax hightlighting. The *Vim* editor recognizes three different types of terminals:

term	A normal black-and-white terminal (no color)
cterm	Color terminal, such as xterm or the Microsoft Windows MS-DOS window.
gui	A window created by <i>gvim</i>

Black-and-White Terminals

To change the highlighting for a normal terminal, use this command:

```
:highlight group-name term=attribute
```

(**:hi** for short.)

The *group-name* is the name of the syntax group to be highlighted. This is the name of a syntax-matching rule set used by *Vim* to tell what part of the program to highlight. A list of the standard group names can be found later in this section.

The Vim Tutorial and Reference

The *attribute* is a terminal attribute. The terminal attributes for a normal black-and white terminal are as follows:

bold ***italic***

underline reverse (also standout called inverse)

You can combine attributes by separating them with commas, as follows:

```
:highlight Keyword term=reverse,bold
```

Suppose, however, that you have a terminal that has very unusual terminal codes. You can define your own attributes with the **start** and **stop** highlight options. These define a string to be sent to start the color and one to stop it. For example:

```
:highlight Keyword start=<Esc>X stop=<Esc>Y
```

With this definition, when *Vim* displays keywords (for example, **if**, it will output <Esc>Xif<Esc>Y). If you are familiar with the terminal definition files used on UNIX (called *termcap* or *terminfo* files), you can use terminal codes. The *termcap* entry **us** defines the underline start code, for example, and **ue** is the exit underline-mode string. To specify these in a highlight entry, you use the following command:

```
:highlight Keyword start=t_us stop=t_ue
```

Color Terminals

The color entries are defined by the *cterm* settings. You can set them using **cterm=attribute** just like a normal term entry.

But there are additional options for a color terminal. The setting **ctermfg=colordnumber** defines the foreground color number. The **ctermbg=colordnumber** defines the background .

Color names are recognized as well as color numbers. The following tells *Vim* to display comments in red on blue, underlined:

```
:highlight Comment cterm=underline ctermfg=red ctermbg=blue
```

(Incidentally, this looks really ugly.)

GUI Definition

The GUI terminal uses the option **gui=attribute** to display the attributes of a syntax element in the GUI window. The options **guifg** and **guibg** define the colors. These colors can be named. If the name contains a space, the color name should be enclosed in single quotation marks. To keep things portable, the *Vim* people suggest you limit your color names to the following.

Black	Blue	Brown	Cyan
DarkBlue	DarkCyan	DarkGray	DarkGreen
DarkMagenta	DarkRed	Gray	Green
LightBlue	LightCyan	LightGray	LightGreen
LightMagenta	LightRed	LightYellow	Magenta
Orange	Purple	Red	SeaGreen
SlateBlue	Violet	White	Yellow

You can define the color as well by using the standard X11 color numbers. (This works on all systems, regardless of whether you are using X11.) These are of the form **#rrggbb**, where **rr** is the amount of red, **gg** is the amount of green, and **bb** is the amount of blue. (These three numbers are in hexadecimal.) Under Microsoft Windows, the following colors are available:

Black	Blue	Brown
Cyan	DarkBlue	DarkCyan
DarkGray	DarkGreen	DarkMagenta
DarkRed	Green	LightBlue
LightCyan	LightGray	LightGreen
LightMagenta	LightRed	Magenta
Red	Sys_3DDKShadow	Sys_3DFace
Sys_3DHighlight	Sys_3DHilight	Sys_3DLight
Sys_3DShadow	Sys_ActiveBorder	Sys_ActiveCaption
Sys_AppWorkspace	Sys_Background	Sys_BTNFace
Sys_BTNHighlight	Sys_BTNHilight	Sys_BTNShadow
Sys_BTNTText	Sys_CaptionText	Sys/Desktop
Sys_GrayText	Sys_Highlight	Sys_HighlightText
Sys_InactiveBorder	Sys_InactiveCaption	Sys_InactiveCaptionText
Sys_InfoBK	Sys_InfoText	Sys_Menu
Sys_MenuText	Sys_ScrollBar	Sys_Window
Sys_WindowFrame	Sys_WindowText	White
Yellow		

You can use the **font=x-font** as well to define which font to use. This is not for the faint of heart, because X11 font names are complex. For example:

```
:highlight Comment font=
```

The Vim Tutorial and Reference

```
\font=-misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
```

Microsoft Windows fonts can be used as well:

```
:highlight Comment font=courier_helv:h12
```

Combining Definitions

You can define colors for multiple terminals in a single highlight command. For example:

```
:highlight Error term=reverse cterm=bold ctermfg=7 ctermbg=1
```

Syntax Elements

The syntax elements are defined by the macros in `$VIMRUNTIME/syntax`. To make things easier, however, the following names are generally used.

Boolean	Character	Comment	Conditional
Constant	Debug	Define	Delimiter
Error	Exception	Float	Function
Identifier	Include	Keyword	Label
Macro	Number	Operator	PreCondit
PreProc	Repeat	Special	SpecialChar
SpecialComment	Statement	StorageClass	String
Structure	Tag	Todo	Type
Typedef			

In addition to these syntax elements, *Vim* defines the following for the various things it generates:

Cursor	The character under the cursor.
Directory	Directory names (and other special names in listings).
ErrorMsg	Error messages displayed on the bottom line.
IncSearch	The result of an incremental search.
Mode Msg	The mode shown in the lower-left corner (for example, --INSERT--).
MoreMsg	The prompt displayed if <i>Vim</i> is displaying a long message at the bottom of the screen and must display more.
NonText	The <i>Vim</i> editor displays ~ for lines past the end of the file. It also uses @ to indicate a line that will not fit on the screen. (See <i>Chapter 20: Advanced Text Blocks and Multiple Files</i> .) This syntax element defines what color to use for these elements.
Question	When <i>Vim</i> asks a question.

The Vim Tutorial and Reference

SpecialKey	The :map command lists keyboard mapping. This defines the highlight to use for the special keys, such as <Esc> , displayed.
StatusLine	The status line of current window.
StatusLineNC	Status lines of the other windows.
Title	Titles for output from :set all , :autocmd , and so on.
Visual	This color is used to highlight the visual block.
VisualNOS	Visual-mode selection when <i>Vim</i> is "Not Owning the Selection." This works only on X Windows Systems.
WarningMsg	Warning messages displayed on the last line of the window.
WildMe nu	Current match in ' wildmenu ' completion.
LineNr	Line number for :number and :# commands, and when the ' number ' option is set.
Normal	Normal text.
Search	The results of the last search when the ' hlsearch ' option is enabled.
User1 through User9	The ' statusline ' option enables you to customize the status line. You can use up to nine different highlights on this line, as defined by these names.
Menu	Menu color for the GUI.
Scrollbar	Scrollbar color for the GUI

Color Chart

If you want to see what the various colors look like on your terminal, you can use *Vim*'s color chart. To access this chart, either pull down the Syntax|Color test menu (*gvim*) or follow these steps:

1. Edit the file **\$VIMRUNTIME/syntax/colortest.vim**. Your directories might be different if you install *Vim* in a different place.
2. Execute a **:source (:so)** command to read in the test:

```
:source %
```

3. Browse the color list (in the third column). Figure 23-21 shows the results. Unfortunately, this book is in black and white, but you can imagine what it would look like in color.

The Vim Tutorial and Reference

```
                                grey on black
" lightred      lightred_on_white
"                               lightred on black
" lightgreen     lightgreen_on_white
"                               lightgreen on black
" lightyellow    lightyellow_on_white
"                               lightyellow on black
" lightblue      lightblue_on_white
"                               lightblue on black
" lightmagenta  lightmagenta_on_white
"                               lightmagenta on black
" lightcyan      lightcyan_on_white
"                               lightcyan on black

" Open this file in a window if it isn't edited yet.
" Use the current window if it's empty.
if expand('%:p') != expand('<file>:p')
  if &mod || line('$') != 1 || getline(1) != ''
    exe "new " . expand('<file>')
  else
    exe "edit " . expand('<file>')
  endif
endif
:source %
```

```
black_on_grey
white_on_lightred
black_on_lightred
white_on_lightgreen
black_on_lightgreen
white_on_lightyellow
black_on_lightyellow
white_on_lightblue
black_on_lightblue
white_on_lightmagenta
black_on_lightmagenta
white_on_lightcyan
black_on_lightcyan
```

Figure 23-21: Color test.

The '`syntax`' Option

The '`syntax`' ('`syn`') option contains the name of the current language used for syntax highlighting. You can turn syntax highlighting off by entering the command:

```
:set syntax=off
```

To turn it back on, use this command:

```
:set syntax=on
```

Note: The **on** and **off** value for the '`syntax`' option have special meaning, they are not the name of programming languages, yet.

Chapter 24: All About Abbreviations and Keyboard Mapping

In *Chapter 8: Basic Abbreviations, Keyboard Mapping, and Initialization Files* you learned about abbreviations and keyboard mappings, but that discussion focuses on only the most useful subset of the commands. This chapter examines things in complete detail. These commands have a lot of different variations, and this chapter covers them all. In this chapter, you learn about the following:

- How to remove an abbreviation
- Creation of mode-specific abbreviations
- Listing abbreviations
- How to force abbreviation completion in insert mode
- Mode-specific mappings
- Clearing and listing mappings
- Other mapping options

Removing an Abbreviation

To remove an abbreviation, use the command **:unabbreviate (:una)**. Suppose you have the following abbreviation, for example:

```
:abbreviate @a fresh
```

(The abbreviation of **:abbreviate** is **:ab.**)

You can remove it with this command:

```
:unabbreviate @a
```

To clear out all the abbreviations, use the following command:

```
:abclear
```

(You can use **:abc** as well.)

The Vim Tutorial and Reference

Note: One problem with this command is that the abbreviation @a is expanded on the command line. *Vim* is smart, however, and it will recognize that `fresh` is really @a expanded and will remove the abbreviation for @a.

Abbreviations for Certain Modes

The `:abbreviate` command defines abbreviations that work for both insert mode and command-line mode. If you type the abbreviation @a in the text, for example, it will expand to `fresh`. Likewise, if you put it in a command-mode (:) command, it will also expand .

Normal mode	<code>i@ a< ESC></code>	Inserts <code>fresh</code>
Command mode	<code>:s/xx/@a/</code>	Executes <code>:s/xx/fresh/</code>

If you want to define an abbreviation that works only in insert mode, you need the `:iabbrev (:ia)` command:

```
:iabberv @a fresh
```

This means that in command mode, @a is just @a. The `:noremap (:no)` version of this command is `:inoreabbrev (:inorea)`. To unabbreviate an insert-mode abbreviation, use the command `:iunabbreviate (:inua)`. To clear out all the insert abbreviations, use the following command:

```
:iabclear
```

(Or you can shorten this to `:iabc.`)

If you want an abbreviation defined just for command mode, use the `:cababbreviate (:ca)` command. The `:noremap` version of this command is `:cnoreabbrev (:cnorea)`. To remove a definition, use the `:cunabbreviate (:cuna)` command; and to clear out the entire abbreviation list, use the command `:cabclear (:cabc)`.

Listing Abbreviations

You can list all abbreviations by using the `:abbreviate (:ab)` command with no arguments (see Figure 24-1).

```
~  
~  
~  
c r      :rewind  
i ab     abbreviate  
! h      Help  
Press RETURN or enter command to continue
```

Figure 24-1: **:abbreviate** output.

The first column contains a flag indicating the abbreviation type. The flags are:

- c** Command mode
- i** Insert mode
- !** Both

Forcing Abbreviation Completion

In insert mode, the command **CTRL-]** causes *Vim* to insert the current abbreviation.

The command **CTRL-C** causes *Vim* to exit insert mode. The difference between **CTRL-C** and **<Esc>** is that **CTRL-C** does not check for an abbreviation before entering normal mode.

Mapping and Modes

The **:map** command enables you to define mappings limited to certain modes. Suppose, for example, that you want to use the **<F5>** key to yank the current visual-mode select into register **v**. You can define the following command:

```
:map <F5> "vy
```

This maps the **<F5>** key for normal, visual, and operator-pending modes. But you want this mapping to be valid only for visual mode. To do that, use a special version of the mapping command:

```
:vmap <F5> "vy
```

(Or **:vm** for short.)

The Vim Tutorial and Reference

The "v" flavor of the :map command tells *Vim* that this mapping is valid only for visual mode. Table 24-1 lists seven different flavors of the :map command.

Table 24.1 :map Commands

Command	Normal	Visual	Operator Pending	Insert	Command Line	Select Mode
:map	✓	✓	✓			
:nmap :nm	✓					
:vmap :vm		✓				
:omap :om			✓			
:map!				✓	✓	
:imap :im				✓		
:cmap :cm					✓	
:smap						✓

Note: Operator-pending mode is the mode that occurs when you enter a command such as d that expects a motion to follow. (For example, dw deletes a word. The w is entered in operator-pending mode.)

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

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

This causes <F7> to perform a select block (a{}) in operator-pending mode. With this mapping in place, when you press the d of d<F7>, you enter operator-pending mode. Pressing <F7> executes the command a{} in operator-pending mode, selecting the block. Because you are performing a d command, the block is deleted.

Other :map Commands

A number of commands relate to mapping. The first is this:

```
:map lhs rhs
```

This adds the mapping of **lhs** to **rhs**. Therefore, pressing **lhs** results in the execution of **rhs**. The **:map** command allows remapping of **rhs**. The command **:noremap** (**:no**) however, does not

```
:noremap lhs rhs
```

For example:

```
:map ^A dd  
:map ^B ^A
```

This causes *Vim* to delete a line when you type **CTRL-A**. It also causes the **CTRL-B** command to do the same thing as **CTRL-A** – that is, delete a line. Note: When entering the control characters, you must "quote" them with **CTRL-V**. In other words, you must type

```
:map CTRL-V CTRL-A dd
```

to get:

```
:map ^A dd
```

Suppose you use the following **:noremap** command:

```
:map ^A dd  
:noremap ^B ^A
```

When you type **CTRL-B**, you execute a normal **CTRL-A** (not mapped) **CTRL-A** command. Therefore, **CTRL-B** will now increment the value of the number under the cursor.

Undoing a Mapping

The **:unmap** (**:unm**) command removes a mapping. To cause a mapped **CTRL-A** command to revert to the default, use the following command:

```
:unmap ^A
```

This also proves useful if you want to map a command for a limited set of modes. To define a command that exists in only normal and visual modes, but not operator pending mode, for example, use the following commands:

```
:map ^A 3w  
:ounmap ^A
```

The Vim Tutorial and Reference

(**:ou** can be used for **:ounmap**.)

The first command maps the **CTRL-A** to **3w** in normal, visual, and operator-pending modes. The second removes it from the operating-pending mode map.

Clearing Out a Map

The command **:mapclear** (**:mapc**) removes all mapping:

```
:mapclear
```

Be careful with this one because it also removes any default mappings you might have.

Listing the Mappings

The **:map** command with no arguments lists out the mappings (see Figure 24-2).

```
~  
~  
~  
  
<xHome>          <Home>  
<xEnd>           <End>  
<SxF4>           <SF4>  
<SxF3>           <SF3>  
<SxF2>           <SF2>  
<SxF1>           <SF1>  
<xF4>            <F4>  
<xF3>            <F3>  
<xF2>            <F2>  
<xF1>            <F1>  
  
Press RETURN or enter command to continue
```

Figure 24-2: Output of :map command.

The first column lists flags indicating the modes for which the mapping is valid.

Character Mode

<Space> Normal, visual, and operator-pending

n Normal

The Vim Tutorial and Reference

v	Visual
o	Operator-pending
!	Insert and command line
i	Insert
c	Command line

The second column indicates the various **lhs** of any mappings. The third column is the value of the **rhs** of the mapping. If the **rhs** begins with an asterisk (*), the **rhs** cannot be remapped.

The **:map** command lists all the mappings for normal, visual, and operator-pending modes. The **:map!** command lists all the mappings for insert and command-line mode. The **:imap**, **:vmap**, **:omap**, **:nmap**, and **:cmap** commands list only the mappings for the given modes.

Recursive Mapping

By default, *Vim* allows recursive command mapping. To turn off this feature clear the '**remap**' with the command:

```
:set noremap
```

This may break some scripts. Using **:noremap** will avoid this problem.

Remapping Abbreviations

Abbreviations can cause problems with mappings. Consider the following settings, for example:

```
:abbreviate @a ad
:imap ad adder
```

Now when you type **@a**, the string **ad** is inserted. Because **ad** is mapped in insert mode to the string **adder**, the word **adder** is inserted in the text.

If you use the command **:noreabbrev** (**:norea**), however, you tell *Vim* to avoid this problem. Abbreviations created with this command are not candidates for mapping.

The Vim Tutorial and Reference

One of the problems with the `:abbreviate` command is that the abbreviations on the right side are expanded when the abbreviations are defined. There is a clumsy way of avoiding this: Type an extra character before the word, type the word, then go back and delete the extra character.

Language Dependent Mappings

The `:lmap` (`:lm`) command defines a mapping that's to be used in language dependent mode. This mode is encountered when you are entering text in command mode, insert mode, and search patterns that is not an actual *Vim* command, but text or arguments to that command.

The usual suite of commands applies. :map Mode Table

The following chart shows the commands and the various modes they are associated with:

<i>Mode</i>	<i>Commands</i>				
<i>Normal, Visual Operator Pending</i>	<code>:map</code>	<code>:noremap</code>	<code>:</code>	<code>unmap</code>	<code>:mapclear</code>
		<code>no</code>		<code>unm</code>	<code>:mapc</code>
<i>Normal</i>	<code>:nmap</code>	<code>:nnoremap</code>	<code>:</code>	<code>:unmap</code>	<code>:nmapclear</code>
	<code>:nm</code>	<code>nn</code>		<code>:nun</code>	<code>:nmapc</code>
<i>Visual Select</i>	<code>:vmap</code>	<code>:vnoremap</code>	<code>:</code>	<code>:vunmap</code>	<code>:vmapclear</code>
	<code>:vm</code>	<code>:vn</code>		<code>:vu</code>	<code>:vmapc</code>
<i>Visual</i>	<code>:xmap</code>	<code>:xnoremap</code>	<code>:</code>	<code>:xunmap</code>	<code>:xmapclear</code>
	<code>:xm</code>	<code>:xn</code>		<code>:xu</code>	<code>:xmapc</code>
<i>Operator Pending</i>	<code>:omap</code>	<code>:onoremap</code>	<code>:</code>	<code>:ounmap</code>	<code>:omapclear</code>
	<code>:om</code>	<code>:ono</code>		<code>:ou</code>	<code>:omapc</code>
<i>Insert Command Line</i>	<code>:map!</code>	<code>:noremap!</code>	<code>:</code>	<code>:unmap!</code>	<code>:mapclear!</code>
		<code>:no!</code>		<code>:unm!</code>	<code>:mapc!</code>
<i>Insert</i>	<code>:imap</code>	<code>:inoremap</code>	<code>:</code>	<code>:iunmap</code>	<code>:imapclear</code>
	<code>:im</code>	<code>:ino</code>		<code>:iu</code>	<code>:imapc</code>
<i>Command Line</i>	<code>:cmap</code>	<code>:cnoremap</code>	<code>:</code>	<code>:cunmap</code>	<code>:cmapclear</code>
	<code>:cm</code>	<code>:cno</code>		<code>:cu</code>	<code>:cmapc</code>
<i>Language Dependent</i>	<code>:lmap</code>	<code>:lnoremap</code>	<code>:</code>	<code>:lunmap</code>	<code>:lmapclear</code>
	<code>:lm</code>	<code>:ln</code>		<code>:lu</code>	<code>:lmapc</code>

The Vim Tutorial and Reference

<i>Mode</i>	<i>Commands</i>			
<i>Select</i>	<code>:smap</code>	<code>:snoremap</code>	<code>:sunmap</code>	<code>:smapclear</code>

Chapter 25: Complete Command-Mode (:) Commands

Although the *Vim* editor is superb when it comes to doing things visually, sometimes you need to use command mode. For example, command-mode commands are much easier to use in scripts. Also, a number of other specialized commands are found only in command mode.

Being expert in the command-mode means that you are a *Vim* power user with the ability to execute a number of amazing high-speed editing commands.

Advanced Command Entry

Command mode maintains a history of the command mode command you've entered. You can browse this history by going to command mode (:) and using the **<Up>** and **<Down>** keys.

You can also open the command mode history window with the **q:** or **CTRL-F** command. This window lets you use the up (**j**) and down (**k**) commands to select a command. You can edit this command using the normal Vim commands, then execute it by pressing **<Enter>**.

Actually, the key (**CTRL-F**) which opens the command history window is configurable by setting the option '**cedit**'. The default just happens to be **CTRL-F**.

The number of lines that appear for the command window is controlled by the value of the '**cmdwinheight**' ('**cwh**') option.

Editing Commands

The **:delete (:d)** command deletes a range of lines. To delete lines 1 through 5 (inclusive), for example, use the following command:

```
:1,5 delete
```

The general form of the :delete command is as follows:

```
:[range] delete [register] [count]
```

The **register** parameter specifies the text register in which to place the deleted text. This is one of the named registers (**a-z**). If you use the uppercase version of the name (**A-Z**), the text is appended to what is already in the register. If this parameter is not specified, the unnamed register is used.

The Vim Tutorial and Reference

The **count** parameter specifies the number of lines to delete (more on this later in this section).

The **range** parameter specifies the lines to use. Consider the following example (spaces added for readability):

```
:1, 3 delete
```

Figure 25-1 shows the results of this command.

```
1 A UNIX sales lady, Lenore,  
2 Enjoys work, but she likes the beach more.  
3 She found a good way  
4 To combine work and play:  
5 She sells C shells by the seashore.
```

```
:1,3 delete
```

```
1 To combine work and play:  
2 She sells C shells by the seashor  
~  
~  
~  
~  
3 fewer lines
```

Figure 25-1: `:1,3 delete`.

You have learned how to use search patterns for line specification. For example, the following command deletes starting from the first line with `hello` to the first line that contains `goodbye`.

```
:/hello/,/goodbye/ delete
```

Note: If `goodbye` comes before `hello`, the line range will be backwards, and the command will not work.

You can refine the search string specification by adding an offset. For example, `/hello/+1` specifies the line one line after the line with the word `hello` in it. Therefore, the following command results in the screen shown in Figure 25-2.

The Vim Tutorial and Reference

```
1 A UNIX sales lady, Lenore,  
2 Enjoys work, but she likes the beach more.  
3 She sells C shells by the seashore.  
~  
~  
~  
: /beach/+1, /seashore/-1 delete
```

Figure 25-2: Results of **:/beach/+1, /seashore/-1 delete**.

You can also use special shorthand operators for patterns, as follows:

- \/ Search forward for the last pattern used.
- \? Search backward for the last pattern used.
- \& Search forward for the pattern last used as substitute pattern.

You can also chain patterns. The following command, for example, finds the string first and then searches for the string second.

```
/first//second/
```

Figure 25-3 shows the result of the command:

```
:/found//work/ delete
```

```
1 A UNIX sales lady, Lenore,  
2 Enjoys work, but she likes the beach more.  
3 She found a good way  
4 She sells C shells by the seashore.  
~  
~  
:/found//work/ delete
```

Figure 25-3: **:/found//work/delete**.

You can also specify a line number on which to start. To start the search at line 7, for instance, use the following command-line specification:

```
7/first/
```

Other Ways to Specify Ranges

If you execute a : command with no count (as most users generally do), the *Vim* editor puts you into command mode and enables you to specify the range. If you give the command a count(5:, for example), the range is count lines (including the current one). The actual specification of this arrangement is that if you include a count, your line range is as follows:

The Vim Tutorial and Reference

```
:.,count - 1
```

In the example original file, for instance, if you move to the top line and then execute the following command, you get the results shown in Figure 25-4:

```
:3delete
```

```
1 To combine work and play:  
2 She sells C shells by the seashore.  
~  
~  
~  
~  
3 fewer lines
```

Figure 25-4: **3 :delete**.

Deleting with a Count

Another form of the delete command is as follows:

```
:line delete count
```

In this case, the **:delete** command goes to line (default = the current line) and then deletes count lines. If you execute the following command on the original example file, for instance, you get the results shown in Figure 25-5:

```
:3 delete 2
```

```
1 A UNIX sales lady, Lenore,  
2 Enjoys work, but she likes the beach more.  
3 She sells C shells by the seashore.  
~  
~  
~  
:3 delete 2
```

Figure 25-5: **:3 delete 2**.

Note: You can specify a line range for this type of command, but the first line is ignored and the second one is used.

Copy and Move

The **:copy** (**:co**, **:t**) command copies a set of lines from one point to another. The general form of the copy command is as follows:

```
:[range] copy address
```

If not specified, range defaults to the current line. This command copies the line in range to the line specified after address. Consider the following command, for example:

```
:1,3 copy 4
```

Executed on the original joke, you get the results shown in Figure 25-6.

```
1 A UNIX sales lady, Lenore,  
2 Enjoys work, but she likes the beach more.  
3 She found a good way  
4 To combine work and play:  
5 A UNIX sales lady, Lenore,  
6 Enjoys work, but she likes the beach more.  
7 She found a good way  
8 She sells C shells by the seashore.  
~  
3 more lines
```

Figure 25-6: **:1,3 copy 4**.

The **:move** command is much like the **:copy** command, except the lines are moved rather than copied. The following command results in what is shown in Figure 25-7:

```
:1,3 move 4
```

```
1 To combine work and play:  
2 A UNIX sales lady, Lenore,  
3 Enjoys work, but she likes the beach more.  
4 She found a good way  
5 She sells C shells by the seashore.  
~  
~  
~  
~  
3 lines moved
```

Figure 25-7: **:1,3 move 4**.

Inserting Text

Suppose that you want to insert a bunch of lines and for some reason you want to use command mode. You need to go to the line above where you want the new text to appear. In other words, you want the text to be inserted after the current line. Now start the insert by executing the :**append** (:a) command. Type the lines that you want to add and finish by typing a line that consists of just a period (.). The following example illustrates the :**append** command.

```
:% print
A UNIX sales lady, Lenore,
Enjoys work, but she likes the beach more.
    She found a good way
        To combine work and play:
She sells C shells by the seashore.
:1 append
This line is appended.
.
:% print
A UNIX sales lady, Lenore,
This line is appended.
Enjoys work, but she likes the beach more.
    She found a good way
        To combine work and play:
She sells C shells by the seashore.
```

The general form of the :**append** command is as follows:

```
:[line] append
```

The **line** is the line after which to insert the new text.

The :**insert** command also inserts text. It has a similar form:

```
:[line] insert
```

It works just like :**append** except that :**append** inserts after and :**insert** inserts before the current line.

The :**stopinsert** (:**stopi**) command stops insert mode as if you had typed <ESC>. But this command is a little hard to enter manually, since you must stop insert mode before you can type it in. It is useful however in :**autocmd** declarations. For example, if you want to exit insert mode when you enter a buffer, use the command:

```
:autocmd BufEnter * :stopinsert
```

The Vim Tutorial and Reference

The **:startreplace** (**:startr**) command starts a replace just as if you had typed an **R**. If the override (!) option is present, replacement begins at the end of line as if you typed **\$R**.

Printing with Line Numbers

You do not have to turn on the number option to print the text with line numbers. The **:#** (**:number**, **:nu**) command accomplishes the same thing as **:print** but includes line numbers:

```
:1 print
A UNIX sales lady, Lenore,
:1 #
    1 A UNIX sales lady, Lenore,
```

Printing with list Enabled

The '**list**' option causes invisible characters to be visible. The **:list** (**:l**) command lists the specified lines, assuming that this option is on:

```
:1,5 list
```

The following example shows the difference between **:print** and **:list**:

```
:100,111 print
open_db(void)
{
    if (proto_db == NULL) {
        proto_db = gdbname_open(proto_db_name, 512, GDBM_READER ,
                                0666, NULL);
    if (proto_db == NULL) {
        fprintf(stderr, "Error: Could not open database %s\n",
                proto_db_name);
        exit (8);
    }
}
:100,111 list
open_db(void)$
{$
    if (proto_db == NULL) {$
^Iproto_db = gdbname_open(proto_db_name, 512, GDBM_READER, $
^I^I^I^I0666, NULL);$
^Iif (proto_db == NULL) {$
```

The Vim Tutorial and Reference

```
^I      fprintf(stderr, "Error: Could not open database %s\n",
$  
^I      ^I^I proto_db_name);$  
^I      exit (8);$  
^I}{$  
      }$  
}{$
```

Print the Text and Then Some

The **:z** command prints a range of lines (the current one being the default) and the lines surrounding them. For example, the following command prints line 100 and then a screen full of data:

```
:100 z
```

The **:z** command takes a count of the number of extra lines to list. For example, the following command lists line 100 and three additional lines:

```
:100 z 3
```

The **:z** command can be followed by a code indicating how much to display. The following table lists the codes:

Code	Listing Start	Listing End	New Current Line
+	Current line	One screen forward	One screen forward
-	One screen back	Current line	Current line
^	Two screens back	One screen back	One screen back
.	One-half screen back	One-half screen forward	One-half screen forward
=	One-half screen back	One-half screen forward	Current line

Substitute

The format of the basic substitute command is as follows:

```
:[range] s /from/to/[flags] [count]
```

(You can use **:substitute** for **:s**, but in practice this is almost never done.)

Note: This example uses a slash (/) to separate the patterns. Actually you can use almost any character that does not appear in the patterns. The following, for example, is perfectly valid:

```
:s +from+to+
```

This can prove extremely useful when you are dealing with patterns that contain slashes, such as filenames:

```
:1,$ s +/home/user+/apps/product+
```

The Vim Tutorial and Reference

Delimiters can be any character except letters, digits, backslash, double quote, or vertical bar.

The *Vim* editor uses a special set of magic characters to represent special things. For example, star (*) stands for "repeat 0 or more times." If you set the '**nomagic**' option, however, the magic meanings of some of these characters are turned off. (For a complete list of the magic characters and how the '**nomagic**' option affects them, see *Chapter 19: Advanced Searching Using Regular Expressions*.)

The **:smagic** (**:sm**) command performs a substitute but assumes that the '**magic**' option is set during the command. For example, start in command mode with a one-line file. You start by printing the entire file:

```
:%print  
Test aaa* aa* a*
```

Now set the '**magic**' option and perform a substitution. The **p** flag tells the editor to print the line it changed:

```
:set magic  
:1 s /a*/b/p  
bTest aaa* aa* a*
```

This command made only one change at the beginning of the line. So why did it change **Test** to **bTest** when there is no **a** around? The answer is that the magic character star (*) matches *zero* or more times. **Test** begins with zero **a**'s.

But why did it make only one change? Because the **:substitute** command changes only the first occurrence unless the **g** flag is present. Now undo the change and try again:

```
:undo  
:1 s /a*/b/pg  
bTbebsbtb b*b b*b b*b
```

This time you got what you wanted. Now try it again with the '**nomagic**' option set:

```
:undo  
:set nomagic  
:1 s /a*/b/pg  
Test aab ab b
```

Without '**magic**', a star (*) is just a star. It is substituted directly. The **:smagic** command forces magic on the star (*) and other characters while the substitution is being made, resulting in the following:

The Vim Tutorial and Reference

```
:undo  
:1 smagic /a*/b/pg  
bTbebsbtb b*b b*b b*b
```

The **:snomagic** (**:sno**) forces '**magic**' off.

```
:undo  
:set magic  
:1 snomagic /a*/b/pg  
Test aab ab b
```

The **&** command repeats the substitution. This enables you to keep your old from and to strings, but also to supply a different range or flags. The general form of this command is:

```
:[range]& [flags] [count]
```

For example:

```
:1 s /a\+/b/p  
Test b* aa* a*
```

The command changes the first occurrence of from on the line. You want the entire line, so you repeat the substitution with the **g** option:

```
:&g
```

Of course, this does not print (because the new flags--in this case **g**--replaces the flags and you did not specify **p** or, more specifically, **pg**). Take a look at the result:

```
:1 print  
Test b* b* b*
```

This is what you wanted.

The **:&** command and the **:substitute** command with no from or to specified acts the same. The normal-mode **&** command repeats the last **:substitute** command. If you were to execute the following command, for instance, you would change the first manager on line 5 to an idiot:

```
:5 s /manager/idiot/
```

Now if you enter normal mode (through the **:vi** command) and execute an **&** command, the next manager on this line would change as well. If you were to move down to another line and execute an **&** command, you would change that line as well. If you give **&** a count, it will work on that many lines.

The Vim Tutorial and Reference

The `:~` command acts just like the `&r` command, except that it uses as from the last search pattern (used for a `/` or `?` search) rather than the last `:substitute from` string.

The general form of this command is:

```
:[range]~ [flags] [count]
```

Substitute flags

The flags or the `:substitute` and other similar command are:

- #** Print the line number and line after substitution.
- &** Use the previous flags again. This must be the first flag in the list.
- c** Confirm each substitution.
- e** Do not output an error message if no text matches.
- g** Replace every occurrence on the line instead of just the first one.
- i** Ignore case.
- I** Do not ignore case even if the options tell you to.
- l** Print the line in 'list' mode after substitution.
- n** Don't actually do the work, just report the number of changes that would be made if you did.
- p** Print the line after substitution.
- r** If the search pattern is empty, use the previous one.

Making g the Default

Generally the `:substitute` command changes only the first occurrence of the word unless you use the '`g`' option. To make the '`g`' option the default set the '`gdefault`' ('`gd`') option.

```
:set gdefault
```

Note:This can break some scripts you may use.

Global Changes

The command-mode commands covered so far have one limitation: They work only on a contiguous set of lines. Suppose, however, that you want to change just the lines that contain a certain pattern. In such a case, you need the **:global (:g)** command.

The general form of this command is:

```
:[range] global /pattern/ command
```

This tells *Vim* to perform the given command on all lines that contain the pattern in the specified range.

To print all the lines within a file that contain the word **Professor**, for instance, use the following command:

```
:% global /Professor/ print
Professor: Yes.
Professor: You mean it's not supposed to do
Professor: Well there was no Computer Center
Professors of mathematics will prove the
```

The **:global! (:g!)** command applies the command to all the lines that do not match the given pattern, as will the **:vglobal** command.

Commands for Programs

The following sections describe some commands for programs.

Include File Searches

The **:ijump (:ij)** command searches for the given pattern and jumps to the first occurrence of the word in the given range. It searches not only the current file, but all files brought in by #include directives. The general form of this command is as follows:

```
:[range] ijump [count] [/]pattern[]
```

If a count is given, jump to the *count* occurrence of the pattern. If the *pattern* is enclosed in slashes it must be whole word. With slashes it is a regular expression. Consider the file *hello.c*, for example:

The Vim Tutorial and Reference

```
#include <stdio.h>
int main() {
    printf("Hello World\n");
    return (0);
}
```

The following command goes to the first line that contains define EOF:

```
:ijump /define\s*EOF/
```

In this case, it is in the include file *stdio.h*. The **:ilist** (**:il**) command acts like **:ijump**, except it lists the lines instead of jumping to them:

```
:ilist EOF
/usr/include/libio.h
1: 84 #ifndef EOF
2: 85 # define EOF (-1)
3: 327 && __underflow (_fp) == EOF ? EOF \
/usr/include/stdio.h
4: 83 #ifndef EOF
5: 84 # define EOF (-1)
6: 408 null term.), or -1 on error or EOF. */
/usr/include/bits/stdio.h
7: 138 if (__c == EOF) \
8: 157 if ((*__ptr++, __stream) == EOF) \
```

The **:isearch** (**:is**) command is like **:ilist**, except that the first occurrence is listed:

```
:isearch EOF
#ifndef EOF
```

Finally, the command **:isplit** (**:isp**) works like a **:split** and a **:ijump**.

Jumping to Macro Definitions

You learned how to use the command [**CTRL-D**] to jump to the definition of the macro under the cursor. The **:djump** (**:dj**) command accomplishes the same thing for the macro named name:

```
:djump name
```

To jump to the macro MAX, for example, use this command:

```
:djump MAX
```

You do not have to know the full name of the macro to find its definition. If you know only part of a name, you can perform a search for a partial string by enclosing the name in slashes, as follows:

The Vim Tutorial and Reference

```
:djump /MAX/
```

This command finds the first definition of the macro with the word MAX in it. You can give the `:djump` command a range argument that restricts the search to the given range:

```
:50,100 djump /MAX/
```

This command finds the first definition of any macro containing the word MAX in lines 50 through 100.

If you do not want the first definition, but the second, you can add a count to the command. To find the second definition of MAX, for instance, use this command:

```
:djump 2 MAX
```

Split the Window and Go to a Macro Definition

The `:dsplit (:dsp)` command is shorthand for `:split` and `:djump`:

```
:[range] dsplit [count] [/]pattern[/]
```

Listing the Macros

The `:dlist (:dl)` command works just like `:djump`, except that instead of moving to the macro definition, the command just lists all the definitions that match:

```
:dlist EOF  
/usr/include/libio.h  
1: 85 # define EOF (-1)  
/usr/include/stdio.h  
2: 84 # define EOF (-1)
```

Listing the First Definition

The `:dsearch (:ds)` command works just like `:dlist`, except that it displays only the first definition:

```
:dsearch EOF  
# define EOF (-1)
```

Override Option (!)

The `:ilist`, `:ijump`, `:djump`, `:dlist`, and `:dsearch` commands take an override option `!`. If the `!` is present, definitions within comments are found as well.

Directory Manipulation

To change the current working directory, use the following command:

```
:cd dir
```

(Also known as :chdir, :chd.)

This command acts just like the system *cd* command. On UNIX, it changes the current working directory to the given directory. If no directory is specified, it goes to the user's home directory.

On Microsoft Windows, it goes to the indicated directory. If no directory is specified, it prints the current working directory. The following command changes the directory to the previous path:

```
:cd -
```

In other words, it does a *cd* to the last directory you used as the current working directory.

To find out which directory *Vim* is currently using, use the :pwd (:pw) command:

```
:pwd
```

Start deep in the directory tree, for instance:

```
:pwd  
/mnt/sabina/sdo/writing/book/vim/book/11
```

You are working on a UNIX system, so go to your \$HOME directory:

```
:cd  
:pwd  
/home/sdo
```

Jump into another directory:

```
:cd tmp  
:pwd  
/home/sdo/tmp
```

Return to the previous directory:

```
:cd -  
:pwd  
/home/sdo
```

Return to the previous directory before this one:

The Vim Tutorial and Reference

```
:cd -
:pwd
/home/sdo/tmp
```

The **:cd** command is global. In other words, the current directory is changed for everyone. If you want to change it for just a single window use the **:lcd** (**:lc**, **:lchdir**, **:lch**) command.

The '**cdpath**' ('**cd**') option can be used to give Vim a set of directories to search when doing a **:cd**. For example, if you enter the command:

```
:set cdpath=/usr/src,/new/src,
:cd tools
```

Then Vim will attempt to go to the first directory it finds in the list:

```
/usr/src/tools
/usr/new/tools
./tools
```

Note: The current directory is just the empty string: **,** (**<comma>****<comma>**).

Also if enable the '**auctochdir**' ('**acd**') option then the directory will automatically change when you change files to the directory of the file you are currently editing.

Current File

The **:file** (**:f**) command prints out the current file and line information:

```
:file
```

If you want to change the name of what *Vim* thinks is the filename, use this command:

```
:file name
```

Suppose, for example, that you start editing a file called *complete.txt*. You get this file just right, so you write it out using the **:write** (**:w**) command.

Now you want to shorten the file and write it out as *summary.txt*. So now you execute this command:

```
:file summary.txt
```

Now when you continue to edit, any changes are saved to *summary.txt*.

Take a look at how this works. You start by editing the file *star.txt*.

The Vim Tutorial and Reference

```
:file  
"star.txt" line 1 of 1 --100%-- col 1
```

The **:write** command with no arguments writes the file to the current filename (in this case, *star.txt*).

```
:write  
"star.txt" 1 line, 18 characters written
```

Now you want to change the filename to *new.txt*. The editor tells you that this is a new filename.

```
:file new.txt  
"new.txt" [Not edited] line 1 of 1 -100%- col 1
```

The **:write** command is used to write the file. In this case, the current filename differs; it is *new.txt*.

```
:write  
"new.txt" [New File] 1 line, 18 chars written
```

The following command prints the current line number:

```
:=
```

For example:

```
:=  
line 1
```

Advanced :write Commands

The **:write** command writes the buffer (or a selected range of lines) to a file. It has some additional options. The following command, for example, appends the contents of the file you are editing to the file named *collect.txt*:

```
:write >> collect.txt
```

If the *collect.txt* file does not exist, this command aborts with an error message. If you want to "append" to the file even if it does not exist, use the force (!) option:

```
:write! >> collect.txt
```

The **:write** command can not only write to a file, but it can also be used to pipe the file to another program. On Linux or UNIX, for instance, you can send the file to a printer by using the following command:

```
:write !lpr
```

Warning: The following two commands are different; the difference being only the spacing:

```
:write! lpr  
:write !lpr
```

The first writes to the file named *lpr* with the force option in place. The second sends the output to the command *lpr*.

Updating Files

The **:update** (**:up**) command acts just like the **:write** command, with one exception: If the buffer is not modified, the command does nothing.

Reading Files

The **:read** (**:r**) command reads in a file. The general form of this command is as follows:

```
:[line] read file
```

The preceding command reads the file in and inserts it just after line. If no file is specified, the current file is used. If no line is supplied, the current line is used.

Like **:write**, the **:read** command can use a command rather than a file. To read the output of a command and insert it after the current line, use the following command:

```
:[line] read !command
```

Register Execution

Chapter 2: Editing a Little Faster showed you how to record macros in registers. If you want to use these macros in command mode, you can execute the contents of a register with the command **:@** (**:***).

```
:[line]@register
```

This command moves the cursor to the specified line, and then executes the register. This means that the following command executes the previous command line:

```
:@:
```

To execute the previous **:@register** command, use this command:

```
:[line]@@
```

Simple Edits

The following sections describe simple edits.

Shifting

The **:>** command shifts lines to the right. The **:<** command shifts lines to the left. The following command, for example, shifts lines 5 through 10 to the right:

```
:5, 10 >
```

Changing Text

The **:change (:c)** command acts just like the **:delete** command, except that it performs an **:insert** as well.

Entering Insert Mode

The **:startinsert (:star)** command starts insert mode as if you were in normal mode and were to press **i**.

Joining Lines

The **:join (:j)** command joins a bunch of lines (specified by the range parameter) together into one line. Spaces are added to separate the lines. If you do not want the added spaces, use the **:join!** command.

Yanking Text

The **:yank (:y)** command yanks the specified lines into the register:

```
:[range] yank [register]
```

If no register is specified, the unnamed register is used.

Putting Text

The **:put** command puts the contents of a register after the indicated line. To dump the contents of register a after line 5, for example, use the following command:

```
:5put a
```

If you want to put the text before the line, use this command:

```
:5put! a
```

Undo/Redo

The **:undo** (**:u**) command undoes a change just like the **u** command does. The **:redo** command redoing a change like **CTRL-R** does.

Marks

To mark the beginning of the line, use the **:mark** (**:ma**) command.

```
:mark register
```

command. If a line is specified, that line will be marked. The **:k** command does the same thing, with the exception that you don't have to put a space in front of the register name.

The following two commands are equivalent:

```
:100 mark x  
:100 kx
```

Miscellaneous Commands

The following sections describe some miscellaneous commands you can use.

The :preserve Command

The **:preserve** (**:pre**) command writes out the entire file to the "swap" file. This makes it possible to recover a crashed editing session without the original file. (If you do not use this command, you need both the swap file and the original to perform recovery.) See *Chapter 14: File Recovery and Command-Line Arguments*, for information on recovering crashed sessions.

The Shell Commands

To execute a single shell command, use the following *Vim* command (where *cmd* is the system command to execute):

```
: !cmd
```

To find the current date, for instance, use this command:

```
: !date
```

The following command repeats the last shell command you executed:

The Vim Tutorial and Reference

```
:!!
```

Finally, the following command suspends *Vim* and goes to the command prompt:

```
:shell
```

You can now enter as many system commands as you want. After you have finished, you can return to *Vim* with the **exit** command.

Shell Configuration

The following several options control the actual execution of a command.

' shell '	The name of the shell (command processor).
' sh '	
' shellcmdflag '	Flag that comes after the shell.
' shcf '	
' shellquote '	The quote characters around the command.
' shq '	
' shellxquote '	The quote characters for the command and the redirection.
' sxq '	
' shellpipe '	String to make a pipe.
' sp '	
' shellredir '	String to redirect the output.
' srr '	
' shellslash '	Use forward slashes in filenames (MS-DOS only).
' ssl '	
' quoteescape '	Defines the character to use for quoting the escape (/) character.
' qe '	
' shelltmp '	Use temporary files if set. Otherwise use pipes.
' stmp '	

Command History

The :**history** (:**his**) command prints out the current command-mode command history:

The Vim Tutorial and Reference

```
:history
# cmd history
2 1 print
3 5
4 7 print
5 . print
> 6 history
```

The *Vim* editor maintains a set of histories for various commands. A code identifies each of these:

Code History Type

c	cmd	:	Command-line history (command-mode commands)
s	search	/	Search strings (See <i>Chapter 3: Searching</i>)
e	expr	=	Expression register history
i	input	@	Input line history (data typed in response to an : input operator)
a	all		All histories

Therefore, to get a list of all the various history buffers, use the **:history all** command:

```
:history all
# cmd history
2 1 print
35
4 7 print
5 . print
6 history
7 history all
# search history
1 human
2 uni
3 comp
4 Seem
5 \<At\>
# expr history
1 55
2 2*88
3 5+99
# input history
Press Return or enter command to continue
```

The general form of the **:history** command is as follows:

```
:history code first , last
```

The Vim Tutorial and Reference

If no first and last are specified, the whole history is listed. The first parameter defaults to the first entry in the history, and the last defaults to the last. Negative numbers indicate an offset from the end of the history. For example, **-2** indicates the next-to-last history entry. The following command, for example, list history entries 1 through 5 for command-mode commands:

```
:history c 1,5
```

And, this next command lists the last 5 search strings:

```
:history s -5,
```

Setting the Number of Remembered Commands

The '**history**' ('**hi**') option controls how many commands to remember for command mode (: mode) commands. To increase the number of commands to remember (to 50, for instance), use this command:

```
:set history=50
```

Viewing Previous Error Messages

The *Vim* editor keeps track of the last few error and information messages displayed on the last line of the screen. To view the message history, use the **:messages** (**:mes**) command:

```
:messages
"../joke.txt" 6092 lines, 174700 characters
Entering Ex mode. Type "visual" to go to Normal mode.
search hit BOTTOM, continuing at TOP
Not an editor command: xxxxx
search hit BOTTOM, continuing at TOP
search hit BOTTOM, continuing at TOP
Pattern not found: badbad
Not an editor command: :^H
Invalid address
```

Redirecting the Output

The **:redir** (**:redi**) command causes all output messages to be copied to the file as well as to appear on the screen:

```
:redir > file
```

To end the copying, execute the following command:

```
:redir END
```

The Vim Tutorial and Reference

This command proves useful for saving debugging information or messages for inclusion in a book.

You can also use the **:redir** command to append to a file by using this command:

```
:redir >> file
```

Executing a :normal Command

The **:normal** (**:norm**) command executes a normal-mode command. The following command, for instance, changes the word where the cursor is located to the word **DONE**:

```
:normal cwDONE<Esc>
```

The group of commands is treated as one for the purposes of undo/redo. The command should be a complete command. If you leave *Vim* hanging (suppose that you executed a command **cwDone**, for instance), the display will not update until the command is complete. If you specify the '!' option, mappings will not be done on the command.

Getting Out

The **:exit** (**:exi**, **:xit**, **:x**) command writes the current file and closes the window:

```
:exit
```

When the last window is closed, the editor stops. If the override flag (!) is given, an attempt will be made to write the file even if it is marked read-only. You can also specify a filename on the command line. The data will be written to this file before exiting. The following command, for example, saves the current file in *save-it.txt* and exits.

```
:exit save-it.txt
```

If you want to save only a portion of the file, you can specify a range of lines to write. To save only the first 100 lines of a file and exit, for example, use this command:

```
:1,100 exit save-it.txt
```

Write and Quit

The following command does the same thing that **:exit** does, except it always writes the file:

```
:range wq! file
```

The **:exit** command writes only if the file has been changed.

Advanced Hardcopy

The **:hardcopy** (**:ha**) command will print the file to the line printer. You can also send the printed output to a file with the command:

```
:[range]hardcopy[!] > file
```

Warning: The command always overwrites the file without warning. The override option (!) is used only to automatically select the default printer when using Microsoft Windows.

On Linux and UNIX the output file is written in Postscript format. On Microsoft Windows, it's written in the printer's native format using the "print to file" feature of Microsoft Windows.

Normally the file is printed to the default printer. To change the name of the printer, set the '**printerdev**' ('**pdev**') option. For example, to go to the printer named "Fred", execute the command:

```
:set printerdev=Fred
```

The '**printfont**' ('**pfn**') tells the printer what font to use. The actual font name is the same as for the '**guifont**' option. (See page 482.)

The '**encoding**' ('**enc**') option *Vim* what encoding to use for the text in the file. If you wish to use a different encoding for printing set the '**printencoding**' ('**penc**') option.

If you are printing CJK (Korean), the '**printmbfont**' ('**pmbfn**') option controls what fonts are used for printing. Also the '**printmbcharset**' ('**pmbcs**') option controls the character set to be used. Unfortunately the Korean language is beyond the scope of this book.

The format of the header is controlled by the '**printheader**' ('**pheader**') expression. The format of this option is exactly the same as that of the '**statusline**' option (see page 482).

Finally the '**printexpr**' ('**pexpr**') option sets the expression to use for printing. The expression is passed in the variables `v:fname_in` as the file to be printed. This file must be deleted by the expression.

The Vim Tutorial and Reference

Also the variable `v:cmdarg` contains any arguments to the `:hardcopy` command.

Chapter 26: Advanced GUI Commands

The *Vim* editor is highly configurable. This chapter shows you how to customize the GUI. Among other things, you can configure the following in *Vim*:

- The size and location of the window
- The display of menus and toolbars
- How the mouse is used
- The commands in the menu
- The buttons on the toolbar
- The items in the pop-up menu

The remainder of this chapter introduces you to the commands that enable you to customize all these features.

Switching to the GUI Mode

Suppose you are editing in a terminal window and want to switch to the GUI mode. To do so, use the `:gui` (`:gu`, `:gvim`, `:gv`) command:

```
:gui
```

Window Size and Position

When you first start *gvim* (GUI Vim), the window is positioned by the windowing system. The size of the window on UNIX is set to the size of the terminal window that started the editor. In other words, if you have a 24×80 xterm window and start *gvim*, you get a 24×80 editing window. If you have a larger window, say 50×132 , you get a 50×132 editing window.

On UNIX you can tell *gvim* to start at a given location and size by using the `-geometry` flag. The general format of this option is as follows:

```
-geometry {width}x{height}-{x_offset}+{y_offset}
```

The *width* and *height* options specify the width and height of the window (in characters). The *x-offset* and *y-offset* tell the X Windows System where to put the window.

The Vim Tutorial and Reference

The *x-offset* specifies the number of pixels between the left side of the screen and the right side of the window. If the *x-offset* specification is negative, it specifies the distance between the left edge of the editor and the right side of the screen.

Similarly, the *y-offset* specifies the top margin, or if negative, the bottom margin.

Thus, the **-geometry +0+0** option puts the window in the upper-left corner, whereas **-geometry -0-0** specifies the lower-right corner. The *width* and *height* parameters specify how big the editing window is to be in lines and columns. To have a 24-by-80 editing window, for example, use the option **-geometry 80x24**.

Figure 26-1 shows how these options work.

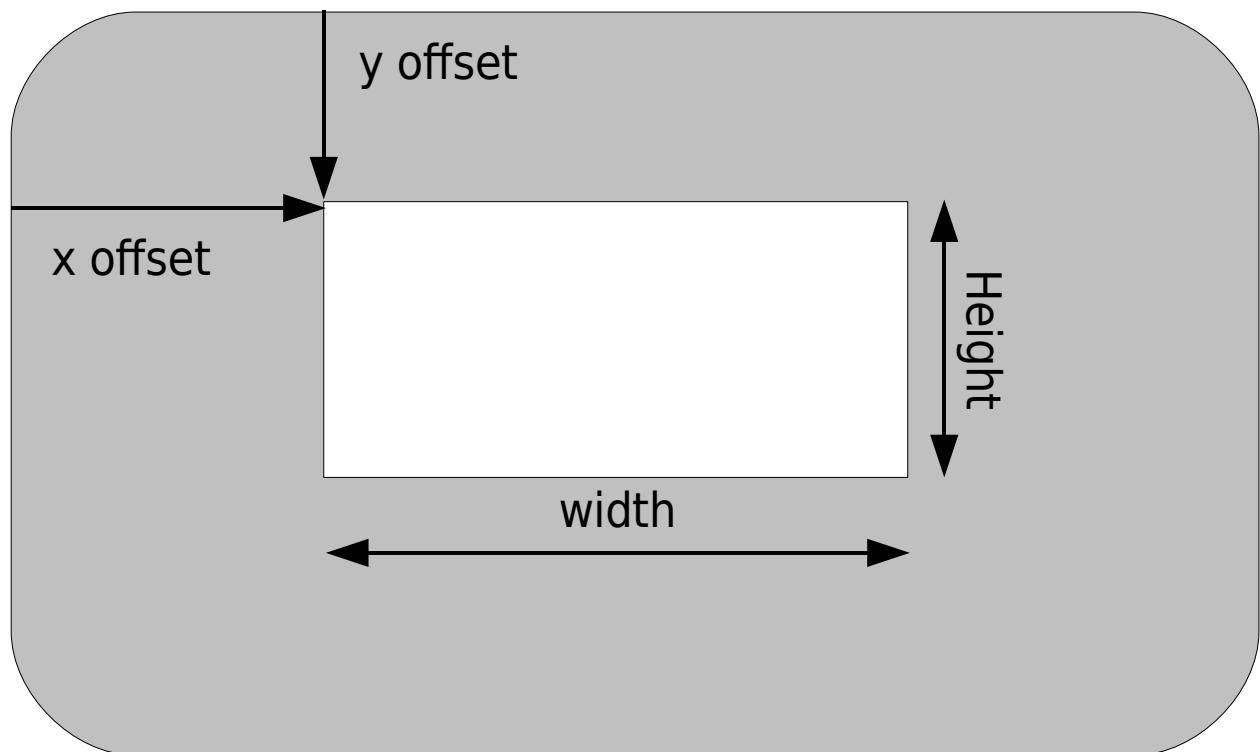


Figure 26-1: **-geometry** option.

Microsoft Windows Size and Position Command-Line Specification

The Microsoft Windows version of *gvim* starts with an editing window of 80×25 . The *gvim* editor uses the standard Microsoft Windows command-line option to specify an initial size and position. Because Microsoft Windows does not have a standard option for this value, *gvim* does not have one either.

Moving the Window

The **:winpos** (**:winp**) command displays the current location (in pixels) of the upper-left corner of the window:

```
:winpos
```

If you want to move the window, you can use this command:

```
:winpos X Y
```

To position the screen 30 pixels down and 20 pixels over from the left, for instance, use the following command:

```
:winpos 20 30
```

Window Size

The following command displays the number of lines in the editing window:

```
:set lines?
```

To change this number, use this command:

```
:set lines=lines
```

Lines is the number of lines you want in the new editing window.

To change the number of columns on the screen, use the '**columns**' ('**co**') option:

```
:set columns=columns
```

The :winsize Command

Older versions of *Vim* used a **:winsize** (**:wi**) command. This command is deprecated because the **:set lines** and **:set columns** commands have superseded it.

The 'guioptions'

You can control a number of GUI-based features with the '**guioptions**' ('**go**') option. The general form this command is as follows:

```
:set guioptions=options
```

Options is a set of letters, one per option.

The following options are defined:

- | | |
|---------------------|---|
| a Autoselect | When set, if you select text in the visual mode, <i>Vim</i> tries to put the selected text on the system's global clipboard. This means that you can select text in one <i>Vim</i> session and paste it in another using the " *p " command. Without this option you must use the " *y " command to copy data to the clipboard.

It also means that the selected text has been put on the global clipboard and is available to other applications. On UNIX, for example, this means that you can select text in visual mode, and then paste it into an <i>xterm</i> window using the middle mouse button. |
|
 | |
| | If you are using Microsoft Windows, any text selected in visual mode is automatically placed on the clipboard. (Just as the Copy ^c menu item does in most other applications.) This means that you can select the text in <i>Vim</i> and paste it into a Microsoft Word document. |
|
 | |
| f Foreground | On UNIX, the <i>gvim</i> command executes a fork() command so that the editor can run in the background. Setting this flag prevents this, which is useful if you are writing a script that needs to run the <i>gvim</i> command to let the user edit a file, and that needs to wait until the editing is done. (The -f command-line option accomplishes the same thing.) The f flag also proves useful if you are trying to debug the program. |

Note: You must set this in the initialization file (because by the time you can set it from the edit window, it is too late).

The Vim Tutorial and Reference

i Icon

If set, *gvim* displays an attractive icon when the editor is minimized if you are using a window manager which displays programs on the background.. If not present, the program just displays the name of the file being edited with no icon, (see Figure 26-2). (Note: This works only on certain types of window managers.)



`:set guiopt+=i :set guiopt-=i`

Figure 26-2: *i* option

m Menu

Display menu bar (see Figure 26-3)

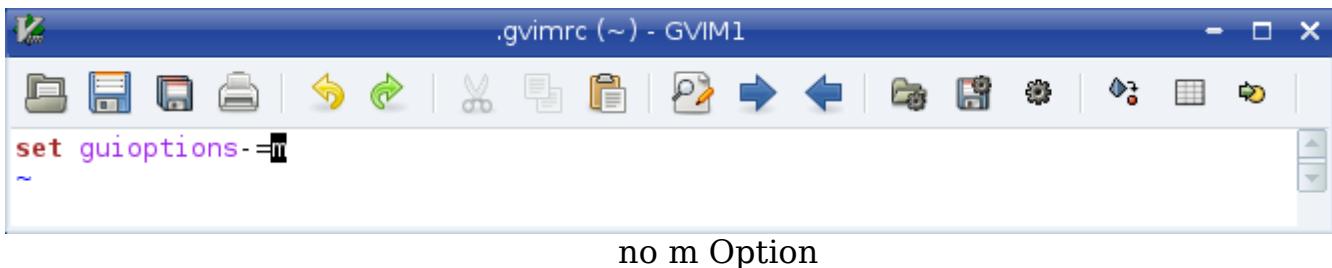
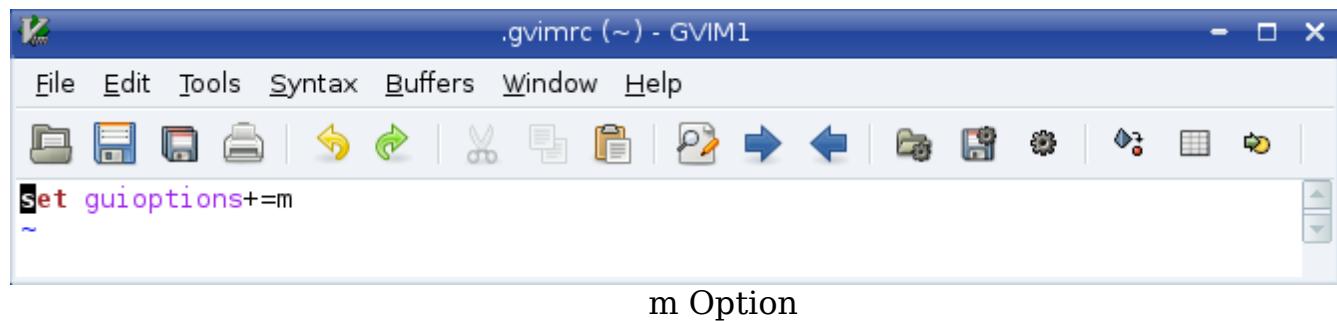


Figure 26-3: *m* option.

M No menu

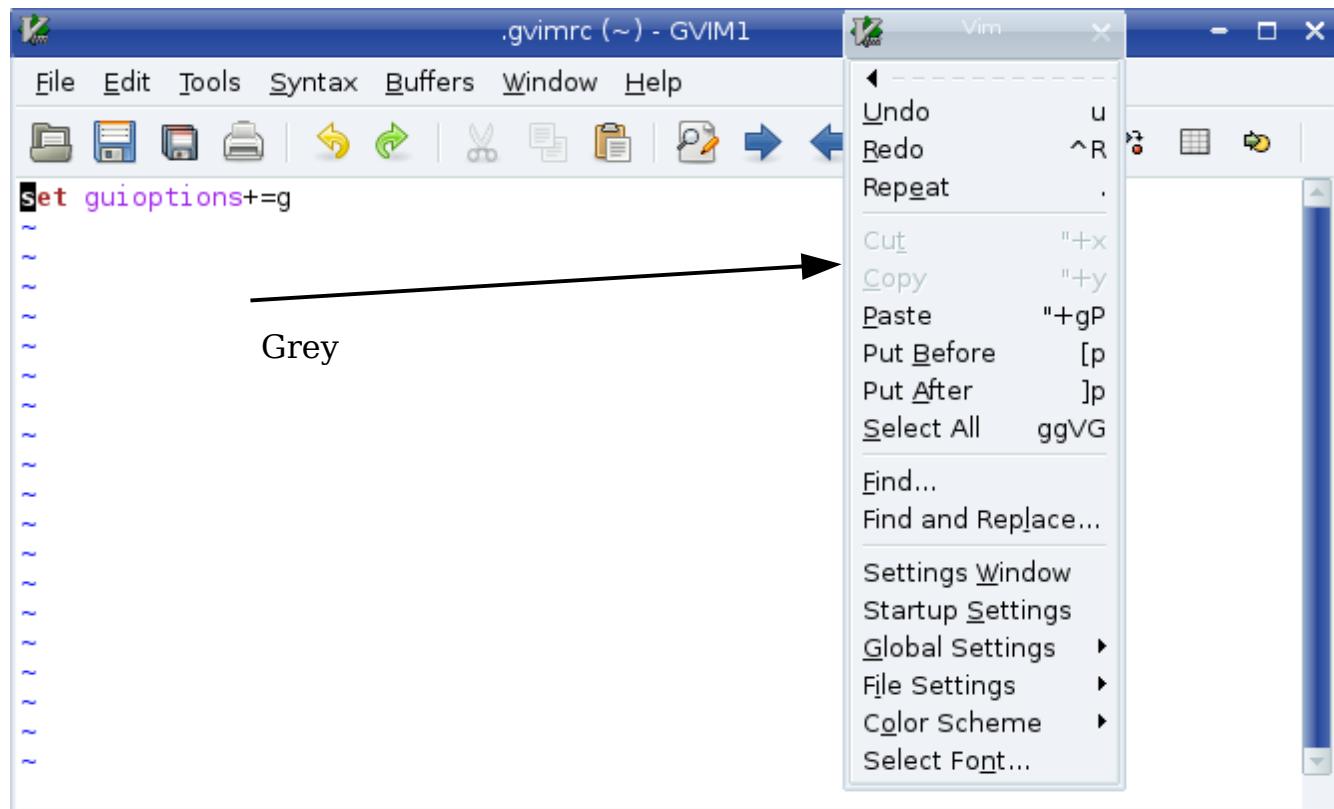
If this option is present during initialization, the system menu definition file *\$VIMRUNTIME/menu.vim* is not read in.

The Vim Tutorial and Reference

Note: This option must be set in the `.vimrc` file. (By the time `.gvimrc` is read, it is too late.)

`g` Gray

Turn menu items that cannot be used gray. If not present, these items are removed from the menu (see Figure 26-4).



The Vim Tutorial and Reference

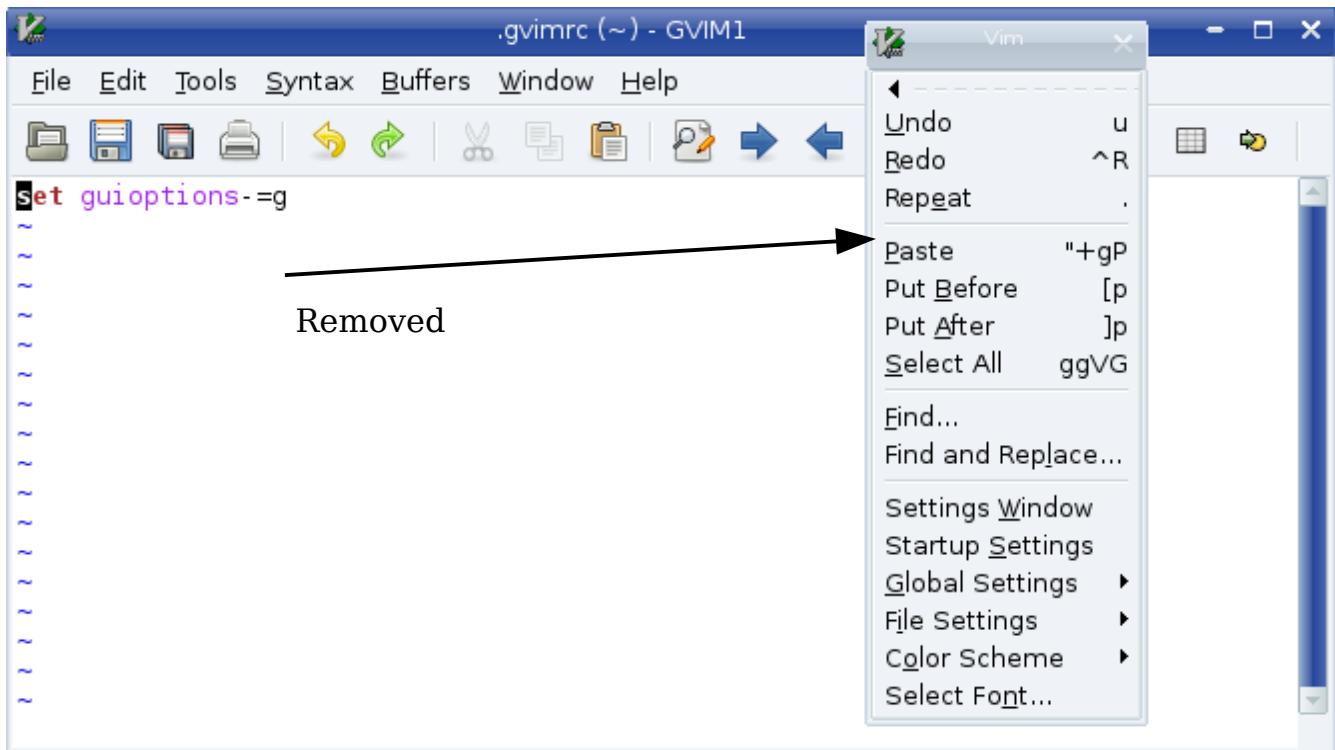
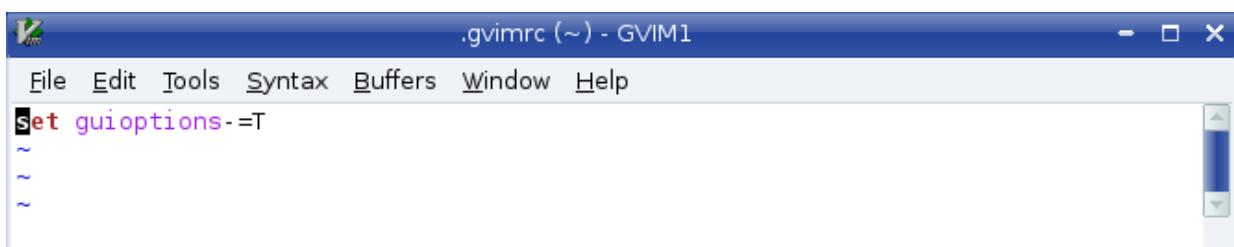
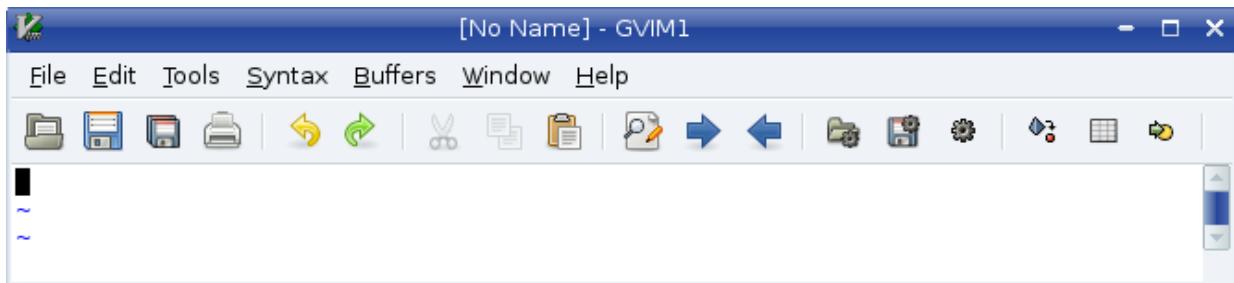


Figure 26-4: **g** option.

t Tear off Enable tear off menus.

T Toolbar Include toolbar (see Figure 26-5)



The Vim Tutorial and Reference

Figure 26-5: **t** option

- r Right scrollbar Put a scrollbar on the right (see Figure 26-6).
- l Left scrollbar Put a scrollbar on the left (see Figure 26-6).
- b Bottom scrollbar Put a scrollbar on the bottom (see Figure 26-6).

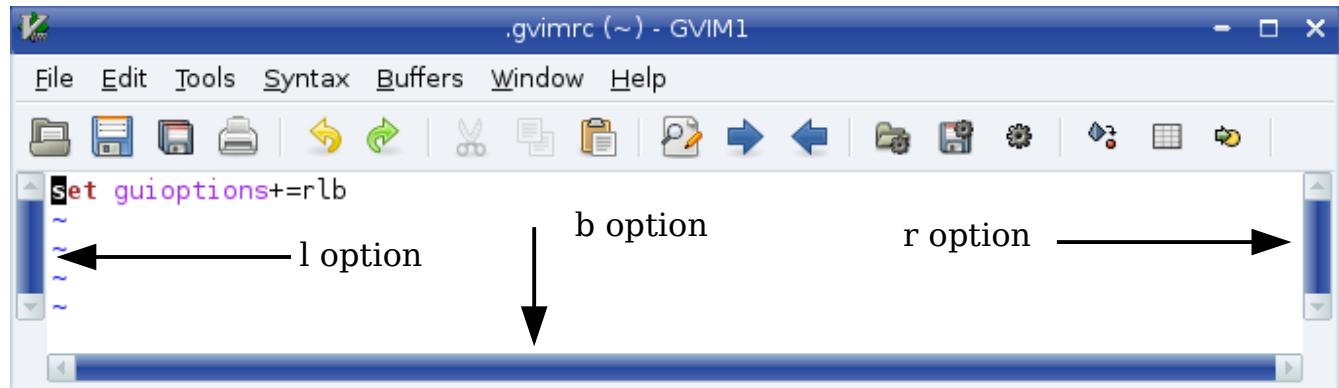


Figure 26-6: Scrollbars.

- v Vertical dialog boxes

Use vertical alignment for dialog boxes (see Figure 26-7).

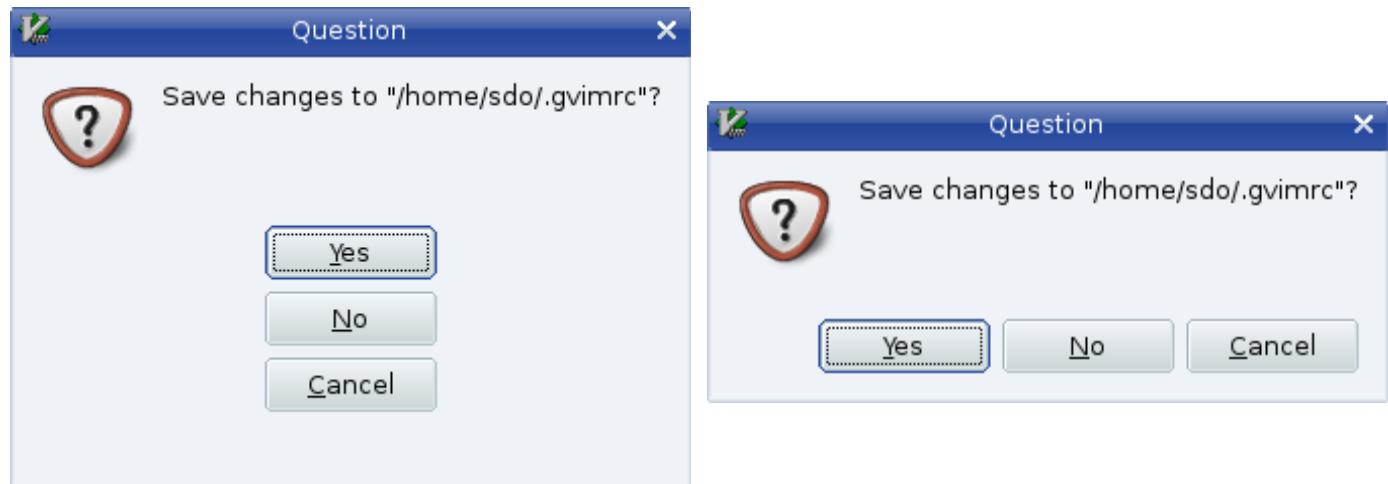


Figure 26-7: **v** option.

The Vim Tutorial and Reference

p Pointer callback fix

This option is designed to fix some problems that might occur with some X11 window managers. It causes the program to use pointer callbacks. You must set this in the `.gvimrc` file.

Changing the Toolbar

The '`toolbar`' ('`tb`') option controls the appearance of the toolbar. It is a set of values:

icon Display toolbar icons

text Display text

tooltips When the cursor hovers over an icon, display a ToolTip.

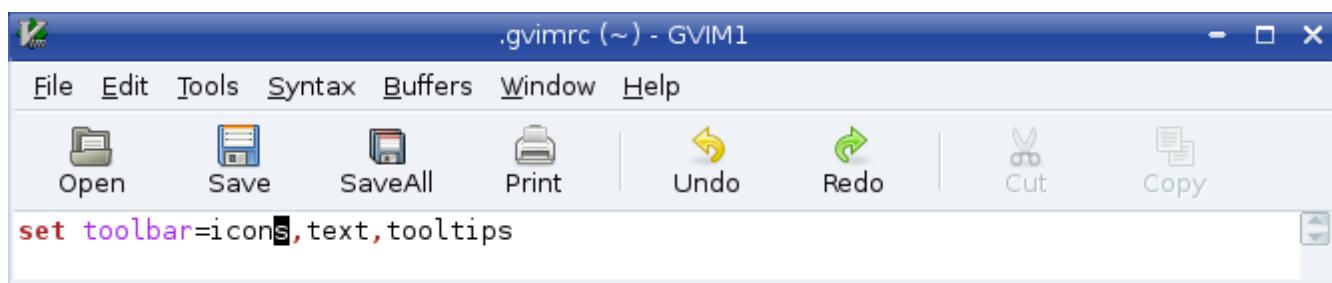
The default displays ToolTips and icons:

```
:set toolbar=icons,tooltips
```

Figure 26-8 shows how this option affects the screen.



```
:set toolbar=icons,tooltips
```



```
:set toolbar=icons,text,tooltips
```

The Vim Tutorial and Reference

```
:set toolbar=text,tooltips
```



Figure 26-8: The '**toolbar**' option.

Note: To turn off the toolbar, you cannot set this option to the empty string. Instead use the following command:

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

You can also customize the size of the icons using the '**toolbariconsize**' ('**tbis**') option. (This only works if you are using the GTK+2 gui.) The value of this option are **tiny**, **small**, **medium**, and **large**.

Customizing the Icon

If you are editing in a terminal window, some terminals enable you to change the title of their window and their icon. If you want *Vim* to try to change the title to the name of the file being edited, set this option:

```
:set title
```

Sometimes the name of the file with its full path name is longer than the room you have for the title. You can change the amount of space used for the filename with the following command:

```
:set titlelen=85
```

In this case, the title text can consume 85% of the title bar. For example:

```
:set titlelen=45
```

Figure 26-9 shows how '**titlelen**' can affect the display.

The Vim Tutorial and Reference

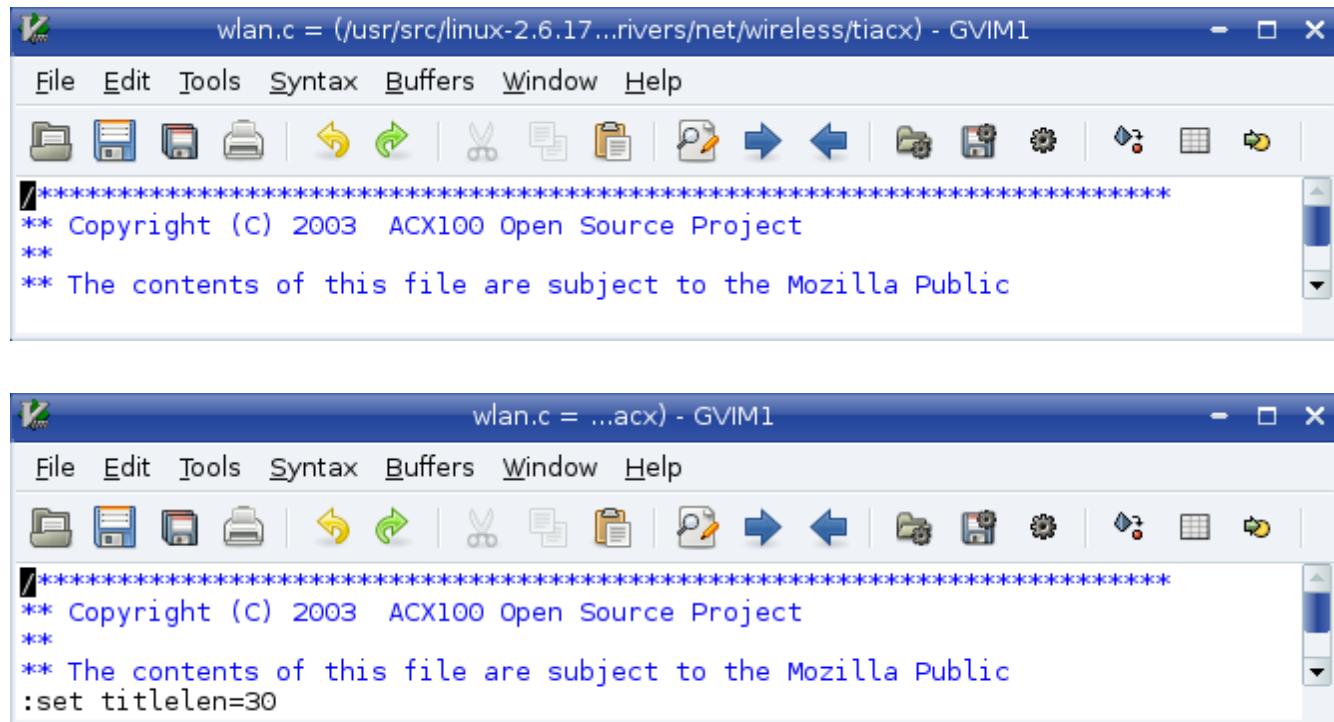


Figure 26-9: '**titlelen**' option.

If you do not like what *Vim* selects for your title, you can change it by setting the following:

```
:set titlestring>Hello\ World!
```

When you exit *Vim*, it tries to restore the old title. If it cannot restore it (because it is impossible to remember it), the editor sets the title to the string specified by the '**titleold**' option. For example:

```
:set titleold=vim\ was\ here!
```

When the window is iconified, the icon option tells *Vim* whether to attempt to put the name of the file in the icon title. If this option is set, *Vim* attempts to change the icon text.

If you do not like the *Vim* default, you can set the '**iconstring**' option and that text will be used for the icon string.

The '**icon**' option, if set causes the string under the icon to contain the name of the file being edited (or the value of '**iconstring**' if set). If the option is turned off, the icons just have the generic title *Vim*.

This only works with certain types of window managers which do *not* include the major Linux desktops KDE and Gnome.

Mouse Customization

The *Vim* editor is one of the few UNIX text editors that is mouse-aware. That means you can use the mouse for a variety of editing operations. You can customize the utility of the mouse by using the options discussed in the following sections.

Mouse Focus

Generally when you want to move from one *Vim* editing window to another, you must use one of the window change commands such as **CTRL-Wj** or **CTRL-Wk** or click the mouse inside that window. If you set the '**mousefocus**' ('**mousef**') option, the current *Vim* editing window is the one where the mouse pointer is located:

```
:set mousefocus
```

Note: This option affects only the windows within a *Vim* session. If you are using the X Windows System, the selection of the current window (X Client window) is handled by the window manager. On Microsoft Windows, the current window selection is handled by Microsoft Windows, which always forces you to use "click to type."

The '*mousemodel*' Option

The '**mousemodel**' ('**mousem**') option defines what the mouse does. There are three possible modes: **extend**, **popup**, and **popup_setpos**. To set the mouse model, use the following command:

```
:set mousemodel=mode
```

In all modes, the left mouse button moves the cursor, and dragging the cursor using the left button selects the text.

In **extend** mode, the right mouse button extends the text and the middle button pastes it in. This behavior is similar to the way an *xterm* uses the mouse.

In **popup** mode, the right mouse button causes a small pop-up menu to appear. This behavior is similar to what you find in most Microsoft Windows applications.

The **popup_setpos** mode is exactly like **popup** mode, except when you press the right mouse button, the text cursor is moved to the location of the mouse pointer, and then the pop-up menu appears.

The Vim Tutorial and Reference

The following table illustrates how '**mousemodel**' affects the mouse buttons.

Mouse	extend	popup	popup_setpos
Left	Place cursor	Place cursor	Place cursor
Drag-left	Select text	Select text	Select text
Shift-left	Search word	Extend selection	Extend selection
Right	Extend selection	Pop-up menu	Move cursor, and then pop-up menu
Drag-right	Extend selection		
Middle	Paste	Paste	Paste

Mouse Configuration

The '**mouse**' option enables the mouse for certain modes. The possible modes are as follows:

- n** Normal
- v** Visual
- i** Insert
- c** Command-line
- h** All modes when in a help file except "hit-return"
- a** All modes except the "hit-return"
- r** "more-prompt" and "hit-return" prompt

Mouse Mapping

The left mouse button (**<LeftMouse>**) moves the text cursor to where the mouse pointer is located. The **<RightMouse>** command causes *Vim* to enter visual mode. The area between the text cursor and the mouse pointer is selected. The **<MiddleMouse>** acts like the **P** command and performs a put to insert in the unnamed register in the file. If you precede the mouse click with a register specification (such as "**a**", for instance), the contents of that register are inserted.

The Vim Tutorial and Reference

If you have a wheel on your mouse, the up-wheel (**<MouseUp>**) moves three lines up. Similarly, a down-wheel movement (**<MouseDown>**) moves three lines down. If you press **Shift**, the screen moves a page. In other words, **<S-MouseUp>** goes up a page and **<S-MouseDown>** goes down a page.

Double-Click Time

The '**mousetime**' ('**mouse**') option defines the maximum time between the two presses of a double-click. The format of this command is as follows:

```
:set mousetime=time
```

Time is the time in milliseconds. By default, this is half a second (500ms).

Hiding the Mouse Cursor

When you are editing with the GUI, you have a text cursor and a mouse pointer to deal with. If that is too confusing, you can tell *Vim* to turn off the mouse pointer when not in use. To enable this feature, set the '**mousehide**' ('**mh**') option:

```
:set mousehide
```

When you start typing, the mouse pointer disappears. It reappears when you move the mouse.

Select Mode

The '**selectmode**' ('**s1m**') option defines when the editor starts select mode instead of visual mode. The following three events can trigger select mode:

mouse Moving the mouse (see '**mousemodel**', discussed earlier)

key Using some special keys

cmd The **v**, **V**, or **CTRL-V** command

The general form of the command is as follows:

```
:set selectmode=mode
```

Mode is a comma-separated list of possible select events (**mouse**, **key**, **cmd**).

The Vim Tutorial and Reference

The '**keymodel**' ('**km**') option allows the keys <Left>, <Right>, <Up>, <Down>, <End>, <Home>, <PageUp>, and <PageDown> to do special things. If you set this option as follows, *Shift+key* starts a selection: The

```
:set keymodel=startsel
```

If the option is set as follows, an unshifted key results in the section being stopped:

```
:set keymodel=stopsel
```

You can combine these options as well, as follows:

```
:set keymodel=startsel,stopsel
```

Custom Menus

The menus that *Vim* uses are defined in the file *\$VIMRUNTIME/menu.vim*.

If you want to write your own menus, you might first want to look through that file. To define a menu item, use the **:menu** (**:me**) command. The basic form of this command is as follows:

```
:menu menu-item command-string
```

(This command is very similar to the **:map** command.) The menu-item describes where on the menu to put the item. A typical menu-item is **File.Save**, which represents the item **Save** under the menu **File**. The ampersand character (&) is used to indicate an accelerator. In the *gvim* editor, for instance, you can use **Alt-F** to select **File** and **S** to select save. Therefore, the menu-item looks like **&File.&Save**. The actual definition of the **File.Save** menu item is as follows:

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

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

The second number (**340**) determines the location of the item within the pulldown menu. Lower numbers go on top, higher number on the bottom.

Figure 26-10 diagrams the priorities of the current menu items. The menu-item in this example is **&File.&Save<Tab>:w**. This brings up an important point: menu-item must be one string. If you want to put spaces or tabs in the name, you either use the <> notation (<space>, <tab>, for instance) or use the backslash (\) escape.

The Vim Tutorial and Reference

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

In this example, the name of the menu item contains a space (Do It) and the command is `:exit<CR>`. Finally, you can define menu items that exist for only certain modes. The general form of the menu command is as follows:

```
:mode menu priority menu-item command-string
```

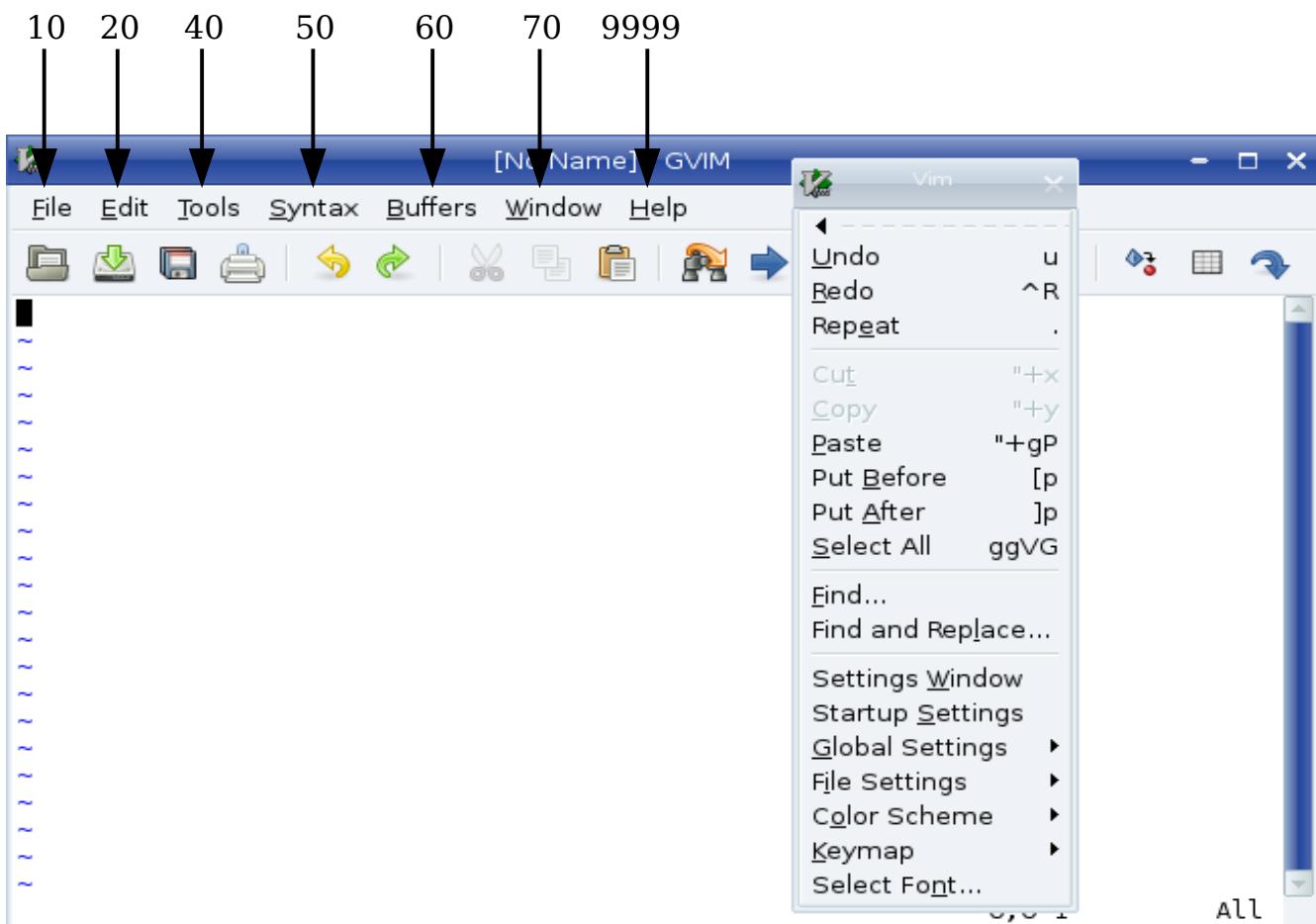


Figure 26-10: Menu item priorities.

The *mode* parameter is one of the following:

Character Mode

a	Normal, visual, a 335 operator-pending
n	Normal
v	Visual / Select
x	Visual
o	Operator-pending. (:omenu, :ome)

- ← 310
- ← 320
- ← 330
- ← 340
- ← 350
- ← 360
- ← 370
- ← 380
- ← 400
- ← Pa 410 → 68
- ← 420
- ← 430
- ← 440

Character	Mode
i	Insert
c	Command line

Special Menu Names

There are some special menu names. These are

- ToolBar** The toolbar (the icons under the menu)
PopUp The pop-up window that appears when you press the right mouse button in the edit window in certain modes

Limiting the Maximum Number of Generated Items

Some menus like the **Buffers** menu contain a dynamically generated list. In the case of the **Buffers** menu it's a list of buffers. The '**menuitems**' ('**mis**') option controls how many items can be generated.

Toolbar Icons

The toolbar uses icons rather than text to represent the command. The name of the icon is taken from the name of the menu item. For example, the *menu-item* named **ToolBar. New** causes the *New* icon to appear on the toolbar. The *Vim* editor has 28 built-in icons. The following table lists these.

Each icon has two names. The *New* icon, for instance, can be specified as **ToolBar.New** or **ToolBar.builtin00**.

The editor looks for a file in the *\$VIMRUNTIME/pximaps* directory for the icon, then will search its internal list of icons. The name of the icon file is *NAME.BMP* on Microsoft Windows and *tb_{name}.xpm* on UNIX. On Microsoft Windows, the icon may be any size. On UNIX, it must be 20×20 pixels.

Icon	Name	Alternative Name
	New	builtin00
	Open	builtin01
	Save	builtin02
	Undo	builtin03

The Vim Tutorial and Reference

Icon	Name	Alternative Name
	Redo	builtin04
	Cut	builtin05
	Copy	builtin06
	Paste	builtin07
	Print	builtin08
	Help	builtin09
	Find	builtin10
	SaveAll	builtin11
	SaveSesn	builtin12
	NewSesn	builtin13
	LoadSesn	builtin14
	RunScript	builtin15
	Replace	builtin16
	WinClose	builtin17
	WinMax	builtin18
	WinMin	builtin1
	WinSplit	builtin20
	Shell	builtin21
	FindPrev	builtin22
	FindNext	builtin23
	FindHelp	builtin24

The Vim Tutorial and Reference

Icon	Name	Alternative Name
	Make	builtin25
	TagJump	builtin26
	RunCtags	builtin27

Toolbar Tips

The toolbar can display a "tip" when the cursor is placed over an icon. To define the tip, issue the **:tmenu** (**:tm**) command:

```
:tmenu menu-item tip
```

For example, the following command causes the tip Open file to display when the cursor rests over the Open icon (see Figure 26-11):

```
:tmenu ToolBar.Open Open file
```

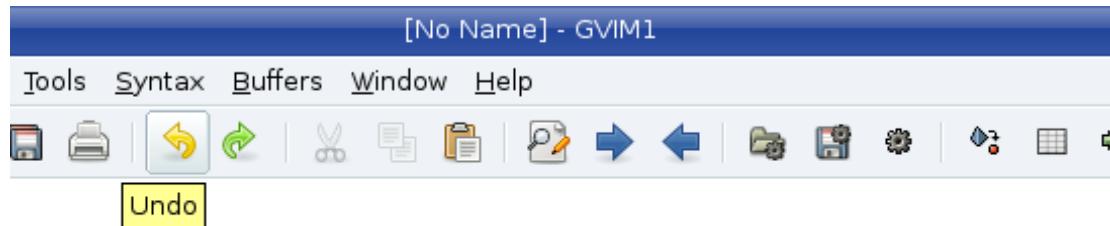


Figure 26-11: ToolTip.

Listing Menu Mappings

The following command lists all the menu mappings:

```
:menu
```

The Vim Tutorial and Reference

```
:menu  
---- Menus ---  
1ToolBar  
10 Open  
    n*:browse confirm e<CR>  
    v*:<C-C>:browse confirm e<CR><C-\><C-G>  
    s*:<C-C>:browse confirm e<CR><C-\><C-G>  
    o*:<C-C>:browse confirm e<CR><C-\><C-G>  
20 Save  
    n*s :if expand("%") == ""|browse confirm w|else|confirm w|endif<CR>  
    v*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><  
C-\><C-G>  
    s*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><  
C-\><C-G>  
    o*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><  
  
... lots of other lines ...
```

The problem with the **:menu** command is that you get 51 screens of data. That is a lot. To get just the menu items for a specific top-level menu, use the following command:

```
:menu menu
```

For example, the following command lists only the menu items for the File menu:

```
:menu File
```

The next command lists the items for just the **File.Save** menu:

```
:menu File.Save
```

```
:menu File.Save  
--- Menus ---  
340 &Save^I:w  
    n*s :if expand("%") == ""|browse confirm w|else|confirm w|endif<CR>  
    v*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><C-  
\><C-G>  
    s*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><C-  
\><C-G>  
    o*s <C-C>:if expand("%") == ""|browse confirm w|else|confirm w|endif<CR><C-  
\><C-G>
```

The letters at the beginning of each line denote the mode in which the command applies. They correspond to the letters used for the mode parameter described earlier.

Executing a Menu Item

The `:emenu (:en)` command executes the menu-item as if the user had selected the command from the menu:

```
:emenu menu-item
```

No Remapping Menus

The `:menu` command defines a menu item. If you want to define an item and make sure that no mapping is done on the right side, use the `:noremenu` (`:noreme`) command.

Like other mapping commands this has several forms depending on mode:

Command	Normal	Visual	Operator Pending	Insert	Command Line
<code>:noremenu</code>	√	√	√		
<code>:nnoremenu</code> <code>:nnoreme</code>	√				
<code>:vnoremenu</code> <code>:vnoreme</code>		√			
<code>:onoremenu</code> <code>:onoreme</code>			√		
<code>:noremenu!</code>				√	√
<code>:inoremenu</code> <code>:inoreme</code>				√	
<code>:cnoremenu</code> <code>:cm</code>					√

Removing Menu Items

The following command removes an item from the menu:

```
:mode unmenu menu-item
```

If you use an asterisk (*) for the menu-item, the entire menu is erased. To remove a ToolTip, use the `:tunmenu` (`:tu`) command:

The Vim Tutorial and Reference

```
:tunmenu menu-item
```

Tearing Off a Menu

You can tear off a menu by using the dotted tear-off line on the GUI. Another way to do this is to execute the **:tearoff** (**:te**) command:

```
:tearoff menu-name
```

Translating A Menu

The base language of all built-in menus is English. If you want to add a translation to your own language, use the **:menutranslate** (**:menut**) command. For example:

```
:menutranslate Open Abierto
```

Now when you have “Open” as an item in a menu, “Abierto” will be displayed instead.

You'll find a whole lot of menu translation files in the `$VIMRUNTIME/lang` directory. Depending on the setting of **v:lang**, a file in this directory called `menu_{v_lang}.vim` will be loaded.

If you want to have your own translations properly loaded into *Vim* you need to execute the following commands in order:

```
:source $VIMRUNTIME/delmenu.vim  
  
:menutranslate {English} {Other}  
: ... more :menutranslate commands  
  
:source $VIMRUNTIME/menu.vim
```

The actual translation that's used for the menu is controlled by the '**language menu**' ('`lm`') option. Set this to your local language if you need to understand the menus.

To clear the current set of menu translations, use the **:menutranslate clear** command.

Special GUI Commands

The *Vim* editor has many commands designed for use in GUI-based menus. These commands are all based around various dialog boxes (such as the file browser connected with the **File.Open** menu).

The File Browsers

The **:browse** (**:bro**) command opens up a file browser and then executes a command on the file selected. For example, the following command opens a file browser and enables the user to select a file (see Figure 26-12):

```
:browse edit
```

The editor then performs an **:edit** file command.

The Vim Tutorial and Reference

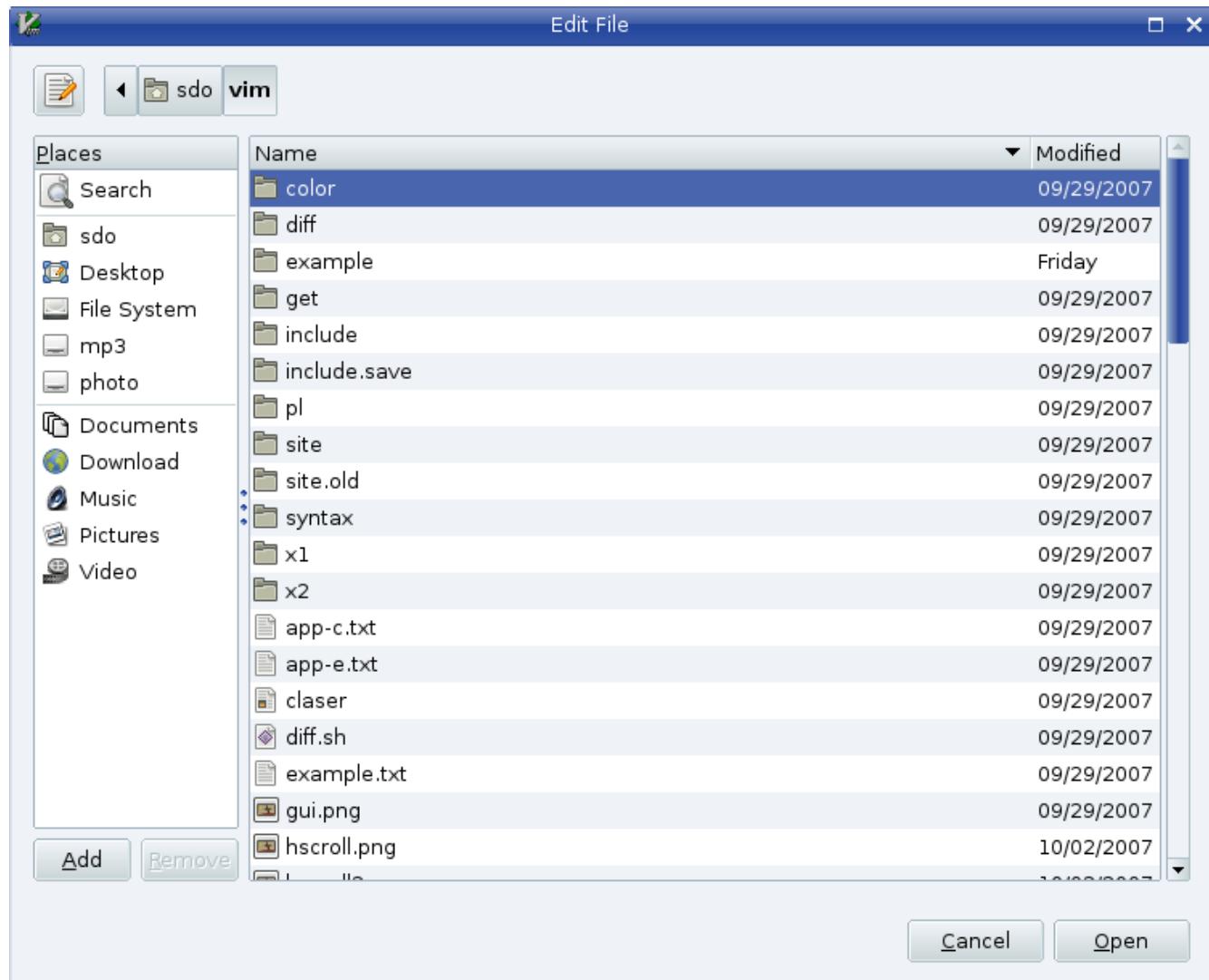


Figure 26-12: File browser.

The general form of the **:browse** command is as follows:

```
:browse command [directory]
```

The **command** is any editor command that takes a filename as an argument. Commands such as **:read**, **:write**, and **:edit** fall into this category.

The **[directory]** parameter, if present, determines the directory in which the browser starts.

The Vim Tutorial and Reference

If the **[directory]** parameter is not present, the directory for the browser is selected according to the '**browsedir**' ('**bsdir**') option. This option can have one of three values:

- last** Use the last directory browsed (default).
- buffer** Use the same directory as the current buffer.
- current** Always use the current directory.

Therefore, if you always want to start in the current directory, put the following command in your initialization file:

```
:set browsedir=current
```

Finding a String

The **:promptfind** (**:pro**) command displays a search dialog box (see Figure 26-13):

```
:promptfind [string]
```



Figure 26-13: **:promptfind** dialog box.

If a string is specified, the string is used as the initial value of the **Find what** field. When the user presses the **Find Next** button, the *Vim* editor searches for the given string.

Replace Dialog Box

There is a similar dialog box for the replace command. The **:promptrep1** (**:promptr**) command displays a Replace dialog box (see Figure 26-14):

```
:promptrep1 [string]
```

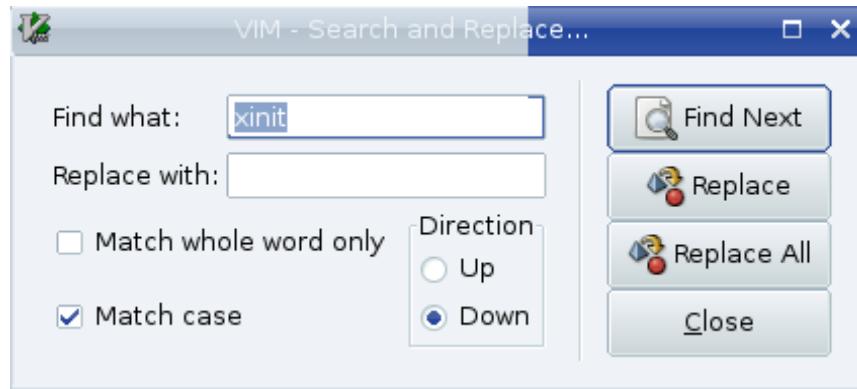


Figure 26-14: `:promptrep1` dialog box.

If a *string* parameter is present, it is used for the **Find What** parameter.

Finding Help

The `:helpfind` (`:helpf`) command brings up a dialog box that enables you to type in a subject that will be used to search the help system:

```
:helpfind
```

Confirmation

The `:confirm` (`:conf`) command executes a command such as `:quit` that has the potential for destroying data. If the execution of the command would destroy data, a confirmation dialog box displays. For example, the following command on a modified buffer results in the dialog box displayed in Figure 26-15:

```
:confirm :quit
```

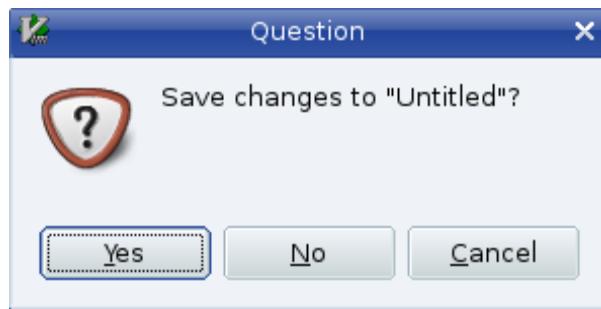


Figure 26-15: Confirmation dialog box.

Note: This command works for the terminal version of *Vim* as well, but the Confirmation dialog box does not look as nice. This is illustrated in Figure 26-16.

```
This is a test
```

```
:confirm :q
Save changes to "Untitled"?
[Y]es, (N)o, (C)ancel:
```

Figure 26-16: Confirmation without a GUI.

Browsing the Options

The `:browse set` (`:bro set`, `:opt`, `:options`) command opens a window that enables you to browse through the options:

```
:browse set
```

Figure 26-17 shows the screen.

The Vim Tutorial and Reference

```
" Each "set" line shows the current value of an option (on the left).
" Hit <CR> on a "set" line to execute it.
"           A boolean option will be toggled.
"           For other options you can edit the value.
" Hit <CR> on a help line to open a help window on this option.
" Hit <CR> on an index line to jump there.
" Hit <Space> on a "set" line to refresh it.

1 important
2 moving around, searching and patterns
3 tags
option-window

~
[No Name]
"option-window" [New File]
```

Figure 26-17: **:browse set.**

This window gives you access to all the options. The beginning is a short table of contents, starting with the following:

```
1 important
2 moving around, searching and patterns
```

You can use the cursor commands to position the cursor on one of the items and press **<CR>** to get a list of the options for this section. If you were to move the cursor down to the first entry (important) and press **<CR>**, for instance, you would get Figure 26-18.

```
1 important

compatible      behave very Vi compatible (not advisable)
                set nocp          cp
coptions        list of flags to specify Vi compatibility
                set cpo=aABceFs
insertmode       use Insert mode as the default mode
                set noim          im
paste            paste mode, insert typed text literally
                set nopaste       paste
pastetoggle      key sequence to toggle paste mode
option-window

~
~

[No Name]
/ 1 important
```

Figure 26-18: **:browse set detail screen.**

The Vim Tutorial and Reference

The first option in this window is the '**compatible**' option. According to the help text beside it, setting this option causes *Vim* to "behave very Vi compatible (not advisable)." The abbreviation for this option is '**cp**', and the current setting is '**nocp**'. If you move down to the '**compatible**' line and press <CR>, you get a full help screen on the option (see Figure 26-19).

```
*'compatible'* *'cp'* *'nocompatible'* *'nocp'*  
'compatible' 'cp'      boolean (default on, off when a |vimrc| or |gvimrc|  
                                file is found)  
                      global  
                      {not in Vi}  
This option has the effect of making Vim either more Vi-compatible, or  
make Vim behave in a more useful way.  
This is a special kind of option, because when it's set or reset,  
other options are also changed as a side effect. CAREFUL: Setting or  
resetting this option can have a lot of unexpected effects: Mappings  
are interpreted in another way, undo behaves differently, etc. If you  
set this option in your vimrc file, you should probably put it at the  
very start.  
By default this option is on and the Vi defaults are used for the  
options. This default was chosen for those people who want to use Vim  
just like Vi, and don't even (want to) know about the 'compatible'  
option.  
When a |vimrc| or |gvimrc| file is found while Vim is starting up,  
options.txt [Help][RO]  
compatible      behave very Vi compatible (not advisable)  
option-window  
  
[No Name]
```

Figure 26-19: **:browse set** help screen.

If you close the help screen and move down to the **set nocp** line and press <CR>, the option is toggled. (This works for all Boolean options.) The next option ('**cpoptions**') is a string. To change its value, just edit it using the normal *Vim* editing commands, and then press <CR> to set the option to this value.

Using the Clipboard

The '**clipboard**' ('**cl**') option controls how *Vim* treats text selected with the mouse. If you use the following command, *Vim* takes all the text that should go in the unnamed register and puts it in the clipboard register:

```
:set clipboard=unnamed
```

This means that the text is placed on the system clipboard and can be pasted into other applications.

The Vim Tutorial and Reference

Another option is this:

```
:set clipboard=autoselect
```

When this option is set, any text selected in visual mode is put on the system clipboard (if possible). (The **a** flag of the '**guioptions**' ('**go**') option does the same thing.)

The '**autoselect**' option works for both the GUI and console versions of *Vim*.

Coloring

opt-guioptionsd(27-3)When *Vim* starts the GUI, it tries to figure out whether you have a light or dark background and performs the '**background**' ('**bg**') option to set the proper value:

```
:set background=value
```

The syntax files use the value of this option to determine which colors to use.

Warning: When the GUI is started, the value of this option is light. The *.gvimrc* file is then read and processed. After this, the window is created. Only after the window is created can *Vim* tell the color of the background, so it is only after this that the background option is set. This means that anything in the *.gvimrc* that depends on the background being correct will fail.

Selecting the Font

If you do not like the font that *Vim* uses for its GUI, you can change it by using the '**guifont**' ('**gfn**') option:

```
:set guifont=font
```

Font is the name of a font. On an X Windows System, you can use the command *xlsfonts* to list out the available fonts. On Microsoft Windows, you can get a list of fonts from the Control Panel. You can also use the following command:

```
:set guifont=*
```

This command causes *Vim* to bring up a font selection window from which you can pick your font.

The Vim Tutorial and Reference

You can also turn on and off anti-aliasing with the '**antialias**' ('**anti**') option. Currently this only works on the Mac.

Customizing Select Mode

The '**selection**' ('**sel**') option defines how a selection is handled. The possible values are as follows

old	Does not allow selection for one character past the end of a line. The last character of the selection is included in the operation.
inclusive	The character past the end of the line is included, and the last character of the selection is included in the operation.
exclusive	The character past the end of the line is included, and the last character of the selection is not included in the operation.

Mouse Usage in Insert Mode

Clicking the left mouse button (<LeftMouse> in *Vim* terminology) causes the cursor to move to where the mouse pointer is pointing.

If you have a wheel on your mouse, the mouse-wheel commands act just like they do in normal mode.

Microsoft Windows - Specific Commands

By setting the '**winaltkeys**' ('**wnk**') option to "no," *Vim* will take over the entire keyboard. This means that you can use the **Alt** key for keyboard commands and mappings. However, Microsoft Windows generally uses the **Alt** key to access menus.

The **:simalt** (**:si**) key simulates the pressing of **Alt+key**. You can use this in the following command, for example:

```
:map <M-f> :simalt f<CR>
```

This command tells *Vim* that when Meta-F (Alt+f in Microsoft Windows terminology) is pressed, the editor is to simulate the pressing of the Alt+f key. This brings down the File menu.

Changing the Appearance of the Cursor

The '**guicursor**' ('**gcr**') option defines how the cursor looks for the GUI version of *Vim*. The format of this command is as follows:

```
:set guicursor=mode:style[-highlight],mode:style[-highlight],...
```

You can set the mode as follows:

n	Normal mode
v	Visual mode
ve	Visual mode with ' selection ' exclusive (same as v , if not specified)
o	Operator-pending mode
i	Insert mode
r	Replace mode
c	Command-line normal (append) mode
ci	Command-line insert mode
cr	Command-line replace mode
sm	showmatch in insert mode
a	All modes

You can combine by separating them with hyphens, as follows:

n-v-c For normal, visual, and command modes

The style is as follows:

horN	Horizontal bar, N percent of the character height
verN	Vertical bar, N percent of the character width
block	block cursor, fills the whole character
blinkwaitN	
blinkonN	
blinkoffN	When these options are specified, the system waits for blinkwait milliseconds, and then turns the cursor off for blinkoff and on for blinkon . The off/on cycle repeats.

And *highlight* is a highlight group name.

The '**mouseshape**' ('**mouses**') option defines which cursor to use for the various modes in the GUI version of *Vim*. The format of this command is as follows:

```
:set mouseshape=mode:cursor,mode:cursor,...
```

The modes are:

- a** Everywhere.
- c** Appending to the command-line.

The Vim Tutorial and Reference

ci	Inserting in the command-line.
cr	Replacing in the command-line.
e	Any mode, pointer below last window.
i	Insert mode.
m	At the Hit ENTER or More prompts.
ml	At the Hit ENTER or More prompts, with cursor in the last line.
n	Normal mode.
o	Operator-pending mode.
r	Replace mode.
s	Any mode, pointer on a status line.
sd	Any mode, while dragging a status line.
vd	Any mode, while dragging a vertical separator line.
ve	Visual mode with 'selection' "exclusive" (same as ' v ', if not specified).
vs	Any mode, pointer on a vertical separator line.
v	Visual mode.

The cursor names come from the following list:

arrow	Normal mouse pointer. (Microsoft Windows, X Window System)
beam	I-beam. (Microsoft Windows, X Window System)
blank	No pointer at all. (Do not use this one unless you really know what you are doing.) (Microsoft Windows, X Window System)
busy	The system's usual busy pointer. (Microsoft Windows, X Window System)
crosshair	Like a big thin +. (X Windows System)
hand1	Black hand. (X Windows System)
hand2	White hand. (X Windows System)
leftright	Left-right sizing arrows. (Microsoft Windows, X Window System)
lrsizing	Indicates left-right resizing. (X Windows System)
no	The system's usual 'no input' pointer. (Microsoft Windows, X Window System)
pencil	Cursor that looks like a pencil. (X Windows System)
question	Large question mark. (X Windows System)
rightup-arrow	Arrow pointing right-up. (X Windows System)
udsizing	Indicates up-down resizing. (X Windows System)
up-arrow	Arrow pointing up. (Microsoft Windows, X Window System)
updown	Up-down sizing arrows. (Microsoft Windows, X Window System)
<number>	Any X11 pointer number. (See X11/cursorfont.h.) (X Windows System)

Line spacing

The '**linespacing**' ('**lsp**') option tell *Vim* how many extra pixels of space to put between lines.

X Windows System - Specific Commands

In the X Windows System, the window manager is responsible for the border around the window and other decorations. The '`guiheadroom`' ('`ghr`') option tells the *Vim* editor how big the margin is around the window (top and bottom) so that when it goes into full screen mode, it can leave room for the border.

Selecting the Connection with :shell Commands

What happens when you are using the GUI window and you try to execute a `:shell` (`:sh`) command? Usually the system uses a UNIX device called *pty* to handle the command interface.

If you want to connect using a pipe, clear the '`guipyty`' option:

```
:set noguipyty
```

Otherwise the default is used and a *pty* connection made between the shell and the GUI:

```
:set guipyty
```

MS-DOS-Specific Commands

The `:mode` (`:mod`) command changes the screen mode of an MS-DOS window:

```
:mode mode
```

This command is effective only if you are editing inside MS-DOS; it does not work inside a Microsoft Windows GUI.

Mode is an MS-DOS screen mode such as B80, B40, c80, c40, or one of the screenmode numbers.

Chapter 27: Expressions and Functions

The *Vim* Editor contains a rich scripting language. This command language gives you tremendous flexibility when it comes to customizing your editor for specialized tasks.

This chapter covers the following:

- Basic variables and expressions
- The **:echo** statement Control statements
- User-defined functions
- A complete list of built-in functions

Basic Variables and Expressions

The *Vim* editor enables you to define, set, and use your own variables. To assign a value to a variable, use the **:let** command. The general form of this command is as follows:

```
:let variable = expression
```

The *Vim* editor uses the same style variable names as most other languages--that is, a variable begins with a letter or underscore and then consists of a series of letters, digits, and the underscore.

To define the variable `line_size`, for example, use this command:

```
:let line_size = 30
```

To find out what the variable is, use the **:echo (:ec)** command:

```
:echo "line_size is" line_size
```

When entered, this command results in *Vim* displaying the following on the last line:

```
line_size is 30
```

Variables can contain numbers (such as 30) or strings (such as "foo"). For example:

```
:let my_name = "Steve Oualline"
```

Special Variable Names

The *Vim* editor uses special prefixes to denote different types of variables. The prefixes are as follows:

Name	Use
All uppercase, digits, and underscore	Variables which can be stored in the <code>.viminfo</code> file if the ' <code>viminfo</code> ' (' <code>vi</code> ') option contains the ! flag.
Initial uppercase letter, lowercase letter somewhere inside	Variable saved by the make session (<code>:mksession</code> , <code>:mks</code>) command.
	<pre>:let Save_this_option = 1 " Options saved in session :let forget_this = "yes" " Discarded between sessions</pre>
All lowercase, digits, and underscore	A variable not stored in any save file.
<code>\$environment</code>	Environment variable.
<code>@register</code>	Text register.
<code>&option</code>	The name of an option.
<code>b:name</code>	The variable is local to the buffer. Each buffer can have a different value of this variable.
<code>w:name</code>	A variable local to a window.
<code>g:name</code>	A global variable. (Used inside functions to denote global variables.)
<code>a:name</code>	An argument to a function.
<code>v:name</code>	A <i>Vim</i> internal variable

Some examples:

The Vim Tutorial and Reference

```
" The environment variable $PAGER contains
" the name of the page viewing command
:let $PAGER = "/usr/local/bin/less"

" Display the value of the last search
pattern :echo "Last search was "@/

" The following two commands do the same
thing :let &autoindent = 1
:set autoindent

" Define the syntax for the current buffer
:let b:current_syntax = "c"

"Note: This doesn't handle all the side
" effects associated with
"changing the language of the buffer
```

The internal variables (**v:name**) are used by *Vim* to store a variety of information. The following table shows the full list of variables.

v:count	The count given for the last normal-mode command.
v:count1	Like v:count , except that it defaults to 1 if no count is specified.
v:errmsg	The last error message.
v:warningmsg	The last warning message.
v:statusmsg	The last status message.
v:shell_error	Result of the last shell command. If 0, the command worked; if non-0, the command failed.
v:this_session	Full filename of the last loaded or saved session file.
v:version	Version number of <i>Vim</i> . Version 5.01 is stored as 501.

Constants

The *Vim* editor uses a variety of constants. There are the normal integers:

123	Simple integer
0123	Octal integer
0xAC	Hexadecimal

There are also string constants:

"string"	A simple string
-----------------	-----------------

The Vim Tutorial and Reference

'string' A literal string

The difference between a simple string and a literal string is that in a simple string, characters escaped by backslash are expanded, whereas in a literal string a backslash is just a backslash. For example:

```
:echo ">\100<"  
>@<  
:echo '>\100<'  
>\100<
```

Note: The character number octal 100 is @.

Expressions

You can perform a variety of operations on integers. These include the arithmetic operator s:

int + int	Addition
int - int	Subtraction
int * int	Multiplication
int / int	Integer divide (and truncate)
int % int	Modulo
- int	Negation

Note: Strings are automatically converted to integers when used in conjunction with these operators.

In addition, a number of logical operators work on both strings and integers. These return a 1 if the comparison succeeds and 0 if it does not.

var == var	Check for equality.
var != var	Inequality.
var < var	Less than.
var <= var	Less than or equal to.
var > var	Greater than.
var >= var	Greater than or equal to.

In addition, the comparison operators compare a string against a regular expression. For example, the following checks the given string ("word") against the regular expression "\w*" and returns a 1 if the string matches the expression:

```
"word" =~ "\w*"
```

The two regular expression comparison operators are as follows:

string =~ regexp	Regular expression matches.
string !~ regexp	Regular expression does not match.

The Vim Tutorial and Reference

In addition, strings have the following special comparisons:

string ==? string	Strings equal, ignore case.
string ==# string	Strings equal, case must match.
string !=? string	Strings not equal, ignore case.
string !=# string	Strings not equal, case must match.
string <? string	Strings less than, ignore case.
string <# string	Strings less than, case must match.
string <=? string	Strings less than or equal, ignore case.
string <=# string	Strings less than or equal, case must match.
string >? string	Strings greater than, ignore case.
string ># string	Strings greater than, case must match.
string >=? string	Strings greater than or equal, ignore case.
string >=# string	Strings greater than or equal, case must match.

There are three forms of each operator. The bare form (i.e. `==`) honors the '`ignorecase`' option. The `? form` (i.e.`==?`) always ignores case differences while the `# form` (i.e.`==#`) never ignores different case characters.

Deleting a Variable

The `:unlet (:unl)` command deletes a variable:

```
:unlet[!] name
```

Generally, if you try to delete a variable that does not exist, an error result. If the override (!) character is present, no error message results.

Locking and unlocking a variable

A variable can be locked so that its value can not be changed using the `:lockvar (:lockv)` command:

```
:lockvar line_size
```

The Vim Tutorial and Reference

After this command is executed any **:let** command that attempts to change this variable will fail. To unlock a variable, use the **:unlockvar** (**:unlo**) command.

Locking Arrays and Dictionaries

The **:lockvar** command takes a depth as an argument. A depth of 1 means that the variable itself is locked. You can not add or remove items. However, you can change items. That because the item is indirectly referenced and is considered stored at a depth of 2.

If you lock with a depth of two then the array is locked and any items in the array are locked. But any items referenced by items in the array (which require two levels of indexing to access) are not locked.

A level of 3 locks the array, the items, items referenced by the items, but items referenced by items inside items (requiring three levels of indexing to get to.)

The **:lockvar!** command with the override(!) sets the locking level to 100 and pretty much locks everything.

Entering Commands

When you are entering arguments to a command, you can use a number of special words and characters. The words work when you need to enter an argument such a file name, directory, or other word argument such as the argument to **:tag** and the **:vimgrep** pattern. The special words are::

%	Current filename
#	Alternate filename
<cword>	The word under the cursor.
<WORD>	The WORD under the cursor.
<file>	The filename under the cursor.
<file>	The name of a file being read or written during the execution of a related autocmd. (See <i>Chapter 13: Autocommands</i> for more information.)
<abuf>	The current buffer number in an autocmd.
<amatch>	Like <abuf> , but when used with a FileType or Syntax event it is not the file name, but the file type or syntax name.
<sfile>	The name of the file currently being :sourced .

The Vim Tutorial and Reference

You can modify each of these words by one or more of the modifiers listed here (for example, the **:p** modifier, which turns a filename into a full pathname). If the name of the file under the cursor is *test.c*, for instance, **<cfile>** would be *test.c*. On the other hand, **<cfile>:p** would be */home/oualline/examples/test.c*.

You can use the following modifiers:

:p	Turn a filename into a full path. Must appear first if multiple modifiers are used.
:~	Turn an absolute path such as <i>/home/oualline/examples/test.c</i> into a short version using the ~ notation, such as <i>~oualline/examples/test.c</i> .
::	Turn the path into one relative to the current directory, if possible.
:h	Head of the filename. For example, <i>../path/test.c</i> yields <i>../path</i> .
:t	Tail of the filename. Therefore, <i>../path/test.c</i> yields <i>test.c</i> .
:r	Filename without extension. Therefore, <i>../path/test.c</i> yields <i>../path/test</i> .
:e	Extension.
:s?from?to?	Substitution changing the pattern from to the pattern to, first occurrence.
:gs?from?to?	Substitution changing the pattern from to the pattern to, all occurrences.

How to Experiment

You can determine how *Vim* will apply modifiers to a filename. First create a text file whose content is the filename on which you want to run experiments. Put the cursor on this filename and then use the following command to test out a modifier:

```
:echo expand("<cword>:p")
```

(Change **:p** to whatever modifier you want to check.)

The following sections discuss the **:echo** statement and expand function in more detail.

The :echo Statement

The **:echo** statement just echoes its arguments. For example:

The Vim Tutorial and Reference

```
:echo "Hello world"  
Hello world
```

You can also use it to display the value of a variable:

```
:let flag=1  
:echo flag  
1
```

The **:echon** command echoes the arguments, but does not output a newline. For example:

```
:echo "aa" | echo "bb"  
aa  
bb  
:echon "aa" | echon "bb"  
aabb
```

Note: The bar (|) is used to separate two commands on the same line.

Echoing in Color

You can use the **:echohl** (**:echoh**) command to change the color of the output **:echo** to a given highlight group. For example:

```
:echohl ErrorMsg  
:echo "A mistake has been made"  
:echohl None
```

Note: Good programming practice dictates that you always reset the highlighting to None after your message. That way you do not affect other **:echo** commands.

If you want to see what highlight groups are defined, use the command **:highlight** (**:hi**):

```
:highlight
```

Printing error messages using :echoerr

The **:echoerr** (**:echoe**) command acts much like **:echo** except the string is highlighted using the same colors as an error message and it is placed in the error history.

If this command is used in a script or function, a line number will be added to the message.

The Vim Tutorial and Reference

Finally, if this command occurs inside a **:try / :catch** block, it acts as if an exception where thrown. (See *Exceptions* on page 497 for more information.)

Echoing message

The **:echomsg** (**:echom**) command acts just like **:echo** only the string is saved in the message buffer and will appear if you issue a **:messages** command.

Control Statements

The *Vim* editor has a variety of control statements that enable you to change the flow of a macro or function. With these, you can make full use of *Vim*'s sophisticated script language.

The :if Statement

The general form of the **:if** statement is as follows:

```
:if {condition}
:    " Statement
:    " Statement
:endif
```

(You can abbreviate **:endif** as **:en**, but please don't. It makes your code hard to read.)

The statements inside the **:if** statement are executed if the condition is non-zero. The four-space indent inside the **:if** is optional, but encouraged because it makes the program much more readable. The **:if** statement can have an **:else** clause:

```
:if {condition}
:    " Statement
:    " Statement
:else
:    " Statement
:    " Statement
:endif
```

(**:el** can be used for **:else** if you wish to make your code hard to read.)

Finally, the **:elseif** (**:elsei**) keyword is a combination of **:if** and **:else**. Using it removes the need for an extra **:endif**:

The Vim Tutorial and Reference

```
:if &term == "xterm"
:    " Do xterm stuff
:elseif &term == "vt100"
:    " Do vt100 stuff
:else
:    " Do non xterm and vt100 stuff
:endif
```

Looping

The `:while (:wh)` command starts a loop. The loop ends with the `:endwhile (:endw)` command:

```
:while counter < 30
:    let counter = counter + 1
:    " Do something
: endwhile
```

The `:continue (:con)` command goes to the top of the loop and continues execution. The `:break (:brea)` command exits the loop:

```
:while counter < 30
:    if skip_flag
:        continue
:    endif
:    if exit_flag
:        break
:    endif
:    "Do something
: endwhile
```

The `:for` Loop

The `:for` statement loops through all the elements of a list. For example:

```
:set list=['a', 'b', 'c']
:for item in list
:    echo item
:endfor
```

(For the lazy, `:endfo` can be used to end the loop. Please don't use it however as it makes the code less readable.)

If the items in the list are arrays, you can specify a set variables in the `:for`. The variables will be assigned each corresponding element in the array. For example, the following two code fragments are equivalent:

The Vim Tutorial and Reference

```
:let m=[[11,12],[21,22],[31,32]]  
:for [x,y] in m  
:  echo x " -> " y  
:endfor  
  
:for q in m  
:  echo q[0] " -> " q[1]  
:endfor
```

The :execute Command

The **:execute** (**:exe**) executes the argument as a normal command-mode command:

```
:let command = "echo 'Hello world!'"  
:execute command  
Hello World
```

Exceptions

When an error occurs *Vim* throws an exception. If this exception is not caught, *Vim* will print an error message and abort processing. For example:

```
:badcommand  
E492: Not an editor command: badcommand
```

If you want to intercept exception you need to create a try/catch block. You start by putting everything in a **:try** block.:

```
:try  
:  " Do something that might cause an exception
```

The **:try** is followed by all the commands you hope to execute.

Next comes the exception handling code. Each exception gets its own **:catch** (**:cat**) statement which tells *Vim*, when an error occurs (exception is throw), come here to handle it.

```
:catch /^E492/  
:  echo 'Someone typed a bad command'
```

The argument to **:catch** is a regular expression. *Vim* is highly text oriented and the exception handling is no exception.⁷ Our regular expression (**/^E492/**) matches any string beginning with “E492”. Since all *Vim* errors are numbered this will match the “Not an editor command” error no matter what language is selected for the error message.

⁷ No pun intended.

The Vim Tutorial and Reference

The regular expression is optional. If omitted, all error messages would be caught.

```
:catch  
:    echo 'Something unexpected happened.'
```

The `:catch` statements are executed in order. So in our little example, the statement will be checked against `/^E492/`, then no-argument `:catch` statement.

If we have any cleanup code we can put it a `:finally (:fina)` block:

```
:finally  
:    " Clean up
```

The entire `:try` block is then ended with a `:endtry (:endt)` statement.

```
:endtry
```

What's left is to cause an error. The preferred method is to use the `:throw (:th)` command. For example:

```
:throw "ERROR: You made a mistake"
```

This will cause current processing to stop and *Vim* will continue execution at the first matching `:catch` statement it finds. If there is no matching `:catch` statement, an “E605: Exception not caught” error will occur.

The other way of throwing an exception is to use the `:echoerr` command. (See *Printing error messages using :echoerr* on page 494 for more information.) This differs from `:throw` in that if the error is not caught, the error message is printed. In other words, an uncaught `:echoerr` message does not cause a “E605”.

All *Vim* errors (like “E492: Not an editor command”) act like they were produced with `:echoerr`.

The `:finish (:fini)` command will cause all subsequent statements inside a `:try` to be skipped. The next statement to be executed will be the first statement in the `:finally` block if there is any. If not, the it will be the first statement after the `:endtry`. If this statement is executed outside a `:try`, all the remaining statement inside the file will be skipped. (The command can not be executed outside of a file or `:try`.)

Defining Your Own Function

The *Vim* editor enables you to define your own functions. The basic function declaration begins with a **:function** (**:fu**) statement:

```
:function name(var1, var2, ...)
```

Note: Function names must begin with a capital letter.

It ends with an **:endfunction** (**:endif**) statement:

```
:endfunction
```

Let's define a short function to return the smaller of two numbers, starting with this declaration:

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

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

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

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

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

The variable **smaller** is a local variable. All variables used inside a function are local unless prefixed by a **g:**.

Warning: A variable outside a function declaration is called **var**; whereas inside if you want to refer to the same variable, you need to call it **g:var**. Therefore, one variable has two different names depending on the context.

You now use the **:return** (**:retu**) statement to return the smallest number to the user. Finally, you end the function:

```
:    return smaller
:endfunction
```

The complete function definition is as follows:

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

The Vim Tutorial and Reference

```
:        let smaller = a:num1
:    else
:        let smaller = a:num2
:    endif
:    return smaller
:endfunction
```

Note: I know that this function can be written more efficiently, but it is designed to be a tutorial of features, not efficiency.

Using a Function

You can now use your function in any *Vim* expression. For example:

```
:let tiny = Min(10, 20)
```

You can also call a function explicitly using the function name with the **:call** (**:cal**) command:

```
:[range] call function([parameters])
```

If a *range* is specified, the function is called for each line, unless the function is a special "range"-type function (as discussed later).

Function Options

If you attempt to define a function that already exists, you will get an error. You can use the force option (!) to cause the function to silently replace any previous definition of the function:

```
:function Max(num1, num2)
:    " Code
:endfunction
```

```
:function Max(num1, num2, num3)
-- error --
```

```
:function! Max(num1, num2, num3)
-- no error --
```

By putting the **range** keyword after the function definition, the function is considered a range function. For example:

```
:function Count_words() range
```

The Vim Tutorial and Reference

When run on a range of lines, the variables **a:firstline** and **a:lastline** are set to the first and last line in the range.

If the word **abort** follows the function definition, the function aborts on the first error. For example:

```
:function Do_It() abort
```

Finally, *Vim* enables you to define functions that have a variable number of arguments. The following command, for instance, defines a function that must have 1 argument (start) and can have up to 19 additional arguments:

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

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

```
:function Show(start, ...)
:    let index = 1    " Loop index
:    echo "Show is" a:start
:    while (index <= a:0)
:        echo "Arg" index "is " a:000[index]
:        let index = index + 1
:    endwhile
:endfunction
```

Listing Functions

The **:function** command lists all the user-defined functions:

```
:function
function FTCheck_nroff()
function FTCheck_asm()
```

To see what is in a single function, execute this command:

```
:function name
```

For example:

```
:function Show
    let index = 1      " Loop index
    echo "Show is" a:start
    while (index <= a:0)
        echo "Arg" index "is " a:index
        let index = index + 1
    endwhile
```

Deleting a Function

To delete a function, use the command **:defun**ction (**:delf**unction) :

```
:defun
```

Running Functions in a Sandbox

Normally when you execute a function, the code has full access to Vim and can alter settings and the files being edited. If you want to run things in a more secure environment use the **:sand**box (**:san**) command.

The general form of this command is:

```
:sand
```

For example:

```
:sand :call Show("foo", "bar")
```

Debugging a Function

Vim has a built-in debugger that lets you debug functions. Let's take a look at a typical function. This one goes through and changes lines such as:

```
stdio.h
```

to lines which look like:

```
#include <stdio.h>
```

The function is fairly smart. It can tell the difference between system includes and user includes and insert the correct C statement for each type of include.

Let's start by listing out the contents of the function so we know what we are dealing with:

The Vim Tutorial and Reference

```
:function Include
function Include()
1 :      " Get the current line
2 :      let l:line = getline(".")
3 :
4 :      let l:dir_list = split(&path, ",")
5 :      " Loop through the local dirs looking for the file
6 :      for l:cur_dir in l:dir_list
7 :          if (filereadable(l:cur_dir."/".l:line))
8 :
9 :              " System directory?
10 :             if (match(l:cur_dir, "/usr/include") == 0)
11 :
12 :                 " Put the #include in the right place
13 :                 let l:line = "#include <".l:line.">"
14 :             else
15 :                 " Put the #include in the right place
16 :                 let l:line = "#include \"\"\".l:line.\"\"\""
17 :             endif
18 :
19 :             call setline(".", l:line)
20 :             return
21 :         endif
22 :     endfor
23 :
24 :     "At this point we did not find anything
25 :     "We could put in a default
endfunction
```

In order to understand what this function does we would like to single step through it. So the first thing we do is use the **:breakadd** (**:breaka**) command to add a breakpoint at the beginning of the function.

```
:breakadd func Include
```

In this example the **func** parameter tells Vim to stop at the first line of the function.

Next we need to start the debugger. The **:debug** (**:deb**) command tells *Vim* to run the following command, but do so inside the debugger. Without the **:debug**, we'd just execute the function normally.

```
:debug call Include()
Entering Debug mode.  Type "cont" to continue.

cmd: call Include()
```

The Vim Tutorial and Reference

```
Breakpoint in "Include" line 1
function Include
line 2: let l:line = getline(".")
>
```

Vim now starts calling the function `Include()` and encounters our breakpoint. So it stops on the first line of code in the function.

Now that we are stopped we can enter debugging commands, or normal *Vim* commands for that matter.

The debug command `step` single steps through the code, stepping into any functions called along the way. If we wanted to step over the function calls, we need the debugging command `next`.

Let's see what happens if we enter `step`.

```
>step
function Include
line 4: let l:dir_list = split(&path, ",")
```

The editor executes the next statement.

Now let's examine one of the function's variables. This is done with the `:echo` command:

```
>:echo l:line
stdio.h
```

Let's now take a look at the `l:dir_list` variable. Since this is an array, the result looks a little different.

```
>echo l:dir_list
[ '.', '/usr/include', '' ]
```

Next we take a look at what breakpoints we have with the `:breaklist` (`:break1`) command.

```
>:breaklist
1 func Include line 1
```

The next interesting point for us in this function is line 10 where we do the `match()` call. Let's put a breakpoint there. The following command tells *Vim* to put a breakpoint in a function (`func`) named `Include` at line `10`:

```
>:breakadd func 10 Include
```

The Vim Tutorial and Reference

Now we continue execution until we hit the breakpoint:

```
>cont

Breakpoint in "Include" line 10
function Include
line 10: if (match(l:cur_dir, "/usr/include") == 0)
```

From here on things just work so we have no more need of debugging.

Other debugging commands

The **:breakadd** command has both a function and a file. The file forms looks like:

```
:breakadd file file-name
:breakadd file line file-name
```

The first form sets a breakpoint that is triggered when a file is loaded. The second when a specific line of a file is executed. These commands are useful for debugging *Vim* command files and the code that exists outside functions.

There's one more **:breakadd** command to discuss:

```
:breakadd here
```

This sets a breakpoint at the current location. (Which you've arrived at through **step** and **next** commands.)

To remove a breakpoint use the **:breakdel** (**:breakd**) command. It has the same syntax as the **:breakadd** command:

```
:breakdel func name
:breakdel func line name
:breakdel file name
:breakdel file line name
:breakdel here
```

You can also delete a breakpoint by breakpoint number:

```
:breakdel number
```

Finally to delete all breakpoints, use the command:

```
:breakdel *
```

Inside the debugger you can type the following commands:

step Single step going into functions.

The Vim Tutorial and Reference

next	Single step treat function calls as a single step.
cont	Continue execution until the next breakpoint.
quit	Abort the current function or file being debugged.
interrupt	Pretend that the script was interrupted with CTRL-C. (Useful for debugging catch and finally blocks.)
finish	Continue execution of the current function or file, then go back to debug mode.

Finally we have the **:debugready** (**:debugg**) command. Normally the debugger reads commands from the console. This makes it impossible to put debug commands in a file. The **:debugready** command tells *Vim* to read debug commands from the input stream. The **:0debugready** command turns this feature off.

Redrawing the screen

Normally *Vim* does not redraw the screen until the current function or script finishes. To force a redraw, use the **:redraw** (**:redr**) command. If you have an extreme problem with the screen **:redraw!** clears and redraws the screen.

To just redraw the status line, use the **:redrawstatus** (**:redraws**). By default this works on the current window only. To make it work on all windows use the override: **:redrawstatus!**.

Profiling a function

Vim has a built-in profiling feature. To start you must turn on profiling with the command:

```
:profile start file-name  
(:profile can be abbreviated :prof.)
```

This tells *Vim* to start profiling and to write the results out to the given file upon exit. Now at this point we told Vim to do profiling but haven't given it anything to profile. Let's tell it to profile our `Include()` function:

```
:profile func Include
```

The Vim Tutorial and Reference

Now when we execute this function *Vim* records the profile information. Let's execute the function once:

```
:call Include()
```

Next we exit with the ZZ command. The profile results are now available in the file we specified at the beginning of this process:

```
FUNCTION Include()
Called 1 time
Total time:  0.000244
Self time:  0.000244

count  total (s)    self (s)
      1          0.000017 :      " Get the current line
      1          0.000015 :      let l:line = getline(".")
file
      2          0.000011 :      let l:dir_list = split(&path, ",")
      2          0.000090 :      :      " Loop through the local dirs looking for the
                           :      for l:cur_dir in l:dir_list
                           :      if (filereadable(l:cur_dir."/".l:line))
                           :
                           :          " System directory?
                           :          if (match(l:cur_dir, "/usr/include") == 0)
                           :
                           :              " Put the #include in the right place
                           :              let l:line = "#include <".l:line.">"
                           :              else
                           :                  " Put the #include in the right place
                           :                  let l:line = "#include \"\"".l:line."\""
                           :              endif
                           :
                           :              0.000005 :      call setline(".", l:line)
                           :              0.000002 :      return
                           :
                           :              0.000003 :      endif
                           1          0.000003 :      endfor
                           :
                           :      :profile continue  "At this point we did not find
anything
                           :      :      "We could put in a default

FUNCTIONS SORTED ON TOTAL TIME
count  total (s)    self (s)  function
      1  0.000244      Include()

FUNCTIONS SORTED ON SELF TIME
count  total (s)    self (s)  function
      1          0.000244  Include()
```

This shows how many times each line was executed and how much time was spent in the line itself.

Other Profile Commands

To profile the code in a file, use the command:

```
:profile file file-name
```

If you want to profile a file and all the functions in a file use the command:

```
:profile! file file-name
```

To temporarily stop profiling use the command:

```
:profile pause
```

To start after a pause use the command:

```
:profile continue
```

Deleting Profile Items

If you wish to turn off profiling you can do so with any of the following commands:

```
:profdel *
:profdel number
:profdel func line function-name
:profdel file line file-name
:profdel here
```

(**:profdel** can be abbreviated **:prof.d.**)

User-Defined Commands

The *Vim* editor enables you to define your own commands. You execute these commands just like any other command-mode command. To define a command, use the **:command** (**:com**) command. For example:

```
:command DeleteFirst :1delete
```

Now when you execute the command

```
:DeleteFirst
```

the *Vim* editor performs a

```
:1delete
```

which deletes the first line.

Note: User-defined commands must start with a capital letter.

The Vim Tutorial and Reference

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

```
:command
```

To remove the definition of a user-defined command, issue the **:delcommand** (**:delc**):

```
:delcommand DeleteOne
```

You can clear all user-defined commands with the command **:comclear** (**:comc**):

```
:comclear
```

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

```
:command -nargs=0 DeleteFirst ldelete
```

However, because **-nargs=0** is the default, you do not need to specify it. The other values of **-nargs** are as follows:

-nargs=0 No arguments

-nargs=1 One argument

-nargs=* Any number of arguments

-nargs=? Zero or one argument

-nargs=+ One or more arguments

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

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

Now when you type

```
:Say Hello World
```

the system echoes

```
Hello World
```

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

The Vim Tutorial and Reference

-range	Range is allowed. Default is the current line.
-range=%	Range is allowed. Default is the whole file.
-range=count	Range is allowed, but it is really just a single number whose default is count.

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

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

Some of the other options and keywords are as follows:

-count=number	The command can take a count whose default is number. The resulting count is stored in the <count> keyword.
-bang	You can use the override (!) modifier. If present, a ! will be stored in the keyword <bang> .
-register	You can specify a register. (The default is the unnamed register.) The register specification is put in the <reg> (a.k.a. <register>) keyword .

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

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

is the same as executing the following command:

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

Finally, you have the **<lt>** keyword. It contains the character <.

The Operator Function

Another way to connect a function to a command is through the operator function. You start by setting the option '**operatorfunc**' ('**opfunc**') to the name of the function you wish to call. This function will be called when you use the **g@{motion}** command.

The function takes one argument which tells it if the motion was line or char (character) oriented. (It is also possible for it to be called with the block argument, but this is almost never done because **g@** is not useful in block visual mode.)

The Vim Tutorial and Reference

When the function is called the starting mark ('[) will be placed where the motion starts and the ending mark (']) will be located where it ends.

This is a highly specialized *Vim* feature and you may need to refer to the on-line help for details.

Built-In Functions

The *Vim* editor has a number of built-in functions. This section lists all the built-in functions.

append(*line_number*, *string*)

What it does: Appends the string as a new line after *line_number*.

Parameters:

line_number The line number after which the text is to be inserted. A value of 0 causes the text to be inserted at the beginning of the file.

string The string to be inserted after the given line.

Returns: Integer flag.

0 = no error; 1 = error caused by a *line_number* out of range.

argc()

What it does: Counts the number of arguments in the argument list.

Returns: Integer.

The argument count.

argv(*number*)

What it does: Returns an argument in the argument list.

Parameter:

number The argument index. Argument 0 is the first argument in the argument list (not the name of the program, as in C programming).

Returns: String.

Returns the requested argument.

```
browse(save, title, initial_directory, default)
```

What it does: Displays a file browser and lets the user pick a file. This works only in the GUI version of the editor.

Parameters:

<i>save</i>	An integer that indicates whether the file is being read or saved. If save is non-0, the browser selects a file to write. If 0, a file is selected for reading.
<i>title</i>	Title for the dialog box.
<i>initial_directory</i>	The directory in which to start browsing.
<i>default</i>	Default filename.

Returns: String.

The name of the file selected. If the user selected Cancel or an error occurs, an empty string is returned.

```
bufexists(buffer_name)
```

What it does: Checks to see whether a buffer exists.

Parameter:

<i>buffer_name</i>	The name of a buffer to check for existence.
--------------------	--

Returns: Integer flag.

Returns true (1) if the buffer exists and false (0) otherwise.

```
bufloaded(buffer_name)
```

What it does: Check to see whether a buffer is loaded.

Parameter:

<i>buffer_name</i>	The name of a buffer to check to see whether it is currently loaded.
--------------------	--

Returns: Integer flag.

Returns true (1) if the buffer is loaded and false (0) otherwise.

bufname(*buffer_specification*)

What it does: Find the indicated buffer.

Parameter:

buffer_specification A buffer number or a string that specifies the buffer. If a number is supplied, buffer number *buffer_specification* is returned. If a string is supplied, it is treated as a regular expression and the list of buffers is searched for a match and the match returned.

There are three special buffers: % is the current buffer, # is the alternate buffer, and \$ is the last buffer in the list.

Returns: String.

String containing the full name of the buffer or an empty string if there is an error or no matching buffer can be found.

bufnr(*buffer_expression*)

What it does: Obtains the number of a buffer.

Parameter:

buffer_expression

A buffer specification similar to the one used by the **bufname** function.

Returns: Integer.

Number of the buffer, or -1 for error.

bufwinnr(*buffer_expression*)

What it does: Obtains the window number for a buffer.

Parameter:

buffer_expression

A buffer specification similar to the one used by the **bufname** function.

Returns:

Integer. The number of the first window associated with the buffer or **-1** if there is an error or no buffer matches the *buffer_expression*.

byte2line(byte_index)

What it does: Converts a byte index into a line number.

Parameter:

byte_index The index of a character within the current buffer.

Returns:

Integer.

The line number of the line that contains the character at *byte_index* or **-1** if *byte_index* is out of range.

char2nr(character)

What it does: Converts a character to a character number.

Parameter:

character A single character to be converted. If a longer string is supplied, only the first character is used.

Returns:

Integer.

The character number. For example, **char2nr("A")** is 65. (The ASCII code for 'A' is 65.)

col(location)

What it does: Returns the column of the specified location.

Parameter:

location Is a mark specification (for example, '**x**') or "**.**" to obtain the column where the cursor is located.

Returns:

Integer.

The column where the mark or cursor resides, or 0 if there is an error.

```
confirm(message, choice_list, [default], [type])
```

What it does: Displays a dialog box giving the user a number of choices and returns what the user chooses.

Parameters:

message A prompt message displayed in the dialog box.

choice_list A string containing a list of choices. The newline ("\\n") separates each choice. The ampersand (&) is used to indicate an accelerator character.

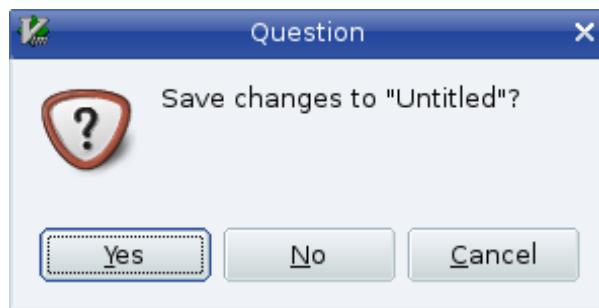
[default] An index indicating default choice. The first button is #1. If this parameter is not specified, the first button is selected.

[type] The type of the dialog box to be displayed. The choices are "Error", "Question", "Info", "Warning", or "Generic". The default is "Generic".

Returns: Integer.

The choice number (starting with 1), or 0 if the user aborted out of the dialog box by pressing <ESC> or CTRL-C.

Figure 27-1 shows the various types of confirmation dialog boxes.



GUI Confirmation

The Vim Tutorial and Reference

```
~  
~  
:confirm :q  
Save changes to "Untitled"?  
[Y]es, (N)o, (C)ancel:
```

Confirmation without a GUI

Figure 27-1: Dialog box types.

Note: This function works on both the GUI and console version of the editor. The console versions of the dialog boxes do not look as nice:

```
:echo confirm("Hello", "&One\n&Two", 1, "Error")  
Hello  
[O]ne, (T)wo: 0
```

```
delete(file_name)
```

What it does: Deletes the file.

Parameter:

file_name The name of the file to delete.

Returns: Integer.

0 means that the file was deleted. Non-0 for error.

```
did_filetype()
```

What it does: Checks to see whether the **FileType** event has been done. This command is useful in conjunction with the autocommands.

Returns: Integer.

Non-0 if autocommands are being executed and at least one **FileType** event has occurred. 0 otherwise.

```
escape(string, character_list)
```

What it does: Turns a {string} of characters into an escaped string. The {character_list} specifies the character to be escaped.

Parameters:

The Vim Tutorial and Reference

<i>string</i>	The string you want to escape.
<i>character_list</i>	The list of characters in <i>string</i> that need to have the escape character (\) put in front of them.
Returns:	String.
	The escaped string.

For example:

```
:echo escape("This is a 'test'.", " '")  
This\ is\ a\ \'test\'."
```

exists(*string*)

What it does:	Checks to see whether the item specified by <i>{string}</i> exists.
----------------------	---

Parameters:

<i>string</i>	An item to check. This can be used to specify an option ('&autoindent'), an environment variable ('\$VIMHOME'), a built-in function name ('*escape'), or a simple variable ('var_name'). (Note: The quotation marks are required because you are passing in a string.)
---------------	--

Returns: Integer.

Returns 1 if the item exists, 0 otherwise.

expand(*string*, [*flag*])

What it does:	Returns a list of files that match the <i>string</i> . This string can contain wildcards and other specifications. For example, <code>expand("*.*c")</code> returns a list of all C files.
----------------------	--

Parameters:

<i>string</i>	A string to be expanded. The string can contain any of the special file characters described earlier. Note: The ' suffixes ' ('su') and ' wildignore ' options affect how the expansion is carried out.
---------------	---

The Vim Tutorial and Reference

When a special word such as '`<cfile>`' is expanded, no further expansion is performed. If the cursor is on the string '`~/.vimrc`', for example, `expand('<cfile>')` results in '`./vimrc`'.

If you want the full filename, however, you need to expand twice. Therefore,
`expand(expand('<cfile>'))` returns
`/home/oualline/.vimrc`.

[flag]

The '`wildignore`' and '`suffixes`' options are honored unless a non-0 *flag* argument is supplied.

Returns:

String list.

A list of filenames that match the *string* separated by newlines. If nothing matches, an empty string is returned.

`filereadable(file_name)`

What it does: Checks to see whether a file is readable.

Parameter:

file_name

The name of the file to check.

Returns:

Integer.

Non-0 number is returned when the file exists and can be read. A 0 indicates that the file does not exist or is protected against reading.

`fnamemodify(file_name, modifiers)`

What it does: Applies the *modifiers* to the *file_name* and return the result.

Parameters:

file_name

The name of a file.

modifiers

Modification flags such as "`:r:h`".

Returns:

String.

The modified filename.

getcwd()

What it does: Obtains the current working directory.

Returns: String.

The current directory.

getftime(*file_name*)

What it does: Gets the modification time of a file.

Parameter:

file_name The name of the file to check.

Returns: Integer.

The modification time of the file (in UNIX format, seconds since January 1, 1970), or **-1** for error.

getline(*line_number*)

What it does: Gets a line from the current editing buffer.

Parameter:

line_number The line number of the line to get or **"."** for the line the cursor is on.

Returns: String.

The text of the line or an empty string if the *line_number* is out of range.

getwinposx() getwinposy()

What they do: Return the x or y position of the *Vim* GUI window.

Returns: Integer.

The location in pixels of the x or y position of the GUI window, or -1 when the information is not available.

glob(*file_name*)

What it does: Expands the wildcards in the filename and return a list of files.

Parameter:

file_name A string representing the filename pattern to match. You can also use an external command enclosed in backticks (`). For example

```
glob(`find . -name '*.c' -print`)
```

Returns: String list.

The list of files matched (separated by <NL>), or the empty string if nothing matches.

has(feature)

What it does: Checks to see whether a particular feature is installed.

Parameter:

feature A string containing the feature name.

Returns: Integer flag.

1 if the feature is compiled in, or 0 if it is not.

histadd(history, command)

What it does: Adds an item to one of the history lists.

Parameters:

history Name of the history to use. Names are as follows:

"cmd"	" : "	Command history
"search"	" / "	Search pattern history
"expr"	" = "	Expressions entered for "="
"input"	" @ "	Input line history

Returns: Integer flag.

1 for okay, 0 for error.

histdel(history, [pattern])

What it does: Removes commands from a history.

Parameters:

<i>history</i>	Which history list to use.
<i>[pattern]</i>	A regular expression that defines the items to be removed. If no pattern is specified, all items are removed from the history.

Returns:

Integer flag.
1 for okay, 0 for error.

histget(*history*, [*index*])

What it does: Obtains an item from the history.

Parameters:

<i>history</i>	Which history list to use.
<i>[index]</i>	An index of the item to get. The newest entry is -1 , the next newest -2 , and so on. The last entry is 1 , next to last 2 , and so on. If no <i>index</i> is specified, the last entry is returned.

Returns:

String.
The specified item in the history, or an empty string if there is an error.

histnr(*history*)

What it does: Returns the number of the current entry in the given history.

Parameter:

history The history to be examined.

Returns:

Integer.
The number of the last item in the history or **-1** for error.

hlexists(*name*)

What it does: Checks to see whether a syntax highlighting group exists.

Parameter:

name Name of the group to check for.

Returns: Integer flag Non-0, the group exists; 0 it does not.

h1ID(*name*)

What it does: Given the name of a syntax highlight group, returns the ID number.

Parameter:

name Name of the syntax highlighting group.

Returns: Integer.

The ID number.

hostname()

What it does: Gets the name of the computer.

Returns: String.

The hostname of the computer.

input(*prompt*)

What it does: Asks a question and gets an answer.

Parameter:

prompt The prompt to be displayed.

Returns: String. What the user types in as a response.

isdirectory(*file_name*)

What it does: Tests to see whether *file_name* is a directory.

Parameter:

file_name The name of the item to be checked.

Returns: Integer flag.

1, it is a directory; 0 it is not a directory or does not exist.

libcall(*dll_name*, *function*, {*argument*})

What it does: Calls a function in a DLL file. (Microsoft Windows only).

Parameters:

dll_name Name of a shared library (DLL) file in which the *function* is defined.

function The name of the function

argument A single argument. If this argument is an integer, it will be passed as an integer. If it is a string, it will be passed as "char *".

Note: The function must return a string or NULL. A function that returns a random pointer might crash *Vim*.

Returns: String.

Whatever the function returns.

line(*position*)

What it does: Given a marker or other position indicator, returns the line number.

Parameter:

position The position marker. This can be a mark: '**x**', the current cursor location **.**, or the end-of-file **\$**.

Returns: Integer.

The line number or 0 if the marker is not set or another error occurs.

line2byte(*line_number*)

What it does: Converts a line number to a byte index.

Parameter:

line_number The line number to be converted. This can also be a mark (**x**), the current cursor position (**.**) or the last line in the buffer (**\$**).

Returns: Integer.

The Vim Tutorial and Reference

The byte index of the first character in the line starting with 1. An error results in a return of **-1**.

localtime()

What it does:	Returns the current time in the UNIX standard time format.
Returns:	Integer. The number of seconds past January 1, 1970.

maparg(name, [mode])

What it does:	Returns what a key is mapped to.
Parameters:	
<i>name</i>	The name of a <i>lhs</i> mapping.
<i>[mode]</i>	The mode in which the string is mapped. This defaults to "".
Returns:	String. The resulting mapping string. An empty string is returned if there is no mapping.

mapcheck(name, [mode])

What it does:	Checks to see whether a mapping exists.
Parameters:	
<i>name</i>	The name of a <i>lhs</i> mapping.
<i>[mode]</i>	The mode in which the string is mapped. This defaults to "".
Returns:	String. This returns any mapping that can match <i>name</i> . This differs slightly from the maparg function in that it looks at mappings for conflicting names. If you have a mapping for "ax," for instance, it will conflict with "axx".

For example:

```
:map ax Test  
:echo maparg("ax")  
Test  
:echo maparg("axx")  
:echo mapcheck("ax")  
Test  
:echo mapcheck("axx")  
Test
```

```
match(string, pattern)
```

What it does: Checks to see whether *string* matches *pattern*. Setting the '**ignorecase**' ('**ic**') option causes the editor to ignore upper/lowercase difference.

Parameters:

string String to check.

pattern Pattern to check it against. This pattern acts like '**magic**' is set.

Integer.

Returns: The index of the first character of *string* where *pattern* occurs. The first character is number 0. If nothing matches, a **-1** is returned.

```
matchend(string, pattern)
```

What it does: Similar to the match function, except that it returns the index of the character of *string* just after where *pattern* occurs.

```
matchstr(string, pattern)
```

What it does: Like the match function, but returns the string that matches.

Parameters:

string String to check.

pattern Pattern to check it against .This pattern acts like '**magic**' is set.

Returns: String. The part of {string} that matches or the empty string if nothing matches.

`nr2char(number)`

What it does: Turns a number into a character.

Parameter:

number The number of an ASCII character.

Returns: String of length 1.

The ASCII character for the number.

`rename(from, to)`

What it does: Renames a file.

Parameters:

from The name of the existing file.

to The name to which we want to rename the file.

Returns: Integer flag.

0 for success, non-0 for failure

`setline(line_number, line)`

What it does: Replaces the contents of line *line_number* with the string *line*.

Parameters:

line_number The number of the line to change.

line The text for the replacement line.

Returns: Integer flag.

0 for no error, non-0 if there was a problem.

`strftime(format, [time])`

What it does: Returns the time formatted according to the *format* string. The conversion characters that can be put in the string are determined by your system's *strftime* function.

Parameters:

The Vim Tutorial and Reference

<i>format</i>	The format string.
<i>[time]</i>	The time (in seconds since 1970) to be used. (Default is now).

Returns: String.
The time string containing the formatted time.

strlen(string)

What it does: Computes the length of a string.

Parameter:

<i>string</i>	The string whose length you want.
---------------	-----------------------------------

Returns: Integer.
The length of the string.

strpart(string, start, length)

What it does: Returns the substring of *string* that starts at index *start* and up to *length* characters long.

For example:

```
:echo strpart("This is a test", 0, 4)  
This  
:echo strpart("This is a test", 5, 2)  
is
```

If the *start* or *length* parameters specify non-existent characters, they are ignored. For example, the following **strpart** command starts to the left of the first character:

```
:echo strpart("This is a test", -2, 4)  
Th
```

Parameters:

<i>string</i>	The string from which you want a piece.
<i>start</i>	The location of the start of the string you want to extract.
<i>length</i>	The length of the string to extract. String.

Returns:	The substring extracted for <i>string</i> .
strtrans(<i>string</i>)	
What it does:	Translates the unprintable characters in {string} to printable ones.
Parameter:	
<i>string</i>	A string containing unprintable characters.
Returns:	String.
The resulting string has unprintable characters, such as CTRL-A translated to ^A .	
substitute(<i>string, pattern, replace, flag</i>)	
What it does:	In <i>string</i> , changes the first match of <i>pattern</i> with <i>replace</i> . This is similar to performing the following on a line containing <i>string</i> :
:. substitute /<i>pattern/replacement/flag</i>	
Parameters:	
<i>string</i>	The string in which the replacement is to be made.
<i>pattern</i>	A <i>pattern</i> used to specify the portion of <i>string</i> to be replaced.
<i>replacement</i>	<i>The replacement text.</i>
<i>flag</i>	If the empty string, replace just the first occurrence. If g , replace all occurrences.
Returns:	String.
The string that results from the substitution.	
synID(<i>line, column, transparent_flag</i>)	
What it does:	Returns the syntax ID of the item at the given <i>line</i> and <i>column</i> .
Parameters:	
<i>line, column</i>	The location of the item in the buffer.

The Vim Tutorial and Reference

transparent_flag If non-0, transparent items are reduced to the items they reveal.

Returns: Integer.

The Syntax ID.

```
synIDattr(syntax_id, attribute, [mode])
```

What it does: Obtains an attribute of a syntax color element.

Parameters:

syntax_id The syntax identification number.

attribute The name of an attribute. The attributes are as follows:

name	The name of the syntax item
fg	Foreground color
bg	Background color
fg#	Foreground color in #RRGGBB form
bg#	Background color in #RRGGBB form
bold	"1" if this item is bold
italic	"1" if this item is italic
reverse	"1" if this item is reverse
inverse	Same as "reverse"
underline	1" if this item is underlined

mode "Which type of terminal to get the attributes for. This can be **gui**, **cterm**, or **term**. It defaults to the terminal type you are currently using.

Returns: String.

The value of the attribute.

```
synIDtrans(syntax_id)
```

What it does: Returns a translated syntax ID.

Parameter:

syntax_id The ID of the syntax element.

Returns: Integer.

The translated syntax ID.

system(*command*)

What it does: Executes the external command specified by *command* and captures the output. The options '**shell**' ('**sh**') and '**shellredir**' ('**ssr**') apply to this function.

Parameter:

command A command to execute.

Returns: String. Whatever the command output is returned.

tempname()

What it does: Generates a temporary filename.

Returns: String.

The name of a file that can be safely used as a temporary file.

visualmode()

What it does: Gets the last visual mode.

Returns: String.

The last visual mode as a command string. This is either **v**, **V**, or **CTRL-V**.

virtcol(*location*)

What it does: Computes the virtual column of the given *location*.

Parameter:

location A location indicator such as . (cursor location), '**a**' (mark a), or '\$' (end of the buffer).

Returns: Integer.

The location of the virtual column (that is, the column number assuming that tabs are expanded and unprintable characters are made printable).

winbufnr(*number*)

What it does: Gets the buffer number of the buffer that is in a window.

Parameter:

number The window number or 0 for the current window.

Returns: Integer.

Buffer number or -1 for error.

winheight(*number*)

What it does: Gets the height of a window.

Parameter:

number The window number or 0 for the current window.

Returns: Integer.

Window height in lines or -1 for error.

winnr()

What it does: Gets the current window number.

Returns: Integer.

The current window number. The top window is #1.

Obsolete Functions

A few functions are currently obsolete and have been replaced with newer versions; however, you might still find some scripts that use them.

<i>Obsolete Name</i>	<i>Replacement</i>
<code>buffer_exists()</code>	<code>bufexists()</code>
<code>buffer_name()</code>	<code>bufname()</code>
<code>buffer_number()</code>	<code>bufnr()</code>
<code>last_buffer_nr()</code>	<code>bufnr("\$")</code>
<code>file_readable()</code>	<code>filereadable()</code>
<code>highlight_exists()</code>	<code>hlexists()</code>
<code>highlightID()</code>	<code>hIID()</code>

Plugins and other scripts

To read in a script file you use the command `:source (:so)` and the script name:

The Vim Tutorial and Reference

```
:source my-file.vim
```

There are a number of files that *Vim* reads in automatically. Any file that is in the `$HOME/.vim/plugin` directory is considered a plugin and read in automatically.

The '**runtimopath**' ('**rtp**') option can also be set to a series of directories in which to look for plugins.

Vim has a couple of funny syntax elements that help avoid name collisions with plugins or any other script. The **<SID>** string will be translated into a magic character (named **<SNR>**) a unique serial number and an underscore. So putting **<SID>** in front of all your variables and functions in a script assures that they will all have a unique name.

Chapter 28: Customizing the Editor

The *Vim* editor is highly customizable. It gives you a huge number of options.

This chapter discusses how to use the ones that enable you to customize the appearance and the behavior of your *Vim* editor.

This chapter discusses the following:

- The **:set** command (in extreme detail)
- Local initialization files
- Customizing keyboard usage
- Customizing messages and the appearance of the screen
- Other miscellaneous commands

Setting

The *Vim* editor has a variety of ways of setting options. Generally, to set an option, you use the **:set** (**:st**) command:

```
:set option=value
```

This works for most options. Boolean options are set with this command:

```
:set option
```

They are reset with the following command:

```
:set nooption
```

To display the value of an option, use this command:

```
:set option?
```

If you want to set an option to its default value, use the following command:

```
:set option&
```

Boolean Options

You can perform the following operations on a Boolean option.

The Vim Tutorial and Reference

<i>Operation</i>	<i>Meaning</i>
<code>:set {option}</code>	Turn the option on.
<code>:set no{option}</code>	Turn the option off.
<code>:set {option}!</code>	Invert the option.
<code>:set inv{option}</code>	Invert the option
<code>:set {option}&</code>	Set the option to the default value.

For example:

```
:set list
:set list?
list

:set nolist
:set list?
nolist

:set list!
:set list?
list

:set list&
:set list?
nolist
```

Numeric Options

You can perform the following operations on a numeric option.

<i>Command</i>	<i>Meaning</i>
<code>:set option += value</code>	Add value to the option.
<code>:set option -= value</code>	Subtract value from the option.
<code>:set option ^= value</code>	Multiply the option by value.
<code>:set option&</code>	Set the option to the default value.

For example:

```
:set shiftwidth=4  
:set shiftwidth += 2  
:set shiftwidth?  
shiftwidth=6  
  
:set shiftwidth-=3  
:set shiftwidth  
shiftwidth=3  
  
:set shiftwidth ^= 2  
:set shiftwidth  
shiftwidth=6  
  
:set shiftwidth&  
:set shiftwidth  
shiftwidth=8
```

String-Related Commands

You can perform the following operations on string options:

<i>Command</i>	<i>Meaning</i>
:set option += value	Add value to the end of the option.
:set option -= value	Remove value (or characters) from the option.
:set option ^= value	Add value to the beginning of the option.

For example:

```
:set cinwords=test  
:set cinwords?  
cinwords=test  
  
:set cinwords+=end  
:set cinwords?  
cinwords=test,end  
  
:set cinwords-=test  
:set cinwords?  
cinwords=end  
  
:set cinwords^=start  
:set cinwords?  
cinwords=start,end
```

Another Set Command

The following command sets a Boolean option (such as '**list**' and '**nolist**'), but it displays the value of other types of options:

```
:set option
```

However, it is not a good idea to use this form of the command to display an option value because it can lead to errors if you are not careful. It is much better to use the following command to display the value of an option:

```
:set option?
```

An alternative form of the

```
:set option = value
```

command is this command:

```
:set option:value
```

Other :set Arguments

Error: Reference source not foundError: Reference source not foundTheError: Reference source not foundError: Reference source not found following command prints out all the options that differ from their default values:

```
:set
```

The following command prints all options:

```
:set all
```

The Vim Tutorial and Reference

This command prints out all the terminal control codes:

```
:set termcap
```

Finally, to reset everything to the default values, use this command:

```
:set all&
```

Chaining Commands

You can put several :set operations on one line. To set three different options, for example, use the following command:

```
:set list shiftwidth=4 incsearch
```

Automatically Setting Options in a File

You can put *Vim* settings in your files. When *Vim* starts editing a file, it reads the first few lines of the file, looking for a line like this:

```
vim: set option-command option-command option-command .... :
```

This type of line is called a modeline. In a program, for instance, a typical modeline might look like this:

```
/* vim: set shiftwidth=4 autoindent : */
```

An alternate format is:

```
vim: option-command:.option-command: ....:
```

The option '**modeline**' ('**ml**') turns on and off this behavior. The '**modelines**' ('**mls**') option controls how many lines are read at the start and end of the file when *Vim* looks for setting commands.

If you set the following option, for instance, *Vim* does not look for modelines:

```
:set nomodeline
```

If the following option is set, *Vim* does look at the top and bottom of each file for the number of lines specified by the '**modeline**' option:

```
:set modeline
```

For example, you may see lines like the following at the end of many of the *Vim* help files:

The Vim Tutorial and Reference

```
vim:tw=78:ts=8:sw=8:
```

This sets the '**tw**' ('**textwidth**') option to 78, the '**ts**' ('**tabstop**') to 8, and the '**sw**' ('**shiftwidth**') to 8. These settings make the text in the help files look nice. By using modelines, the creators of the help file make sure that the text is formatted correctly no matter what local settings you use for your other files.

Another example: For this book, I have had to create a number of C programming examples. When I copy these programs into the word processor, the tabs get really screwed up. The solution to this problem is to make sure that there are no tabs in the program. One way to do this is to put a line like this at the end of the file:

```
/* vim: set expandtab : */
```

This turns on '**expandtab**' and causes *Vim* to never insert a real tab--well, almost never; you can force the issue by using **CTRL-V <Tab>**. If you have some custom settings for your own C programs, you can put a line near the top or bottom of your program like this:

```
/* vim: set cindent shiftwidth=4 smarttabs : */
```

Local .vimrc Files

Suppose you want to have different settings for each directory. One way to do this is to put a *.vimrc* or *.gvimrc* file in each directory. That is not enough, however, because by default *Vim* ignores these files. To make *Vim* read these files, you must set the '**exrc**' ('**ex**') option:

```
:set exrc
```

Note: The *.vimrc* and *.gvimrc* files are read from the current directory, even if the file being edited is located in a different directory.

Setting this option is considered a security problem. After all, bad things can easily be dumped into these files, especially if you are editing files in someone else's directory.

To avoid security problems, you can set the '**secure**' option using this command:

```
:set secure
```

This option prevents the execution of the **:autocommand**, **:write**, and **:shell** commands from inside an initialization file.

Customizing Keyboard Usage

The *Vim* editor is highly customizable. This section shows you how to fine-tune the keyboard usage so that you can get the most out of your editor.

Microsoft Windows

Most programs that run under Microsoft Windows use the Alt keys to select menu items. However, *Vim* wants to make all keys available for commands. The '**winaltkeys**' ('**wak**') option controls how the Alt keys are used.

If you use the following command, for example, all the Alt keys are available for command mapping with the **:map** command:

```
:set winaltkeys=no
```

Typing **ALT-F** will not select the file menu, but will instead execute the command **ALT-F** is mapped to. A typical mapping might be this:

```
:map <M-f> :write
```

(Remember: *Vim* "spells" ALT as M-, which stands for Meta.) If you use the following command, all the Alt keys will select menu items and none of them can be used for mapping:

```
:set winaltkeys=yes
```

The third option is a combination of yes and no:

```
:set winaltkeys=menu
```

In this mode, if an Alt key can be used for a menu, it is; otherwise, it is used for **:map** commands. So **ALT-F** selects the File menu, whereas you can use **ALT-X** (which is not a menu shortcut) for **:map** commands.

Two options control how *Vim* reads the keyboard when you are using the console version of *Vim* from an MS-DOS window. The '**conskey**' ('**consk**') option tells *Vim* to read characters directly from the console:

```
:set conskey
```

Do not set this option if you plan to use your *Vim* editor to read a script file from the standard input.

The '**bioskey**' ('**biosk**') option tells *Vim* to use the BIOS for reading the keyboard:

```
:set bioskey
```

The Vim Tutorial and Reference

Again, do not use this if you plan using a script file for a redirected standard in. By pointing *Vim* at the BIOS, you get faster response to **CTRL-C** and **Break** interrupts.

Customizing Keyboard Mappings

Most UNIX function keys send out a string of characters beginning with <Esc> when they are pressed. But a problem exists: the <Esc> key is used to end insert mode. So how do you handle function keys in insert mode?

The solution is for *Vim* to wait a little after an <Esc> key is pressed to see whether anymore characters come in. If they do, *Vim* knows that a function key has been pressed and acts accordingly. To turn on this feature, execute the set the 'esckeys' option:

```
:set esckeys
```

But what about other key sequences? These are controlled by the '**timeout**' ('**to**') and '**ttimeout**' options:

```
:set timeout  
:set ttimeout
```

The following table shows the effects of these settings.

timeout	ttimeout	Result
notimeout	nottimeout	Nothing times out.
timeout	N/A	All key codes (<F1>, <F2>, and so on) and :map macros time out.
notimeout	ttimeout	Key codes (<F1>, <F2>, and so on) only time out.

The option '**timeoutlen**' ('**tm**') determines how long to wait after <Esc> has been pressed to see whether something follows. The default is as follows, which equals one second (1000 milliseconds):

```
:set timeoutlen=1000
```

Generally, '**timeoutlen**' controls how long to wait for both function keys and keyboard mapping strings. If you want to have a different timeout for keyboard mapping strings, use the '**ttimeoutlen**' ('**ttm**') option:

```
:set ttimeoutlen=500
```

The Vim Tutorial and Reference

These two timeouts tell *Vim* to wait 1/2 second after an <Esc> key has been pressed to see whether you have a function key or one second to see whether it is a keyboard mapping. (In other words, if *Vim* reads enough to determine that what comes after the <Esc> press cannot possibly be a keyboard mapping sequence, it will wait only one second between characters trying to figure out what function key has been typed.)

Confirmation

Generally, when you do something that *Vim* considers questionable, such as quitting from a modified buffer, the command fails. If you set the '**confirm**' ('**cf**') option, however, and use the following command, *Vim* displays a confirmation dialog box instead of failing:

```
:set confirm
```

When you try to :**quit** a buffer containing a modified file, for example, the *Vim* editor displays a confirmation dialog box (see Figure 28-1).

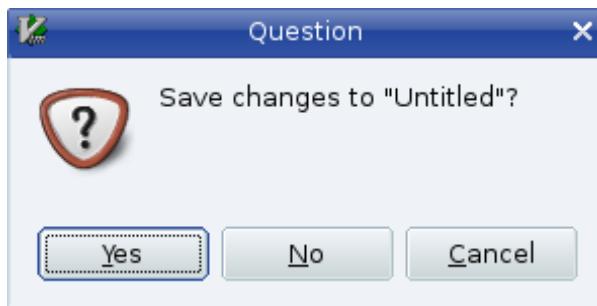


Figure 28-1: Confirmation dialog box.

Customizing Messages

Vim generally uses the bottom line of the screen for messages. Sometimes these messages exceed one line and you get a prompt that states something like Press Return to Continue. To avoid these prompts, you can increase the number of message lines by setting the '**cmdheight**' ('**ch**') options. To change the height of the message space to 3, for instance, use this command:

```
:set cmdheight=3
```

Showing the Mode

When you set the '**showmode**' ('**smd**') option, the *Vim* editor displays the current mode in the lower-left corner of the screen. To enable this feature, use the command:

```
:set showmode
```

Showing Partial Commands

If you set the '**showcmd**' ('**sc**') option, any partial command is displayed at the lower-right of the screen while you type it. Suppose you execute the following command:

```
:set showcmd
```

Now you enter an **fx** command to search for **x**. When you type the **f**, an **f** appears in the lower-right corner.

This is nice for more complex commands because you can see the command as it is assembled. For example, the command displays the entire command (incomplete as it is) in the lower-right corner: "**y2f**".

Figure 28-2 shows how '**cmdheight**', '**showmode**', and '**showcmd**' affect the screen.

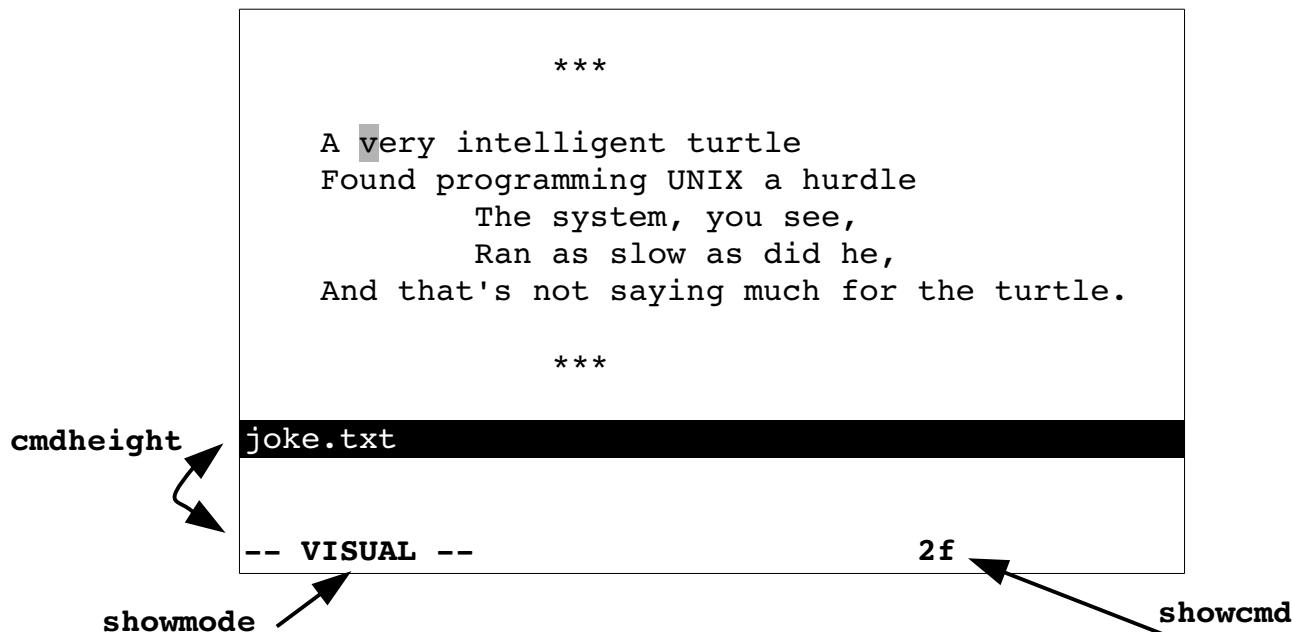


Figure 28-2: '**showmode**' and '**showcmd**'.

Short Messages

Another way to limit the "Press Return" prompts is to set the "short message" option. This shortens many common messages. The flags in the '**'shortmess'** ('**shm**') option determine which messages are shortened. The general form of this command is as follows:

```
:set shortmess=flags
```

The following table lists the flags.

Flag	Short Value	Long Value	Default
f	(3 of 5)	(file 3 of 5)	On
i	[noeol]	[Incomplete last line]	On
l	999L, 888C	999 lines, 888 characters	On
m	[+]	[Modified]	Off
n	[New]	[New File]	On
r	[RO]	[re adonly]	Off
w	[w]	written	Off
x	[dos]	[dos format]	On
x	[unix]	[unix format]	On
x	[mac]	[mac format]	On
a	All the abbreviations: filmnrwx .		
A	Eliminate the "attention" messages issued when <i>Vim</i> finds an existing swap file. (Default = off.)		
I	Eliminate the introduction screen. (Default = off.)		
o	Sometimes you will perform an operation that writes a file and then does something else that writes a message, such as executing a :wnext command.		
	If this option is not set, you will get two messages and will probably have to go through a "Press Return" prompt to see the second one.		
	If this option is set (the default), the first message is overwritten by the second.		
	(Default = on.)		
o	If you get a message stating that you are reading a file, it will overwrite any previous message.		
	(Default = on.)		
s	If set, do not issue a "Search Hit Bottom, Continuing at Top" or "Search Hit Top, Continuing at Bottom" message.		
	(Default = off.)		

Flag	Short Value	Long Value	Default
t		If set, truncate the filename at the beginning of the message if it is too long to fit on one line. Thus, a long filename, such as /home/oualline/writing/books/vim-book/editor-messages/my-replies/tuesday.txt, appears as <itor-messages/my-replies/tuesday.txt (or something similar).	
		(Default = off.) (Does not change the message in ex mode.)	
T		Truncate messages in the middle if they are too long. The deleted portion will appear as an ellipsis (...).	
		(Default = on.) (Does not apply to ex mode.)	
w		Drop "written" or "[w]" when writing a file.	
		(Default = off.)	

The '`terse`' Option

To set the '`terse`' option, issue the command:

```
:set terse
```

This command adds the **s** flag to the '`shortmess`' option. Setting '`noterse`' removes this flag.

The "File Modified" Warning

Generally, *Vim* warns when you do a :shell command and the file is modified before you return to *Vim*. If you want to turn off this option, execute the following command:

```
:set nowarn
```

Error Bells

When *Vim* gets an error, it just displays an error message. It is silent. If you are more audio-oriented than visually proficient, you might want to turn on the '`errorbells`' ('`eb`') option. This following command causes *Vim* to beep when there is an error:

```
:set errorbells
```

The Vim Tutorial and Reference

Beeping can sometimes disturb others in an office or classroom environment. An alternative to audio bells is a "visual" bell. When the '**visualbell**' ('**vb**') option is set, the screen flashes (everything will go into reverse video and back to normal quickly). To set this option, use the following command:

```
:set visualbell
```

Status Line Format

You can customize the status line. You can use the '**statusline**' ('**stl**') option to define your status line:

```
:set statusline=format
```

The format string is a `printf` line format string. A % is used to indicate a special field. For example, **%f** tells *Vim* to include the filename in the status line.

The command

```
:set statusline=The\ file\ is\ \"%f\"
```

gives you the following status line:

```
The file is "sample.txt"
```

You can specify a minimum and maximum width for an item. For example, the command tells *Vim* that the filename must take up 8 characters, but is limited to only 19:

```
:set statusline=%8.19f
```

Items are right-justified. If you want them left-justified, put a - just after the %. For example:

```
->%10.10f<-          ->%-10.10f<-
->    foo.txt<-          >foo.txt   <-
```

Numeric items are displayed with leading zeros omitted. If you want them, put a zero after the %. To display the column number, for instance, with leading zeros, use the following command:

```
:set statusline=%05.10c
```

Format	Type	Description
<code>%(...%)</code>		Define an item group. If all the items in this group are empty, the entire item group (and any text inside it) disappears.

The Vim Tutorial and Reference

Format	Type	Description
<code>%{n}* %{n}*</code>		Uses the highlight group usern for the rest of the line (or until another <code>%n*</code> is seen). The format <code>%0*</code> returns the line to normal highlighting. If the highlight group user1 is underlined, for example, the status line
		<pre>:set statusline=File:\ %1*%f%0*</pre>
		gives you the following status line: <pre>File: sample.txt</pre>
<code>%<</code>		Define a location where the status line can be chopped off if it is too long.
<code>%=</code>		Defines a location in the "middle" of the line. All the text to the left of this will be placed on the left side of the line, and the text to the right will be put against the right margin. For example:
		<pre>:set statusline=<-Left%=Right-></pre>
		results in <pre><-Left Right-></pre>
<code>%%</code>		The character <code>%</code> .
<code>%B</code>	Number	The number of the character under the cursor in hexadecimal.
<code>%F</code>	String	Filename including the full path.
<code>%H</code>	Flag	HLP if this is a help buffer.
<code>%L</code>	Number	Number of lines in buffer.
<code>%M</code>	Flag	<code>+</code> if the buffer is modified.
<code>%O</code>	Number	Byte offset in the file in hexadecimal form.
<code>%P</code>	String	The <code>%</code> of the file in front of the cursor.
<code>%R</code>	Flag	RO if the buffer is read-only.
<code>%V</code>	Number	Virtual column number. This is the empty string if equal to <code>%c</code> .
<code>%W</code>	Flag	PRV if this is the preview window.
<code>%Y</code>	Flag	File type
<code>a%</code>	String	If you are editing multiple files, this string is <pre>"({current} of {arguments})".</pre>
		For example: <pre>(5 of 18)</pre>
		If there is only one argument in the command line, this string is empty.
<code>%b</code>	Number	The number of the character under the cursor in decimal.
<code>%c</code>	Number	Column number.

The Vim Tutorial and Reference

Format	Type	Description
%f	String	The filename as specified on the command line.
%h	Flag	[Help] if this is a help buffer.
%l	Number	Line number.
%m	Flag	[+] if the buffer is modified.
%n	Number	Buffer number.
%o	Number	Number of characters before the cursor including the character under the cursor.
%p	Number	Percentage through file in lines.
%r	Flag	[RO] if the buffer is read-only.
%t	String	The filename (without any leading path information).
%v	Number	Virtual column number.
%w	Flag	[Preview] if this is a preview window.
%y	Flag	Type of the file as [type].
%{expr%}		The result of evaluating the expression expr .

The flag items get special treatment. Multiple flags, such as RO and PRV, are automatically separated from each other by a comma. Flags such as + and help are automatically separated by spaces.

For example:

```
:set statusline=%h%m%r
```

can look like this:

```
[help] [+] [RO]
```

Note: For the purpose of this example, we are ignoring the fact that we have done the improbable and modified a read-only buffer.

The '**fillchars**' ('**fcs**') option controls how extra space in the status lines (and other separation lines) is filled. It is a list of value from the following table:

diff:{char} Deleted lines in a diff window
fold:{char} Lines which indicate folds
slt:{char} Status line of the current window
stln:{char} Status line of non-current windows
vert:{char} Vertical lines separating windows.

The Vim Tutorial and Reference

Rulers

If you do not like the default status line, you can turn on the '**ruler**' ('**ru**') option:

```
:set ruler
```

This causes *Vim* to display a status line that looks like this:

help.txt [help][RO] 1,3-8 Top

After the file name and flags, this displays the current column, the virtual column, and an indicator showing you how far you are through the file. If you want to define your own ruler format, set the '**rulerformat**' ('**ruf**') to the desired format.

:set rulerformat=string

String is the same string used for the '**statusline**' option.

Reporting Changes

When you delete or change a number of lines, *Vim* tells you about it if the number of lines is greater than the value of the '**report**' option. Therefore, to report information on all changes, use the following command:

```
:set report=0
```

On the other hand, if you do not want to be told about what you have changed, you can set this to a higher value.

Help Window Height

You can set the minimum size of the help window by using the '**helpheight**' ('**hh**') command:

```
:set helpheight=height
```

This minimum is used when opening the help window. The height can get smaller afterwards .

Preview Window Height

You can also specify the height of the preview window by using the '**previewheight**' ('**pvh**') option:

```
:set previewheight=height
```

Defining How 'list' Mode Works

Generally, '**list**' ('**li**') uses **^I** for <Tab> and **\$** for the end of the line. You can customize this behavior. The '**listchars**' ('**lcs**') option defines how list mode works. The format for this command is as follows:

```
:set listchars=key:string,key:string,...
```

The possible values for the **key:string** pairs are:

eol:char

Define the character to be put after the end of the line.

tab:char1 char2

A tab is displayed as **char1** followed by enough **char2** to fill the width.

trail:char

Character for showing trailing spaces

extends:char

Character used at the end of a line that wraps to the next line in screen space.

For example:

```
:set listchars=tab:>-
```

shows up with tabs like this:

The Vim Tutorial and Reference

```
>-----Tabbing  
can>----be  
fun if you >----know how  
to set the list >----->-----command.
```

Another example:

```
:set listchars=tab:>-,trail:=
```

Gives us:

```
This line>=====  
has spaces and tabs>=====  
at the end=====  
of the line=====
```

Suppose that you have set the following options:

```
:set nowrap  
:set listchars=extends:+
```

Figure 28-3 displays the results.

```
The student technicians were used to t+  
technician took the back off a termina+  
loose chip and instead found a large h+  
board. He decided to talk to the prof+  
  
Technician: Did you pile papers on+  
  
Professor: Yes.
```

Figure 28-3: *listchars=extends:+*.

Changing the line number size

The '**numberwidth**' ('**nuw**') option controls how many characters are taken up by the line numbers which are displayed if 'number' is set. This is a minimum width. Vim will increase it if the number of lines in the file gets very large.

Changing the Highlighting

You can change the highlighting of various objects by using the '**highlight**' ('**hl**') option. The format for this option is as follows:

```
:set highlight=key:group, [key:group]....
```

The Vim Tutorial and Reference

Key is a key letter listed in the following table, and *group* is the name of a highlight group. The keys are:

Key	Default	Meaning
8	SpecialKey	This highlighting is applied when :map lists out a special key.
@	NonText	Applied to the ~ and @ character that Vim uses to display stuff that is not in the buffer.
M	ModeMsg	The mode information in the lower left of the screen. (See the ' showmode ' option.)
S	StatusLineNC	Status line for every window except the current one.
V	visualNOS	Text selected in visual mode when Vim does not own the selection.
W	WildMenu	Items displayed as part of a wildcard completion set.
d	Directory	Directories listed when you press CTRL-D .
e	ErrorMsg	Error messages.
i	IncSearch	Text highlighted as part of an incremental search.
l	Search	Text highlighted as part of a search.
m	MoreMsg	The -- More -- prompt.
n	LineNr	The line number printed by the :number command.
r	Question	The Press Return prompt and other questions.
s	StatusLine	The status line of the current windows.
t	Title	Titles for commands that output information in sections, such as :syntax, :set all, and others.
v	Visual	Text selected in visual mode.
w	WarningMsg	Warning

You can use a number of shorthand characters for highlighting, including the following:

r	Reverse
i	Italic
b	Bold
s	Standout
u	Underline
n	None
-	None

Therefore, you can specify that the error message use the highlight group **ErrorMsg** by executing the following command:

```
:set highlight=e:ErrorMsg
```

Or, you can use the shorthand to tell Vim to display error messages in reverse, bold, italic, by issuing this command:

```
:set highlight=erb
```

The Vim Tutorial and Reference

In actual practice, you would not define just one mode with a **:set highlight** command. In practice, this command can get quite complex. If a key is not specified, the default highlighting is used.

The 'more' Option

When the '**more**' option is set, any command that displays more than a screen full of data pauses with a **More** prompt. If not set, the listing just scrolls off the top of the screen.

The default is:

```
:set more
```

Number Format

The following command defines which types of numbers can be recognized by the **CTRL-A** and **CTRL-X** commands:

```
:set nrformats{octal,hex}
```

(Decimal format is always recognized.)

Restoring the Screen

When the '**restorescreen**' ('**rs**') option is set, *Vim* attempts to restore the contents of the terminal screen to its previous value:

```
:set restorescreen
```

In other words, it tries to make the screen after you run *Vim* look just like it did before you ran the program.

Pasting Text

The X Windows xterm program enables you to select text by drawing the mouse over it while the left button is held down. This text can then be "pasted" into another window. However, some of *Vim*'s capabilities can easily get in the way when pasting text into the window.

To avoid problems, you can set paste mode the '**paste**' option:

```
:set paste
```

This is shorthand for setting a number of options:

```
:set textwidth=0
```

The Vim Tutorial and Reference

```
:set wrapmargin=0  
:set noautoindent  
:set nosmartindent  
:set nocindent  
:set softtabstop=0  
:set nolisp  
:set norevins  
:set noruler  
:set noshowmatch  
:set formatoptions=""
```

At times, you might want paste mode and you might not. The '**pastetoggle**' option enables you to define a key that toggles you between '**paste**' mode and '**nopaste**' mode. To use the **<F12>** key to toggle between these two modes, for instance, use the command:

```
:set pastetoggle=<F12>
```

When '**paste**' mode is turned off, all the options are restored to the values they had when you set paste mode.

Wildcards

When you are entering a command in ex mode, you can perform filename completion. If you want to read in the file *input.txt*, for example, you can enter the following command:

```
:read input<Tab>
```

Vim will try to figure out which file you want. If the only file in your current directory is *input.txt*, the command will appear as:

```
:read input.txt
```

If several files that with the word *input*, the first will display. By pressing **<Tab>** again, you get the second file that matches; press **<Tab>** again, and you get the third, and so on.

To define which key accomplishes the wildcard completion, set the '**wildchar**' ('**wc**') command:

```
:set wildchar=character
```

If you are using filename completion inside a macro, you need to set the '**wildcharm**' ('**wcm**') (which stand for wild-char-macro). It is the character that accomplishes filename completion from inside a macro.

For example:

The Vim Tutorial and Reference

```
:set wildcharm=<F12>
:map <F11> :read in<F12>
```

Now when you press **<F11>**, it will start a read command for the file in-whatever.

You probably do not want to match backup file or other junk files. To tell *Vim* what is junk, use the '**wildignore**' ('**wig**') option:

```
:set wildignore=pattern, pattern
```

Every file that matches the given pattern will be ignored. To ignore object and backup files, for example, use the following command:

```
:set wildignore=*.o, *.bak
```

The '**suffixes**' ('**su**') option lists a set of file name suffixes that will be given a lower priority when it comes to matching wildcards. In other words if a file has one of these suffixes it will be placed at the end of any wildcard list. Generally, the filename completion code does not display a list of possible matches.

If you set the '**wildmenu**' ('**wmnu**') option

```
:set wildmenu
```

when you attempt to complete a filename, a menu of possible files displays on the status line of the window (see Figure 28-4).

```
This is a test
~
~
~
~
~
~
~
in.txt index.txt indoors.txt input.txt >
:read /tmp/in.txt
```

Figure 28-4: *Filename completion*.

The arrow keys cause the selection to move left and right. The **>** at the end of the line indicates that there are more choices to the right. The **<Down>** key causes the editor to go into a directory. The **<Up>** key goes to the parent directory. Finally, **<Enter>** selects the item.

The Vim Tutorial and Reference

You can customize the behavior of the file completion logic by using the '**wildmode**' ('**wim**') option. The following command causes *Vim* to complete only the first match:

```
:set wildmode=
```

If you keep pressing the '**wildchar**' key, only the first match displays. Figure 28-5 shows how this option works.

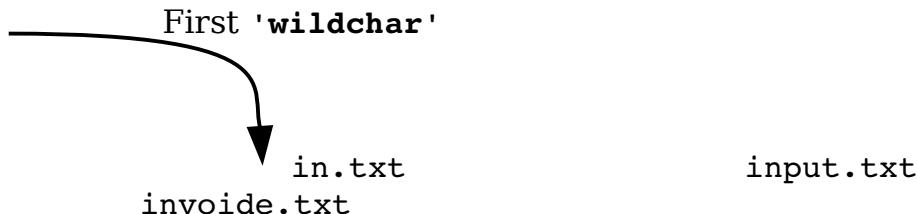


Figure 28-5: **wildmode=**.

The following command causes *Vim* to complete the name with the first file it can find:

```
:set wildmode=full
```

After that, if you keep pressing the '**wildchar**' key, the other files that match are gone through in order. Figure 28-6 shows what happens when this option is enabled.

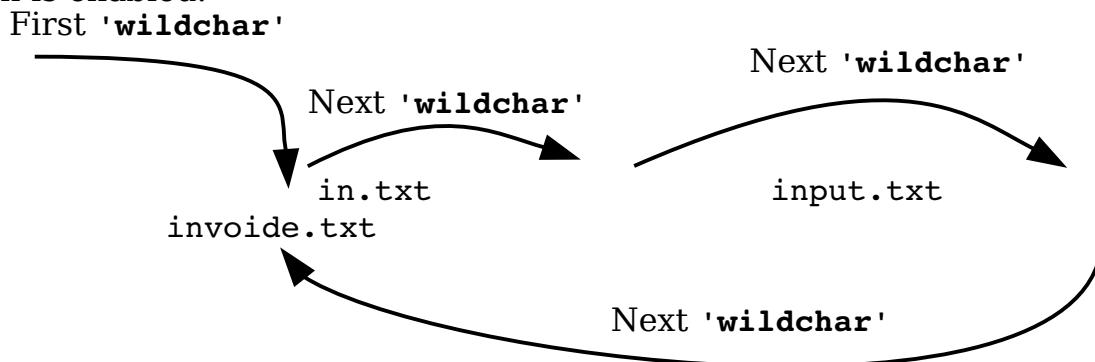


Figure 28-6: **wildmode= full**.

The following command causes the pressing of '**wildchar**' to match the longest common substring and then stop:

```
:set wildmode=longest
```

The Vim Tutorial and Reference

If you use the following command, you accomplish the same thing; but the display is just the list of files on the '**wildmenu**' line:

```
:set wildmode=longest:full
```

The following command displays a list of possible matches when the '**wildchar**' is pressed (see Figure 28-7).

```
:set wildmode=list
```

```
~  
~  
~  
~  
~  
~  
~  
  
:r g  
getchar.c      gui_beaval.c    gui_x11.c  
gui_at_sb.h    gui_xmebw.h  
gui.c          gui_gtk.c     gui_xmdlg.c  
gui_beaval.h   gui_gtk_f.c   gui_xmebw.c  
gui_at_fs.c    gui_gtk_f.h  
gui_gtk_f.h  
gui_at_sb.c    gui_gtk_x11.c  globals.h  
gui_x11_pm.h  
gui_athena.c   gui_motif.c  gui.h  
gui_xmebw.h  
  
:r g
```

Figure 28-7 *wildmode=list*.

This mode does not complete the match. If you want that to happen as well, use this option:

```
:set wildmode=list:full
```

Finally, to complete the longest common substring and list the files, use the following option :

```
:set wildmode=list:longest
```

You can use these options in a set. The first option is used the first time the '**wildchar**' is pressed, the second option is used the second time you press '**wildchar**', and so on, for up to four presses. Therefore, if you want to complete the longest substring (longest) and then go through the list (full), use the following option:

```
:set wildmode=longest,full
```

Customizing Behavior of the Screen Movement Commands

When the '**startofline**' ('**sol**') option is set, the screen movement commands as well as several cursor movement commands such as **H**, **M**, **L**, **G**, and **<C-End>** move the cursor to the start of a line. If it is not set, the cursor remains in the same column (or as close as possible).

File Writing Options

If set, the '**write**' option lets *Vim* write files. If this option is not set, you can view the file only. This is useful if you want to use *Vim* as a secure viewer.

Generally, when you try to write a file that you should not, *Vim* makes you use the override option (!). If you want to live dangerously, you can tell *Vim* to always assume that this option is present for writing type commands by executing setting the '**writeany**' ('**wa**'):

```
:set writeany
```

Memory Options

To set the maximum memory for one buffer, use the '**maxmem**' ('**mm**') option.

```
:set maxmem=size
```

size is the memory limit in kilobytes.

To define total amount of memory for all buffers, use the '**maxmemtot**' ('**mx**') option.

```
:set maxmemtot=size
```

To control the amount of memory used for pattern matching (in search) use the '**maxmempattern**' ('**mmp**') option.

Function Execution Options

The '**maxfuncdepth**' ('**mdf**') option defines the maximum number of nested functions. Similarly, the '**maxmapdepth**' ('**mmd**') parameter defines the maximum number of nested mappings.

Terminal Options

The following sections describe the terminal options.

Terminal Name

The name of your terminal is stored in the '**term**' option. ('**ttytype**' and '**tty**' are aliases for this option.) Generally, you do not need to set this option because it is set by your shell or operating environment. However, you might need to read it to enable terminal-specific macros.

Lazy Redraw

The '**lazyredraw**' ('**lz**') option is useful for a slow terminal. It also prevents *Vim* from redrawing the screen in the middle of a macro. The default is as follows:

```
:set nolazyredraw
```

If you do set this option, you do not see macros being executed.

Internal Termcap

The UNIX system has a database of terminal control codes called termcap. The *Vim* editor has its own built-in database as well. If the '**ttybuiltin**' ('**tbi**') option is enabled, this internal database is searched first.

Fast Terminals

If the '**ttyfast**' ('**tf**') option is set, *Vim* assumes you have a fast terminal connection and changes the output to produce a smoother update, but one with more characters. If you have a slow connection, you should reset this option.

Mouse Usage Inside a Terminal

The '**ttymouse**' ('**ttym**') option controls the terminal mouse codes. This option is of interest to those trying to do fancy things with terminal control codes. For example, if you want to use the mouse buttons **<LeftMouse>** and **<RightMouse>** in console editing, you should enable this option.

How Much to Scroll

The '**ttyscroll**' ('**tls**') option controls how many lines to scroll the screen when an update is required. You can adjust this to a small number if you are on a slow terminal.

Finally we have the '**weirdinvert**' ('**wiv**') option. This is a historical holdover from the 4.x version of *Vim*. If you happen to have an ancient terminal that requires this option, don't set the option. Donate the terminal to a museum and get a modern Linux system to replace it.

Some More Obscure Options

This section discusses some of the more obscure options in *Vim*. These were kept around for compatibility with *Vi* and to support equipment that was long ago rendered obsolete.

Compatibility

The '**compatible**' ('**cp**') option makes *Vim* act as much like *Vi* as possible:

```
:set compatible
```

If you enable this option, many of the examples in this book will not work properly. This option is generally set unless there is a `$HOME/.vimrc` file present.

Similarly, the '**coptions**' ('**cpo**') option enables you to fine-tune *Vi* compatibility:

```
:set coptions=characters
```

This '**edcompatible**' ('**ed**') command makes the **g** and **c** options on the **:substitute** command to act like they do for the UNIX editor Ed:

```
:set edcompatible
```

The following option sets lisp mode. This sets a number of options to make Lisp programming easier:

```
:set lisp
```

The '**lispwords**' ('**lw**') option contains a set of words that help Vim properly indent Lisp.

The Vim Tutorial and Reference

The '**tildeop**' ('**top**') option makes ~ behave like an operator. This is for *Vi* compatibility. If this option is turned off, the ~ command will switch the case of a single character. With the following, the ~ command takes the form of **~motion**:

```
:set tildeop
```

Note: The **g~** command always behaves as an operator regardless of this option.

The '**helpfile**' ('**hf**') option defines the location of the main help file. This option proves useful if you want to redirect where the **:help** command gets its information.

For example:

```
:set helpfile=/usr/sdo/vim/my_help.txt
```

Weirdinvert

The '**weirdinvert**' ('**wiv**') option has been provided for backward compatibility with version 4.0 of *Vim*:

```
:set weirdinvert
```

It has been made obsolete by the **t_xs** string. (See the *Vim* terminal help documentation for more information.) Some terminals, such as *hpterm*, need to have **t_xs** set to work. If you have one of these, you might want to look at the help text:

```
:help hpterm
```

Debugging

The '**writedelay**' ('**wd**') option causes a delay of time (in milliseconds) between each character output:

```
:set writedelay={time}
```

The '**verbose**' ('**vbs**') option controls how much *Vim* chatters while performing its job (the higher the number, the more output). The current numbers are as follows:

- >= 1** When the *viminfo* file is read or written.
- >= 2** When a file read because of a **:source** command.
- >= 5** Every searched tags file and include file.
- >= 8** Files for which a group of autocmds is executed.
- >= 9** Every executed autocmd.

The Vim Tutorial and Reference

- >= 12 Every executed function.
- >= 13 When an exception is thrown, caught, finished, or discarded.
- >= 14 Anything pending in a :finally clause.
- >= 15 Every executed ex command (truncated at 200 characters).

The :verbose (:verb) command does the same thing except it only affects a single command. The general form of this command is:

```
:[count] verbose {command}
```

Where [count] is the verbose number from above. This command is useful for debugging scripts to understand what's happening in them.

One of the problems with turning on verbose output is that things scroll off the screen quickly. One way to solve this problem is to set the 'verbosefile' ('vfile') option and save the output to a file. Then when you're done, you can browse through the file using Vim to look for anything you might have missed.

By default many functions that are called automatically such as those for 'foldexpr', 'formatexpr', and 'indentexpr' do not output error message. If you set the option 'debug' to msg, error messages which normally are suppressed.

They still won't throw an exception however. For that you need to set 'debug' to throw. Finally, if you want an audible notification of the error, add a beep.

You can use the three keywords, msg, throw, and beep in any combination you want such as:

```
:set debug=msg,beep
```

Production

The opposite of :verbose is :silent (:sil). It causes normal information messages produced during the execution of a command to disappear. If used with the override (!) option, error messages disappear as well.

Keyboard Mapping

The :loadkeymap (:loadk) load a keyboard mapping file into the system. These mapping files are highly keyboard and language dependent and beyond the scope of this book.

The Vim Tutorial and Reference

If you really must create a keyboard map file, you can examine the current set of mapping files which come with *Vim* as well as browse the on-line help documentation for more information.

Encoding

The '**encoding**' ('**enc**') option controls the character set encoding used by *Vim*. Unfortunately languages other English are beyond the scope of this book.

Macintosh Silliness

The '**macatsui**' is designed to work around a Macintosh drawing bug. Basically if something is going wrong with *Vim* on a Mac, try setting this. (As soon as they figure out what the bug is, this option goes away.)

Obsolete Options

The '**textauto**' ('**tx**') and '**textmode**' ('**tx**') options are obsolete. Use the options '**fileformats**' and '**fileformat**' instead.

Legacy Options

Vim tries to be as compatible with the old *Vi* editor as possible. *Vi* has a number of options that mean something to *Vi*, but are not relevant to *Vim*. In order to be fully compatible with *Vi*, the *Vim* editor won't generate an error message if you set any of these options. But the options themselves have no effect on the editor.

The options are:

autoprint	beautify	flash	graphic	hardtabs
mesg	novice	open	optimize	prompt
redraw	slowopen	sourceany	window	w300
w1200	w9600			

Chapter 29: Language-Dependent Syntax Options

Syntax coloring is controlled by language-dependent syntax files that reside `$VIMRUNTIME/syntax/language.vim`.

You can make your own copy of these syntax files and modify and update them if their syntax coloring is not what you desire.

You can also set a number of language-specific options. (This chapter covers the language-dependent syntax option for each language *Vim* knows about.) These options must be turned on before you edit the file. You can fake this by setting the option and then turning the syntax coloring off and back on.

Abel

There are only a couple of variable that control the Abel syntax highlighting:

`abelObsoleteOk` Obsolete keywords are statements, not errors
`abelCppCommentsIllegal` Do not interpret `//` as inline comment leader

Ada

Ada is a programming language designed by the United States Defense Department for embedded programming. There are several options which you can use to fine tune Ada editing. These are:

<code>g:ada_standard_types</code>	If this variable is set, highlight standard Ada types
<code>g:ada_space_errors</code>	Highlight spacing errors. These are not really errors, but they shouldn't be in a high quality program.
<code>g:ada_no_trail_space_error</code>	Do not highlight trailing spaces at the end of a line even if <code>g:ada_space_errors</code> is set
<code>g:ada_no_tab_space_error</code>	Do not highlight tabs after spaces.
<code>g:ada_all_tab_usage</code>	Highlight all tab use
<code>g:ada_line_errors</code>	Highlight lines which are too long. (And chew up a lot of CPU figuring out which lines to highlight.)
<code>g:ada_rainbow_color</code>	Use rainbow colors for parenthesis

The Vim Tutorial and Reference

g:ada_folding	This string variable controls how Ada folding is done. For information on this option see :help ft-ada-syntax
g:ada_abbrev	Create some abbreviations to help you in writing your Ada programs.
g:ada_withuse_ordinary	Treat with and use as ordinary keywords.
g:ada_begin_preproc	Show all begin line keywords using special coloring.
g:ada_omni_with_keywords	Add keywords to omni-completion (CTRL-I CTRL-U)
g:ada_extended_tagging	If set to jump then tags will be jumped to using :tjump . If set to list tags will be added to the error list.
g:ada_extended_completion	Use extended completion for CTRL-N and CTRL-R completions in insert mode.
g:ada_gnat_extensions	Support GNAT extensions
g:ada_with_gnat_project_files	Support GNAT project file keywords and attributes.
g:ada_default_compiler	This string tells <i>Vim</i> if the compiler is gnat or decada .

Ant

Ant is a tool for building Java program and just about everything else. One of the problems with Ant is that you can embed scripts written in other languages inside a Ant file. So not only do you have Ant, but you also have JavaScript, Perl, Python, and just about anything else.

By default the Ant syntax file correctly highlights JavaScript and Python. If you want to add another language, for example, Perl, you have to call the **AntSyntaxScript** function:

```
:call AntSyntaxScript('perl', 'perl.vim')
```

This must be done for each language you to embed in Ant except of course JavaScript and Python.

Apache

The variable **apache_version** should be set to the version of Apache you are using. The default is 1.3.x.

Assembly Language

There are a number of different assembly languages out there. By default, *Vim* assumes that you are using a GNU-style assembly language. The other assemblers supported are:

asm68k	Motorola 680x0 assembly
asm	GNU assembly (the default)
asmh8300	Hitachi H-8300 version of GNU assembly
fasm	Flat assembly (http://flat assembler.net)
ia64	Intel Itanium 64
masm	Microsoft assembly (probably works for any 80x86)
nasm	Netwide assembly
pic	PIC assembly (currently for PIC16F84)
tasm	Turbo Assembly (with opcodes 80x86 up to Pentium, and MMX)

To let *Vim* know you are using another assembly language, execute the following command:

```
:let asmsyntax=language
```

The language parameter is one of the languages listed above.

This command sets the global version of the variable. To set the buffer specific one (and change the syntax highlighting for a single buffer) use the command:

```
:let b:asmsyntax=language
```

You can also put a line like:

```
; :asmsyntax=nasm
```

in the first five lines of your code. (Use whatever comment marker is allowed by your assembler.)

The following variables control the optional parts of the assembly highlighting:

nasmloosesyntax	Do not highlight unofficial parser allowed syntax as an error.
nasm_ctx_outside_macro	Do not flag contexts outside a macro as errors.
nasm_no_warn	Do not flag potential risky syntax as TODOs.

ASP

Files that end with `.asp` and `.asa` can contain Perl or Visual Basic code. It's hard for Vim to automatically tell the difference between these two types of file. So you need to help things along by setting the variables: `g:filetype_asa` and `g:filetype_asp` to `aspperl` or `aspvbs` to let *Vim* know what to do.

BaaN

The following variables control the way Vim handles BaaN files:

<code>bann_code_stds</code>	Highlight code that violates coding standards
<code>bann_fold</code>	Enable folding at the function level
<code>baan_fold_block</code>	Enable folding at the block level
<code>baan_fold_sql</code>	Enable folding at the SQL statement level

Basic

Both Visual Basic and Standard Basic both use files that end in `.BAS`. To tell the difference between the two, the *Vim* editor reads the first five lines of the file and checks for the string `VB_Name`. (Files with the extension `.FRM` are always Visual Basic.)

C and C++

You can perform a number of customizations for the C and C++ syntax colors, including the following:

<code>c_comment_strings</code>	Highlight strings and numbers inside comments.
--------------------------------	--

```
/* Example a: "Highlighted String" */
```

The Vim Tutorial and Reference

c_ansi_constants

Highlight ANSI types. (If **c_no_ansi** set.)

```
/*:unlet c_ansi */
/*:unlet c_ansi_constants*/
size_t foo;
int i = INT_MIN;

/* :set c_no_ansi_constants = 1 */
size_t foo;
int i = INT_MIN;
```

c_ansi_typedefs

Highlight ANSI typedefs. (If **c_no_ansi** set.)

```
/*:unlet c_ansi */
/*:unlet c_ansi_typedefs */
size_t foo;
int i = INT_MIN;

/* :let c_ansi_typedefs = 1n */
size_t foo;
int i = INT_MIN;
```

c_gnu

c_no_bracket_error

c_no_curly_error

c_no_ansi

Highlight gcc specific items.

Do not flag {} inside [] as an error.

Don't flag {} inside () or [] as an error.

Do not highlight ANSI types and constants.

```
size_t foo; /* :unlet c_no_ansi */
int i = INT_MIN; /* :unlet c_no_ansi */

size_t foo; /* :let c_no_ansi = 1 */
int i = INT_MIN; /* :let c_no_ansi = 1 */
```

c_no_c99

c_no_comment_fold

Do not highlight C99 items.

Normally you can fold comments when the '**foldmethod**' is set to **syntax**. This disables that feature.

The Vim Tutorial and Reference

c_no_cformat

Do not highlight %-formats in strings.

```
/* :let c_no_cformat = 1 */
printf("Data %3f\n", f);

/* :unlet c_no_cformat */
printf("Data %3f\n", f);
```

c_no_if0_fold

Don't fold **#if 0 / #endif** if the '**foldmethod**' option is set to **syntax**. Do not flag spaces before a **<Tab>** when **c_space_error** is set. In the following we use "." for blank and ---> for tab.

```
int..---->foo;.....
```

c_no_if0

Do not highlight **#if 0 / #endif** blocks as comments.

Do not flag trailing whitespace when **c_space_error** is set . In the following we use "." for blank and ---> for tab.

```
int..---->foo;.....
```

c_no_utf

Highlight \u or \U in strings.

```
/* :let c_no_utf = 1*/
char *f = "\uFF + \Uffff";

/* :unlet c_no_utf */
char *f = "\uFF + \Uffff";
```

c_space_errors

Flag trailing whitespace and spaces in front of a **<Tab>**.

```
int..---->foo;.....
```

c_syntax_for_h

Use C (instead of C++) syntax highlighting for include files.

Sometimes you will notice some highlighting errors in comments or **#if 0 / #endif** blocks. You can fix these by redrawing the screen using the **CTRL-L** command. To fix them permanently, you need to increase the number of lines searched for syntax matches by using the following command:

The Vim Tutorial and Reference

```
:let c_minlines = number
```

The parameter *number* is the minimum number of lines to search. Setting this to a large number helps to eliminate syntax coloration errors.

Doxxygen

Doxxygen is an embedded documentation system for C and C++ programs. To enable Doxygen syntax highlighting you need to set the '**syntax**' option to **c.doxxygen**. This can be done through a :**set** command:

```
:set syntax=c.doxxygen
```

or by putting a modline in your program:

```
// vim:syntax=c.doxxygen
```

Another way of doing this for every C, C++ and Idl file is to set the variable: **g:load_doxxygen_syntax**.

The following variables control the Doxygen syntax coloring:

g:doxygen_enhanced_color	Use non-standard highlighting for Doxygen comments.
g:doxygen_enhanced_colour	Same thing for people who spell color funny.
doxygen_my_rendering	Disable rendering of HTML bold, italic and html_my_rendering underline.
doxygen_javadoc_autobrief	Set to 0 to disable javadoc autobrief colour highlighting.
doxygen_end_punctuation	Set to regexp match for the ending punctuation of brief

CH

CH is a C/C++ like interpreter. To tune the C syntax highlighting for it, set the variable: **ch_syntax_for_h**.

Chill

Chill is another language with a like syntax. The options for this program are listed in the following table:

chill_space_errors	Flag trailing whitespace and spaces in front of a <Tab>.
chill_comment_string	Highlight strings and numbers inside comments.
chill_minlines	Defines the number of lines searched for syntax highlighting.

Changelog

By default the Changelog syntax highlights spaces at the beginning of a line. To turn this off set the variable **`g:changelog_spacing_errors`** to 0.

COBOL

There are two versions of COBOL highlighting: fresh development and legacy. To enable legacy highlighting, use the following command:

```
:let cobol_legacy_code = 1
```

Cold Fusion

If you use Cold Fusion style comments, you'll want to set the variable: **`html_wrong_comments`**.

CSH / TCSH

Vim can't tell the difference between *csh* and *tcsh* files. So to help it you can set the variable **`filetype_csh`** to "csh" or "tcsh" (including the quotes).

CYNLIB

Cynlib files end with *.cc* and *.cpp* which make them very difficult to tell from C and C++ files. To tell *Vim* to prefer Cynlib syntax, set the variables: **`cynlib_cyntax_for_cc`** and **`cynlib_cyntax_for_cpp`**.

CWEB

Files that end in *.w* can be Progress or Cweb files. To help Vim know that they are Cweb files, set the variable: **`filetype_w`**.

Desktop

There is a standard highlighting syntax for *.desktop* and *.directory* files. To for Vim to adhere to the standard, set the variable: **`enforce_freedesktop_standard`**.

Dircolors

To highlight directory colors according to the Slackware standard set the variable: **dircolors_is_slackware**.

DocBook

DocBook files come in two flavors XML and SGML. Normally Vim will guess at the type, but if the variable **docbk_type** is set to “**sgml**” or “**xml**” *before the syntax file is loaded*, that flavor will always be set. (The quotes are part of the value.)

If Vim guesses wrong you can always manually set the file type using one of the following commands:

```
:set filetype=docbksgml  
:set filetype=docbkxml
```

DosBatch

If you are edit MS-DOS batch files you have my deepest sympathy. You also have a couple of options you can use. The **dosbatch_cmdextversion** should be set to 1 if you are using the Windows-NT command processor and 2 if you are using Windows 2000. The default is Windows 2000.

Also if you are using *.btm* files, you need to set the variable **g:dosbatch_syntax_for_btm** to let *Vim* know these are MS-DOS batch files.

Doxygen

See *C/C++* above.

DTD

DTD is usually case sensitive. To make it not case sensitive, use this command:

```
:let dtd_ignore_case = 1
```

The syntax highlighting flags unknown tags as errors. To turn off this feature, use the following command:

```
:let dtd_no_tag_errors = 1
```

The Vim Tutorial and Reference

The parameter entity names are highlighted using the **Type** highlight group with the **Comment** group. You can turn this off by using the following command:

```
:let dtd_no_parameter_entities=1
```

Eiffel

Eiffel is not case sensitive, but the standard style guidelines require the use of upper/lowercase. The syntax highlighting rules are designed to encourage you to follow the standard guidelines. To disable case checking, use the following command:

```
:let eiffel_ignore_case = 1
```

You can cause the syntax coloring to check for the proper capitalization for Current,Void, Result, Precursor, and NONE, by using this command:

```
:let eiffel_strict = 1
```

If you want to really check the syntax against the style guide, use the following command:

```
:let eiffel_pedantic = 1
```

You can use the lowercase versions of current, void, result, precursor, and none by setting **eiffel_lower_case_predef**, as follows:

```
:let eiffel_lower_case_predef = 1
```

To handle ISE's proposed new syntax, use the following command:

```
:let eiffel_ise = 1
```

For support of hexadecimal constants, use this:

```
:let eiffel_hex_constsnts = 1
```

ERLANG

ERLANG stands for ERicsson LANGuage. The syntax coloring has two options:

erlang_keywords
erlang_characters
erlang_functions

Disable the highlighting of keywords.
Disable the highlighting of special characters.
Disable built-in function highlighting

FlexWiki

The syntax file not only defines the syntax coloring for FlexWiki, but also sets a number of options to make editing easier. The only language dependent control is the variable **flexwiki_maps** which if set, allows you to move and down *display* lines with **j** and **k**.

Form

If you enable enhanced color mode, *Vim* makes it easier to differentiate between header and body statements in a program. To enable this feature set the variable **form_enhanced_color**.

Fortran

The following variables control how Fortran is highlighted:

fortran_dialect	Define the dialect (f95 , f90 , f77 , elf , F) of Fortran to be used.
fortran_free_source	For using the free format flavor of Fortran
fortran_fixed_source	Use the fixed format flavor of Fortran
fortran_have_tabs	Do not mark tabs as errors
fortran_fold	Allow syntax directed folding
fortran_fold_conditionals	Allow folding of conditional regions
fortran_fold_multilinecomment_s	Allow folding of multi-line comments
fortran_more_precise	If defined the syntax coloring will be more precise, but slower

FVWM

FVWM is a window manager. If you are editing configuration files for this program, you need to tell *Vim* the location of the color file using the following command:

```
:let rgb_file="/usr/X11/lib/X11/rgb.txt"
```

This example shows the location of the *rgb.txt* file that comes with Linux. Other systems may put it in */usr/lib* or other locations.

Haskell

Haskell is a literate programming language. The options that control the syntax highlighting for this language are:

<code>hs_allow_hash_operator</code>	Highlight # operators
<code>hs_highlight_boolean</code>	Highlight true and false as keywords
<code>hs_highlight_debug</code>	Highlight the names of debugging functions
<code>hs_highlight_delimiters</code>	Highlight delimiters.
<code>hs_highlight_more_types</code>	Treat uncommon types as keywords
<code>hs_highlight_types</code>	Treat primitive types as keywords
<code>lhs_markup</code>	Define the type of markup. This can be <code>none</code> for no markup or <code>tex</code> for TeX markup.

HTML

The HTML syntax file uses the following highlight tags:

<code>htmlTitle</code>	<code>htmlH1</code>	<code>htmlH2</code>
<code>htmlH3</code>	<code>htmlH4</code>	<code>htmlH5</code>
<code>htmlH6html</code>	<code>Boldhtml</code>	<code>BoldUnderline</code>
<code>htmlBoldUnderlineItalic</code>	<code>htmlUnderline</code>	<code>htmlUnderlineItalic</code>
<code>htmlItalic</code>	<code>htmlLink</code>	

If you want to turn off some of syntax coloring, use the following command:

```
:let html_no_rendering = 1
```

If you want to define your own colors for these items, put the color-setting commands in your `.vimrc` and use the following command:

```
:let html_my_rendering = 1
```

Some files contain `<!--` and `--!>` or `<!` and `!>` for comments. If you want these comments highlighted, use the following command:

```
:let html_wrong_comments = 1
```

Inform

Inform language options:

<code>inform_highlight_glulx</code>	Do highlighting for Glulx/Glk programs (as opposed to Z machine programs)
<code>inform_highlight_old</code>	Highlight for the older (before version 6.30) language
<code>inform_highlight_simple</code>	Do not highlight library symbols
<code>inform_suppress_obsolete</code>	Do not highlight obsolete keywords as errors.

IDL (Interface Definition Language)

The syntax for this language can be customized through the following variables:

idl_no_ms_extensions	Disable some of the Microsoft specific extensions
idl_no_extensions	Disable complex extensions
idlsyntax_showerror	Show IDL errors (can be rather intrusive, but quite helpful)
idlsyntax_showerror_soft	Use softer colors by default for errors

Java

The Java syntax has the following options:

`java_mark_braces_in_parens_as_errors`

If set, braces inside parentheses are flagged as errors (Java 1.0.2).
This is legal in Java 1.1.

`java_highlight_java_lang_ids`

Highlight all the identifiers in `java.lang.*`.

`java_highlight_functions = "indent"`

Set if function declarations are always indented.

`java_highlight_function = "style"`

Function declarations are not highlighted.

`java_highlight_debug`

Highlight debug statements (`System.out.println` and `System.err.println`).

`java_allow_cpp_keywords`

If set do not mark all C and C++ keywords as an error. Marking C/C++ keywords helps prevent you from using them, so your code is more portable to C or C++.

`java_ignore_javadoc`

Turn off highlighting for Javadoc.

The Vim Tutorial and Reference

java_javascript

Turn on highlighting for JavaScript inside Javadoc.

java_css

Turn on highlighting for CSS style sheets inside of Javadoc

java_vb

Turn on highlighting for VB scripts.

java_minlines

Number of lines to scan for figuring what syntax to highlight.

Lace

The specification of the language states that it is not case sensitive. Good style is case sensitive. If you want to turn off the "good style" case-sensitive feature, use the following command:

```
:let lace_case_insensitive=1
```

Lex

Lex files are divided up into major sections separated by lines consisting of %.%. If you write long Lex files, the syntax highlighting may not be able to find the %.%. To fix this problem you might need to increase the **minlines** syntax option by using a command such as this:

```
:syntax sync minlines = 300
```

Lisp

The option **g:lisp_instring** highlights data inside strings as if it were lisp. To turn on rainbow parent his (very useful in Lisp) set the variable **g:lisp_rainbow**.

Lite

Lite uses a SQL-like query language. You can enable highlighting of SQL inside strings with the following command:

```
:let lite_sql_query = 1
```

The Vim Tutorial and Reference

If you have large commands, you might want to increase the number of lines used for synchronizing the syntax coloring:

```
:let lite_minlines = 500
```

LPC

LPC files end with *.c* which *Vim* considers C files. To make them LPC files you need to set the variable: **lpc_syntax_for_c**. You can also use a mode line:

```
// vim:set ft=lpc:
```

The following variables can be used to customize LPC.

lpc_pre_v22	Color code for LPC v22 and earlier.
lpc_compat_32	Handle the LpMud 3.2 version of LPC.
lpc_use_lpc4_syntax	Color for version 4.0 and later.

LUA

LUA is another language who's syntax has changed from version to version. To customize the highlighting for a particular version use the variables:

:let lua_version = 4	Version 4
:let lua_version = 5 :let lua_subversion = 0	Version 5.0
:let lua_version = 5 :let lua_subversion = 1	Version 5.1

Mail

The variable **mail_minlines** controls the number of lines used for synchronization.

Make

In *Makefiles* commands are highlighted to make them stand out. If this results in confusion you can turn it off by setting: **make_no_commands**.

Maple

Maple V, by Waterloo Maple Inc., is a symbolic algebra language. It has many different packages that the user can selectively load. If you want to highlight the syntax for all packages, use the following command:

The Vim Tutorial and Reference

```
:let mvpkg_all = 1
```

If you want to enable specific packages, use one or more of the following options:

<code>mv_DEtools</code>	<code>mv_genfunc</code>	<code>mv_networks</code>	<code>mv_process</code>
<code>mv_Galois</code>	<code>mv_geometry</code>	<code>mv_numapprox</code>	<code>mv_simplex</code>
<code>mv_GaussInt</code>	<code>mv_grobner</code>	<code>mv_numtheory</code>	<code>mv_stats</code>
<code>mv_LREtools</code>	<code>mv_group</code>	<code>mv_orthopoly</code>	<code>mv_student</code>
<code>mv_combinat</code>	<code>mv_inttrans</code>	<code>mv_padic</code>	<code>mv_sumtools</code>
<code>mv_combstruct</code>	<code>mv_liesymm</code>	<code>mv_plots</code>	<code>mv_tensor</code>
<code>mv_difforms</code>	<code>mv_linalg</code>	<code>mv_plottools</code>	<code>mv_totorder</code>
<code>mv_finance</code>	<code>mv_logic</code>	<code>mv_powseries</code>	

Mathematica

New files ending in `.m` are assumed to be Mathlib files. To tell Vim that they should be Mathematica files, execute the command:

```
:let filetype_m = "mma"
```

Moo

The variables which control Moo highlighting are:

<code>moo_extended_cstyle_comments</code>	Allow C style comments
<code>moo_no_pronoun_sub</code>	Turn off highlighting of pronoun substitution patterns
<code>moo_no-regexp</code>	Disable highlighting of regular expression operator <code>%!</code> , <code>%(</code> and <code>%)</code> inside strings
<code>moo_unmatched_quotes</code>	Unmatched double quotes are errors
<code>moo_builtin_properties</code>	Highlight built-in property keywords
<code>moo_unknown_builtin_functions</code>	Highlight unknown built-in functions as errors

MSQL

To highlight SQL statements inside strings (stored procedures for example) set the variable: `msql_sql_query`. The only other option is `msql_minlines` which controls the number of lines to look through when doing syntax synchronization.

NCF

To highlight unknown statements as error set the variable:
`ncf_highlight_unknowns`.

Nroff

The variables that control the syntax highlighting for *nroff* type documents are:

b:nroff_is_groff	Nroff files use groff syntax
nroff_space_errors	Highlight spacing errors
b:preprocs_as_sections	Treat pre-processor entires as section markers

OCAML

Ocaml can be customized by two variables:

ocaml_revised	Support revised syntax with camlp4 preprocessor directives.
ocaml_noend_error	Do not highlight end statements as an error. (Useful if you define very long structures and Vim fails to synchronize properly.)

Papp

Normally HMLT inside Papp files is treated as a string. If you want to have it treated as HTML, set the variable: **papp_include_html**.

Pascal

By default files that end in *.p* can be Progress or Pascal. To force **vim** to consider all *.p* file as Pascal, put the following in your *.vimrc*.

```
:let filetype_p = "pascal"
```

The syntax highlighting for Pascal is configured through the following variables:

pascal_delphi	Highlight for the Delphi version of Pascal
pascal_fpc	Use the fpc variant of Pascal for highlighting
pascal_gpc	Configure the syntax highlighting for the gpc version of Pascal
pascal_no_functions	Do not highlight function differently
pascal_no_tabs	Highlight tabs as errors
pascal_one_line_string	Highlight multi-line strings as errors
pascal_symbol_operator	Highlight operators
pascal_traditional	Turn off highlighting of Turbo Pascal 7.0 features (stick to the traditional version)

Perl

If you include POD documentation in your files, you can enable POD syntax highlighting using the following command:

```
:let perl_include_pod = 1
```

If you do not include POD documents, you should. Perl is bad enough when it is documented.

The following option changes how Perl displays package names in references (such as \$PkgName::VarName):

```
:let perl_want_scope_in_variables = 1
```

If you do not want to use complex variable declarations such as @{\$ {"var"}}, and have the highlighted you need the following:

```
:let perl_no_extended_vars = 1
```

The following option tells the syntax coloring to treat strings as a statement:

```
:let perl_string_as_statement = 1
```

If you have trouble with synchronization, you might want to change some of the following options:

```
:let perl_no_sync_on_sub = 1  
:let perl_no_sync_on_global = 1  
:let perl_sync_dist = {lines}
```

To use folding with Perl, set the variable **perl_fold**. Folding is configured by the variables: **perl_fold_blocks** which allows folding inside blocks, **perl_nofold_packages** which turns off package folding and **perl_nofold_subs** which turns off subroutine folding.

PHP3/ PHP4

The following options control the highlighting for PHP:

php_asp_tags	ASP-style short tags get highlighted
php_baselib	Add highlighting for Baselib method
php_folding	Allows folding
php_htmlInStrings	Highlight HTML syntax inside strings
php_noShortTags	Do not highlight short tags
php_oldStyle	Use the older style of highlighting
php_parent_error_close	Highlight closing brace errors
php_parent_error_open	Highlight opening brace errors

The Vim Tutorial and Reference

php_sql_query
php_sync_method

Highlight SQL queries inside strings
Define how Vim synchronizes the highlighting. -1 uses a search method to start synchronization, 0 synchronizes from the start, any positive number defines a number of lines to go backward to begin synchronization.

PlainTex

The only option to PlainTex syntax highlighting is the variable **g:plaintex_delimiters** which turns on highlighting for the delimiters [] and {}.

PPWizard

If you set the variable **ppwiz_highlight_defs** to 1 (the default) **#define** statement retain the color of their contents. If you set this variable to 2, the these statements are highlighted using a single color.

Setting **ppwiz_with_html** to 1 (the default) highlights HTML inside strings. A setting of 0 treats all string contents as strings.

Phtml

To highlight SQL syntax in a string, use the following:

```
:let phtml_sql_query = 1
```

To change the synchronization window, use this command:

```
:let phtml_minlines = lines
```

PostScript

The options for PostScript highlighting are:

postscr_level
postscr_display
postscr_ghostscript
postscr_fonts
postscr_encodings

Set the PostScript language level (default = 2).
Highlight display postscript features.
Highlight GhostScript-specific syntax.
For font highlighting (off by default for speed).
Encoding tables (off by default for speed).

postscr_andornot_binary

Color logical operators differently.

Printcap and Termcap

If you are doing complex printcap or termcap work, I feel for you. These files are cryptic enough as is and dealing with long and complex ones is especially difficult. As far as *Vim* is concerned, you might want to increase the number of lines used for synchronization:

```
:let ptcap_minlines = 100
```

Progress

Progress files can end in *.w*, *.i*, or *.p*. To make *Vim* detects you Progress file as Progress files, you may wish to set some of the following variables:

```
:let filetype_w = "progress"  
:let filetype_i = "progress"  
:let filetype_p = "progress"
```

Python

The Python highlighting can be tuned by setting the following variables:

python_highlight_all	Turn on all possible Python highlighting
python_highlight_builtins	Highlight builtin functions
python_highlight_exceptions	Highlight standard exceptions
python_highlight_numbers	Highlight numbers
python_highlight_space_errors	Highlight trailing spaces

Quake

Quake comes in multiple flavors. To tune Vim for your flavor you should set one of the variables: **quake_is_quake1**, **quake_is_quake2**, or **quake_is_quake3**. You can set all three, and highlight for Quake Version 1+2+3, but as there is no such game, this may not be useful.

ReadLine

To highlight the *bash* extensions to ReadLine, set the variable:

readline_has_bash.

Rexx

You can adjust the number of lines used for synchronization with the following option :

```
:let rexx_minlines = lines
```

Ruby

The variables which control the Ruby syntax are:

ruby_fold	Allow folding based on the Ruby syntax
ruby_minlines	The minimum number of lines to use for syntax synchronization
ruby_no_comment_fold	Do not allow the folding of multi-line comments
ruby_no_expensive	Do not highlight an end statement in a way that matches it to the opening statement. (This is an expensive operation, hence the variable name.)
ruby_no_special_methods	Do not highlight significant methods of the kernel, module, and object.
ruby_operators	Highlight Ruby operators
ruby_space_errors	Highlight space errors

Scheme

By default Vim uses the R5RS version of Schema. To tell it that you are using MzScheme set the variable **is_mzscheme**.

SDL

To enable SDL-2000 keyword highlighting set the variable **sdl_2000**. To disable the older keywords set **sdl_no_96**.

Sed

To make tabs stand out, you can use the **:set list** option. You can highlight them differently by using the following command:

```
:let highlight_sedtabs = 1
```

Hint: If you execute a

```
:set tabstop = 1
```

as well, it makes it easy to count the number of tabs in a string.

SGML

If you set the **sgml_my_rendering** Vim assumes that you've already defined highlighting for SGML and does not overwrite your definitions. (Not syntax, just the colors for highlighting.)

The variable **sgml_no_rendering** disable SGML rendering.

Shell

The following options change the highlighting for shell scripts:

is_bash	The Bash version of the syntax is expected.
is_korn_shell	Assume the Korn Shell flavor of the syntax.
is_posix	Assume POSIX syntax.
is_sh	The old Borne shell syntax is highlighted.
sh_fold_enable	Enable syntax directed folding.
sh_minlines	Set the number of lines for synchronization
sh_maxlines	Limit the number of lines for synchronization (speeds things up).

Speedup

The options for Speedup are:

strict_subsections	Only highlight the keywords that belong in each subsection.
highlight_types	Highlight stream types as a type.
oneline_comments = 1	Allow code after any number of # comments.
oneline_comments = 2	Show code starting with the second # as an error (default).
oneline_comments = 3	If the line contains two or more # characters in it, highlight the entire line as an error.

TCsh

This is a super-set of csh, so all the csh rules apply. It also has the following customizations:

tcsh_backslash_quote	Do not highlight \" as an error
tcsh_minlines	Minimum number of lines for syntax synchronization

TeX

TeX is a complex language that can fool the syntax highlighting. If the editor fails to find the end of a **texZone**, put the following comment in your file:

The Vim Tutorial and Reference

```
%stopzone
```

The TeX options include:

tex_fold_enabled	Enable syntax directed folding
tex_no_error	Disable flagging of errors
tex_stylish	Indicate that your TeX files use @ commands

TinyFugue

To adjust the synchronization limit for TinyFugue files, use this option:

```
:let tf_minlines = lines
```

Vim

Even *Vim* has its own syntax files. These can be customized as well.

vimembedscript	If set to 1, embed scripting languages are highlighted. If set to 0, the syntax file for embedded scripting languages are not loaded. If not defined, the files are loaded, just not used.
vim_maxlines	Maximum lines for syntax synchronization
vim_minlines	Minimum lines for syntax synchronization
vimsyntax_noerror	Do not highlight errors

XF86Config

There are two different major versions of the Xfree86 configuration files. The variable **xf86conf_xfree86_version** should be set to 3 or 4 depending on which version you have.

Xml

Xml namespaces are highlighted by default. To turn off this feature, set the variable: **g:xml_namespace_transparent**.

To turn on syntax directed folding for Xml, set the variable:
g:xml_syntax_folding.

Chapter 30: How to Write a Syntax File

Suppose you want to define your own syntax file. You can start by taking a look at the existing syntax files in the `$VIMRUNTIME/syntax` directory. After that, you can either adapt one of the existing syntax files or write your own.

Basic Syntax Commands

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

```
:syntax clear
```

(`:sy` is the short form of the `:syntax` command.)

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

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

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

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

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

```
:syntax keyword group keyword .....
```

The group name is the name of a highlight group, which is used by the `:highlight` command for assigning colors. The keyword parameter is an actual keyword. Here are a few examples:

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

The Vim Tutorial and Reference

This example uses the group names `xType` and `xStatement`. By convention, each group name is prefixed by a short abbreviation for the language being defined. This example defines syntax for the `x` language (eXample language without an interesting name).

These statements cause the words `int`, `long`, and `char` to be highlighted one way and the words `if` and `endif` to be highlighted another way.

Now you need to connect the `x` group names to standard *Vim* names. You do this with the following commands:

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

(The `:highlight` command can be shortened to `:hi`.)

This tells *Vim* to treat `xType` like `Type` and `xStatement` like `Statement`.

The `x` language allows for abbreviations. For example, `n` and `next` are both valid keywords. You can define them by using this command:

```
:syntax keyword xStatement n[ext]
```

Defining Matches

Consider defining something a bit more complex. You want to match ordinary identifiers. To do this, you define a match syntax group. This one matches any word consisting of only lowercase letters:

```
:syntax match xIdentifier /[a-z]\+ /
```

Now define a match for a comment. It is anything from `#` to the end of a line:

```
:syntax match xComment /#.*/$ /
```

Note: The match automatically ends at the end of a line by default, so the actual command is as follows:

```
:syntax match xComment /#.*/
```

Defining Regions

In the example `x` language, you enclose strings in double quotation marks (""). You want to highlight strings differently, so you tell *Vim* by defining a region. For that, you need a region start (double quote) and a region end (double quote). The definition is as follows:

```
:syntax region xString start=/" end=/" /
```

The Vim Tutorial and Reference

The start and end directives define the patterns used to define the start and end of the region. But what about strings that look like this?

```
"A string with a double quote (\") in it"
```

This creates a problem: The double quotation marks in the middle of the string will end the string. You need to tell *Vim* to skip over any escaped double quotes in the string. You do this with the skip keyword:

```
:syntax region xString start=/"/ skip=\\"/ end=/"/
```

Note: The double backslash is needed because the string you are looking for is \" , so the backslash must be escaped (giving us \\").

Nested Regions

Take a look at this comment:

```
# Do it TODO: Make it real
```

You want to highlight TODO in big red letters even though it is in a comment. To let *Vim* know about this, you define the following syntax groups:

```
:syntax keyword xTodo TODO contained  
:syntax match xComment /#.*/ contains=xTodo
```

In the first line, the **contained** parameter tells *Vim* that this keyword can exist only inside another syntax element. The next line has a **contains=xTodo** parameter. This indicates that the **xTodo** syntax element is inside it. The results (**xTodo=underline**, **xComment=italic**) are as follows:

```
# Do it TODO: Make it real
```

Consider the following two syntax elements:

```
:syntax region xComment start=%.* end=/$/ contained  
:syntax region xPreProc start=#.* end=/$/ contains=xComment
```

You define a comment as anything from % to the end of the line. A preprocessor directive is anything from # to the end of the line. Because you can have a comment on a preprocessor line, the preprocessor definition includes a **contains=xComment** parameter. The result (**xComment=italic**, **xPreProc=underline**) is as follows:

The Vim Tutorial and Reference

```
#define X = Y % Comment  
int foo = 1;
```

But there is a problem with this. The preprocessor directive should end at the end of the line. That is why you put the `end=/$/` directive on the line. So what is going wrong? The problem is the contained comment. The comment start starts with `%` and ends at the end of the line. After the comment is processed, the processing of the preprocessor syntax continues. This is after the end of the line has been seen, so the next line is processed as well.

To avoid this problem and to force contained syntax from eating a needed end of line, use the `keepend` parameter. This takes care of the double end-of-line matching:

```
:syntax region xComment start=/.*/ end=/$/ contained  
:syntax region xPreProc start=/#.*/ end=/$/  
  \ contains=xComment keepend
```

You can use the `contains` parameter to specify that everything can be contained. For example:

```
:syntax region xList start="\[" end="]" contains=ALL
```

You can also use it to include a group of elements except for the ones listed:

```
:syntax region xList start="\[" end="]"  
  \ contains=ALLBUT,xString
```

Multiple Group Options

Some syntax is context-dependent. You expect `if` to be followed by `then`, for example. Therefore, a `then/if` statement would be an error. The *Vim* syntax parser has options that let it know about your syntax order.

For example, you define a syntax element consisting of `>` at the beginning of a line. You also define a element for the `KEY` string. In this case, the `KEY` is important only if it is part of a `>` line. The definition for this is as follows:

```
:syntax match xSpecial "^>" nextgroup=xKey  
:syntax match xKey "KEY" contained
```

The `nextgroup` option tells *Vim* to highlight `KEY`, but only if it follows `xSpecial`. The result (`xKey=italic`, `xSpecial=underline`) is:

The Vim Tutorial and Reference

```
KEY (Normal key, nothing special, no >)
>KEY Key follows the > so highlighted
```

However, the KEY must immediately follow the >. If there is a space present, the KEY will not be highlighted:

```
> KEY
```

If you want to allow whitespace between the two groups, you can use the '**skipwhite**' option :

```
:syntax match xSpecial ">" skipwhite nextgroup=xKey
:syntax match xKey "KEY" contained
```

This gives you:

```
> _____KEY
```

The **skipwhite** directive tells *Vim* to skip whitespace between the groups. There are two other related options. The **skipnl** option skips newline in the pattern. If it were present, you would get the following:

```
>
KEY
```

The '**skipempty**' option causes empty lines to be skipped as well:

```
>
KEY
```

Transparent Matches

The '**transparent**' option causes the syntax element to become transparent, so the highlighting of the containing syntax element will show through. For example, the following defines a string with all the numbers inside highlighted:

```
:syntax region xString start=/"/ skip=/\\\"/ end=/"/
\ contains=xNumbers,xSpecial
:syntax match xNumbers /[0-9]\+/- contained
```

Now add a special group that does not take on any special highlighting:

```
:syntax match xSpecial /12345/ transparent contained
\ contains=xNothing
```

The Vim Tutorial and Reference

This also has a contains argument to specify that **xNothing** can be contained. Otherwise, the contains argument **xString** would be used, and **xNumbers** would be included in **xSpecial**. The results (**xString=italic**, **xNumbers=underline**, **xSpecial=N.A.**) look like this:

```
"String 12 with number 45 in it 12345 "
```

Other Matches

The **oneline** option indicates that the region does not cross a line boundary. For example:

```
:syntax region xPreProc start=/^#/ end=/$/ oneline
```

Things now become a little more complex. Let's allow continuation lines. In other words, any line that ends with \ is a continuation line. The way you handle this is to allow the **xPreProc** syntax element to contain a continuation pattern:

```
:syntax region xPreProc start=/^#/ end=/$/ oneline  
  \ contains=xLineContinue  
:syntax match xLineContinue "\\"$ contained
```

In this case, although **xPreProc** is on a single line, the groups contained in it (namely **xLineContinue**) let it go on for more than one line.

This is what you want. If it is not what you want, you can call for the region to be on a single line by adding **excludenl** to the contained pattern. For example, you want to highlight "end" in **xPreProc**, but only at the end of the line. To avoid making the **xPreProc** continue on the next line, use **excludenl** like this:

```
:syntax region xPreProc start=/^#/ end=/$/  
  \ contains=xLineContinue,xPreProcEnd  
:syntax match xPreProcEnd excludenl /end$/ contained
```

Note that **excludenl** is placed before the pattern. Now you can still use **xLineContinue** as previously. Because it doesn't have **excludenl**, a match with it will extend **xPreProc** to the next line.

Match Groups

When you define a region, the entire region is highlighted according to the group name specified. To highlight the text enclosed in parentheses () with the highlight group **xInside**, for example, use the following command:

```
:syntax region xInside start=/(/ end=)/
```

The Vim Tutorial and Reference

Suppose, however, that you want to highlight the parentheses separately. You can do this with a lot of convoluted region statements, or you can use the **matchgroup** option. This option tells *Vim* to highlight the start and end of a region with a different highlight group (in this case, the **xparen** group).

```
:syntax region xInside matchgroup=xparen start=/(/ end=)/
```

Match Offsets

Several options enable you to adjust the start and end of a pattern used in matching for the **:syntax match** and **:syntax region** directives. For example, the offset **ms=s+2** indicates that the match starts two characters from the start of the match.

For example:

```
:syntax match xHex /0x[a-fA-F0-9]*/ms=s+2
```

The general form of a match offset is as follows:

```
location=offset
```

Location is one of the following:

ms	Start of the element
me	End of the element
hs	Start highlighting here
he	End highlighting here
rs	Marks the start of a region
re	Marks the end of region
lc	Leading context

The offset can be as follows:

- s** Start of match
- e** End of the match

Clusters

One of the things you will notice as you start to write a syntax file is that you wind up generating a lot of syntax groups. The *Vim* editor enables you to define a collection of syntax groups called a cluster. Suppose you have a language that contains **for** loops, **if** statements, **while** loops, and functions. Each of them contains the same syntax elements: numbers and identifiers. You define them like this:

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent  
:syntax match xIf /^if.*/ contains=xNumber,xIdent  
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

You have to repeat the same **contains=** every time. If you want to add another contained item, you have to add it three times. Syntax clusters simplify these definitions by enabling you to group elements. To define a cluster for the two items that the three groups contain, use the following command:

```
:syntax cluster xState contains=xNumber,xIdent
```

Clusters are used inside other **:syntax** elements just like any syntax group. Their names start with @. Thus, you can define the three groups like this:

```
:syntax match xFor /^for.*/ contains=@xState  
:syntax match xIf /^if.*/ contains=@xState  
:syntax match xWhile /^while.*/ contains=@xState
```

You can add new elements to this cluster with the **add** argument:

```
:syntax cluster xState add=xString
```

You can remove syntax groups from this list as well:

```
:syntax cluster xState remove=xNumber
```

Including Other Syntax Files

The C++ language syntax is a superset of the C language. Because you do not want to write two syntax files, you can have the C++ syntax file read in the one for C by using the following command:

```
:source <sfile>:p:h/c.vim
```

(**:so** is short for **:source**.)

The word **<sfile>** is the name of the syntax file that is currently being processed. The **:p:h** modifiers remove the name of the file from the **<sfile>** word. Therefore, **<sfile>:p:h/c.vim** is used to specify the file **c.vim** in the same directory as the current syntax file.

The Vim Tutorial and Reference

Now consider the Perl language. The Perl language consists of two distinct parts: a documentation section in POD format, and a program written in Perl itself.

The POD section starts with =head and the end starts with =cut. You want to construct the Perl syntax file to reflect this. The :syntax include reads in a syntax file and stores the elements it defined in a syntax cluster. For Perl, the statements are as follows:

```
:syntax include @POD <sfile>:p:h/pod.vim  
:syntax region perlPOD start="^=head" end="^=cut"  
    \ contains=@POD
```

In this example, the top-level language is Perl. All the syntax elements of the POD language are contained in the Perl syntax cluster @POD.

Listing Syntax Groups

The following command lists all the syntax items:

```
:syntax
```

To list a specific group, use this command:

```
:syntax list group-name
```

This also works for clusters as well:

```
:syntax list @cluster-name
```

Synchronization

Compilers have it easy. They start at the beginning of a file and parse it straight through. The *Vim* editor does not have it so easy. It must start at the middle, where the editing is being done. So how does it tell where it is?

The secret is the :syntax sync command. This tells *Vim* how to figure out where it is. For example, the following command tells *Vim* to scan backward for the beginning or end of a C-style comment and begin syntax coloring from there:

```
:syntax sync ccomment
```

You can tune this processing with some options. The **minlines** option tells *Vim* the minimum number of lines to look backward, and **maxlines** tells the editor the maximum number of lines to scan.

For example, the following command tells *Vim* to look at 10 lines before the top of the screen:

The Vim Tutorial and Reference

```
:syntax sync ccomment minlines=10 maxlines=500
```

If it cannot figure out where it is in that space, it starts looking farther and farther back until it figures out what to do. But it looks no farther back than 500 lines. (A large **maxlines** slows down processing. A small one might cause synchronization to fail.) By default, the comment to be found will be colored as part of the **Comment** syntax group. If you want to color things another way, you can specify a different syntax group:

```
:syntax sync ccomment xAltComment
```

If your programming language does not have C-style comments in it, you can try another method of synchronization. The simplest way is to tell *Vim* to space back a number of lines and try to figure out things from there. The following command tells *Vim* to go back 150 lines and start parsing from there:

```
:syntax sync minlines=150
```

A large **minlines** option can make *Vim* slower. Finally, you can specify a syntax group to look for by using this command:

```
:syntax sync match sync-group-name  
  \ grouphere group-name pattern
```

This tells *Vim* that when it sees pattern the syntax group named *group-name* begins just after the pattern given. The *sync-group-name* is used to give a name to this synchronization specification.

For example, the *sh* scripting language begins an if statement with **if** and ends it with **fi**:

```
if [ -f file.txt ] ; then  
  echo "File exists"  
fi
```

To define a **grouphere** directive for this syntax, you use the following command:

```
:syntax sync match shIfSync grouphere shIf "\<if\>"
```

The **grouphere** option tells *Vim* that the pattern ends a group. For example, the end of the if/ fi group is as follows:

```
:syntax sync match shIfSync grouphere NONE "\< fi\> "
```

In this example, the **NONE** tells *Vim* that you are not in any special syntax region. In particular, you are not inside an **if** block.

The Vim Tutorial and Reference

You also can define matches and regions that are with no **grouphere** or '**groupthere**' options. These groups are for syntax groups skipped during synchronization. For example, the following skips over anything inside {}, even if would normally match another synchronization method:

```
:syntax sync match xSpecial /{.*}/
```

To clear out the synchronization commands, use the following command:

```
:syntax sync clear
```

To remove just the named groups, use this command:

```
:syntax sync clear sync-group-name sync-group-name ...
```

The **:syntax sync** commands only control the number of *lines* used to do syntax matching. The '**synmaxcol**' ('**smc**') option tells *Vim* how many columns to use for syntax matching. The default is 3000, and if you have a file with more than 3000 characters per line, syntax matching is the least of your problems.

Adding Your Syntax File to the System

Suppose that you have defined your own language and want to add it to the system. If you want it to be a part of *Vim*, you need to perform the following steps:

1. Create your syntax file and put it in the **\$VIMRUNTIME/syntax/language.vim** file.
2. Edit the file **\$VIMRUNTIME/syntax/synload.vim** and add a line to this file for your language. The line should look like this:

```
SynAu language
```

3. Edit the file **\$VIMRUNTIME/filetype.vim** and insert an **:autocmd** that recognizes your files and sets the filetype to your language. For example, for the foo language, execute the following command:

```
:autocmd BufRead,BufNewFile *.foo set ft=foo
```

Now *Vim* should automatically recognize your new language. If the file works well, you might want to send your changes to the *Vim* people so that they can put it in the next version of *Vim*.

Option Summary

contained

Syntax group is contained in another group. Items that are contained cannot appear

	at the top level, but must be contained in another group.
contains=group-list	Define a list of syntax groups that can be included inside this one. The group name can be ALL for all groups or ALLBUT,group-name for all groups except the names specified.
nextgroup=group	Define a group that may follow this one.
skipwhite	Skip whitespace between this group and the next one specified by nextgroup .
skipnl	Skip over the end of a line between this group and the next one specified by nextgroup
skipempty	Skip over empty lines between this group and the next one specified by nextgroup .
transparent	This group takes on the attributes of the one in which it is contained.
oneline	Do not extend the match over more than one line.
keepend	Do not let a pattern with an end-of-line (\$) match in it extend past the end of a line. This avoids the inner pattern inside a nested pattern eating an end of line. If this is a contained match, the match will not match the end-of-line character.
excludenl	Do not let a contained pattern extend the item in which it is contained past the end of a like

Appendix A: Installing Vim

You can obtain *Vim* from the web site at www.vim.org. This site contains the source to *Vim* as well as precompiled binaries for many different systems.

UNIX

You can get precompiled binaries for many different UNIX systems from www.vim.org. Go to <http://www.vim.org>, click the "Download Vim" link, and then follow the "Binaries Page" link. This takes you to the "Binaries" page, which lists the various precompiled binaries for many different systems along with the links to download them.

Volunteers maintain the binaries, so they are frequently out of date. It is a good idea to compile your own UNIX version from the source. Also, creating the editor from the source allows you to control which features are compiled. To compile and install the program, you'll need the following:

- A C compiler (GCC preferred)
- The *bzip* program (you can get it from <http://www.bzip.org>)

To obtain *Vim*, go to <http://www.vim.org> and click the "Download Vim" link. This page displays a list of sites that contain the software. Click a link to one that's near you. This takes you to a directory listing. Go into the "UNIX" directory and you'll find the sources for *Vim*. You'll need to download one files:

21

- vim-7.2.tar.bz2

Unpacking the sources

Unpacking the sources is a simple matter of using *bzcat* (part of the *bzip* software suite) to decompress the archive, and *tar* to extract the members:

```
$ bzcat vim-7.2.tar.bz2 | tar xf -
```

Next we go into the directory containing the sources:

```
$ cd vim
```

Running configure

The *configure* script configures the source so that it can be properly built. First let's take a look at the **--help** option (that is **<dash><dash>help**) to see what the command does:

The Vim Tutorial and Reference

```
$ ./configure --Khlp
`configure' configures this package to adapt to many kinds of systems.

Usage: auto/configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help                  display this help and exit
      --help=short             display options specific to this package
      --help=recursive          display the short help of all the included packages
  -V, --version                display version information and exit
  -q, --quiet, --silent         do not print `checking...' messages
      --cache-file=FILE         cache test results in FILE [disabled]
  -C, --config-cache           alias for `--cache-file=config.cache'
  -n, --no-create              do not create output files
      --srcdir=DIR              find the sources in DIR [configure dir or `..']

Installation directories:
  --prefix=PREFIX               install architecture-independent files in PREFIX
                                [/usr/local]
  --exec-prefix=EPREFIX          install architecture-dependent files in EPREFIX
                                [PREFIX]

By default, `make install' will install all the files in
`/usr/local/bin', `/usr/local/lib' etc. You can specify
an installation prefix other than `/usr/local' using `--prefix',
for instance `--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR                  user executables [EPREFIX/bin]

... three pages of options deleted ...

  --with-tlib=library            terminal library to be used

Some influential environment variables:
  CC                          C compiler command
  CFLAGS                      C compiler flags
  LDFLAGS                     linker flags, e.g. -L<lib dir> if you have libraries in a
                                nonstandard directory <lib dir>
  LIBS                        libraries to pass to the linker, e.g. -l<library>
  CPPFLAGS                    C/C++/Objective C preprocessor flags, e.g. -I<include dir> if
                                you have headers in a nonstandard directory <include dir>
  CPP                         C preprocessor
  XMKMF                       Path to xmkmf, Makefile generator for X Window System
```

The Vim Tutorial and Reference

```
Use these variables to override the choices made by `configure' or to help  
it to find libraries and programs with nonstandard names/locations.
```

Don't worry about all these options. You'll probably never use most of them. They only really important option is --prefix (<dash><dash>prefix) which tells *configure* where you want to install the program. For example, in a users home directory.

```
$ ./configure --prefix=/home/auser/local
```

Now *configure* looks through system and figures out how to make *Vim*.

```
configure: creating cache auto/config.cache  
checking whether make sets $(MAKE)... yes  
checking for gcc... gcc  
  
... many many checks omitted ...  
  
checking for setjmp.h... yes  
checking for GCC 3 or later... yes  
configure: updating cache auto/config.cache  
configure: creating auto/config.status  
config.status: creating auto/config.mk  
config.status: creating auto/config.h
```

Most of this output you can ignore. However one common problem (discussed below) occurs when you attempt to make the GUI version of *Vim* and don't have the development files for the approbate toolkit installed. How to deal with this is discussed in the troubleshooting section below.

Next it's time to actually build the program.

```
$ make 2>&1 | tee log
```

This version of the command tells the shell to redirect standard error to standard out (**2>&1**) and to send the whole thing to the tee command (**| tee**). The *tee* command prints the results on the screen and records the data in the file *log*.

```
$ make 2>&1 | tee log  
Starting make in the src directory.  
If there are problems, cd to the src directory and run make there  
cd src && make first  
make[1]: Entering directory `/mnt/disk/vim/build/vim72/src'  
mkdir objects
```

The Vim Tutorial and Reference

```
CC="gcc -Iproto -DHAVE_CONFIG_H -DFEAT_GUI_GTK -I/usr/include/gtk-2.0
-I/usr/lib gtk-2.0/include -I/usr/include/atk-1.0 -I/usr/include/cairo
-I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
-I/usr/include/freetype2 -I/usr/include/libpng12           " srcdir=.. sh
./osdef.sh

... a few hundred lines of log file later ...

make[2]: Leaving directory `/mnt/disk/vim/build/vim72/src/xxd'
make[1]: Leaving directory `/mnt/disk/vim/build/vim72/src'
$
```

Assuming that you compiled without error, the next step is to install the software with the command:

```
$ make install 2>&1 | tee log.install
```

The results can be seen below.

```
$ make install 2>&1 tee log.install
Starting make in the src directory.
If there are problems, cd to the src directory and run make there
cd src && make install
make[1]: Entering directory `/mnt/disk/vim/build/vim72/src'

... Many lines omitted

if test -d /home/sdo/local/share/icons/locolor/16x16/apps -a -w
/home/sdo/local/share/icons/locolor/16x16/apps \
    -a ! -f
/home/sdo/local/share/icons/locolor/16x16/apps/gvim.png; then \
    cp ../runtime/vim16x16.png
/home/sdo/local/share/icons/locolor/16x16/apps/gvim.png; \
    fi
cp gvimtutor /home/sdo/local/bin/gvimtutor
chmod 755 /home/sdo/local/bin/gvimtutor
make[1]: Leaving directory `/mnt/disk/vim/build/vim72/src'
```

If you are lucky these commands will run without error and you'll have a good *Vim* installation. If not, read the next section;

Dealing with common installation problems

The console (text) version of Vim is fairly robust and has compiled successfully on every machine I've tried it on, and I've compiled on a lot of very strange machines. (Microsoft Windows excluded.)

The Vim Tutorial and Reference

The big problem comes in configuring and compiling the GUI version of Vim. No matter which version you choose, the system will make heavy use of several X Windows GUI libraries. If these libraries are not installed on your system, or you do not have the correct version of the library, Vim will fail to build the GUI version of itself.

To see if you have a GUI problem, after executing the make command (make only, not make install), go into the src directory and execute the command ./vim -g:

```
$ make  
$ cd src  
$ ./vim -g
```

If a GUI appears and it's the correct GUI, then you don't have a problem. You're done.

But if you get an error, then look at your configuration log file: *config.log*. The relevant lines (in this case) are:

```
configure:7800: result: yes  
configure:7841: checking if X11 header files can be found  
configure:7857: gcc -c -g -O2 -I/usr/local/include conftest.c >&5  
conftest.c:17:27: fatal error: X11/Intrinsic.h: No such file or directory  
#include <X11/Intrinsic.h>  
^  
compilation terminated.  
configure:7857: $? = 1
```

This is where the configuration command is attempting to figure out what libraries are installed so it can figure out what GUI to build. In this example we do not have the X11 development library which supplies *X11/Intrinsic.h*.

Now you may think that you have the yy library installed but *Vim* fails to detect it. You need to find out why. Some clues can be found the *config.log* file which is automatically generated by the configure command. This file gives you some idea of what tests are being run to determine which libraries are installed.

If the *config.log* file does not help you, you'll need to examine the contents of the configuration command itself. It is written the shell programming language so it is somewhat readable, although if you are not a expert shelling program going into the script, you will be before you leave.

If all else fails you can hit the mail list and ask for help. But before doing so, please, please search the archive. Someone may have already experienced your problem and found a solution.

The Vim Tutorial and Reference

In closing I must say that the last time I got *Vim* installed it went surprisingly well even though it was on a rather difficult platform. I simply sent a request to the IT department, and a week later I got an E-Mail saying: "We finally got *Vim* installed and you wouldn't believe the trouble we had getting the thing to work." (Yes I would, I wrote the book, literally, on *Vim*.)

Installation for Each UNIX User

Each UNIX user should make sure that *Vim* is in his path. If you have an *.exrc* file, copy it to *.vimrc*:

```
$ cp ~/.exrc ~/.vimrc
```

If you do not have an *.exrc* file, create an empty *.vimrc* file by executing the following command:

```
$ touch ~/.vimrc
```

Note: The presence of the *.vimrc* file turns on all the fancy features of *Vim*. If this file is not present, *Vim* tries very hard to look like *Vi*, even disabling some of its features to do so.

Installing on Microsoft Windows

Installation on Microsoft Windows is now a very simple operation. First of all download the Microsoft Windows binary from <http://www.vim.org>. Next simply run the installer and answer the questions.

Common Installation Problems and Questions

This section describes some of the common problems that occur when installing *Vim* and suggests some solutions. It also contains answers to many installation questions.

I Do Not Have Root Privileges. How Do I Install Vim? (UNIX)

Use the following configuration command to install *Vim* in a directory called *\$HOME/vim* :

```
$ configure --prefix=$HOME/vim
```

This gives you a personal copy of *Vim*. You need to put *\$HOME/vim/bin* in your path to access the editor.

The Colors Are Not Right on My Screen. (UNIX)

Check your terminal settings by using the following command:

```
$ echo $TERM
```

If the terminal type listed is not correct, fix it. UNIX has a database called *termcap*, which describes the capabilities of your terminal. Almost all *xterm* programs support color. Frequently, however, the *termcap* entry for *xterm* defines a terminal without color. To get color to work, you might have to tell the system that you have an *xtermc* or *cxterm* terminal. (Another solution is to always use the GUI version of *Vim* called *gvim*.)

I Am Using RedHat Linux. Can I Use the Vim That Comes with the System?

By default RedHat installs a minimal version of *Vim*. Check your RPM packages for something named *vim-enhanced-version.rpm* and install that.

How Do I Turn Syntax Coloring On? (All)

Use the following command:

```
:syntax on
```

What Is a Good vimrc File to Use? (All)

See the www.vim.org Web site for several good examples.

Appendix B: The <> Key Names

Appendix A: This appendix provide a quick reference for the <> key names in *Vim*.

The Function Keys

<F1>	<F2>	<F3>	<F4>	<F5>	<F6>
<F7>	<F8>	<F9>	<F10>	<F11>	<F12>
<F13>	<F14>	<F15>	<F16>	<F17>	<F18>
<F19>	<F20>	<F21>	<F22>	<F23>	<F24>
<F25>	<F26>	<F27>	<F28>	<F29>	<F30>
<F31>	<F32>	<F33>	<F34>	<F35>	

Line Endings

<CR>	<Return>	<Enter>
<LF>	<LineFeed>	
<NL>	<NewLine>	

Other Special Characters

<BS >	<BackSpace>
<Ins>	<Insert>
	<Delete>

Editing Keys

<End>	<Home>	<PageDown>	<PageUp>
-------	--------	------------	----------

Arrow Keys

<Left>	<Right>	<Up>	<Down>
--------	---------	------	--------

Keypad Keys

<kDivide>	<kPlus>	<kEnd>	<kEnter>
<kHome>	<kPageUp>	<kMinus>	<kMultiply>
<kPageDown>			

VT100 Special Keys

The VT100 terminal has an extra set of function keys:

<xF1>	<xF2>	<xF3>	<xF4>	<xEnd>	<xHome>
-------	-------	-------	-------	--------	---------

Printable Characters

<Bar>	
<Bslash>	\
<Space>	
<Tab>	
<Lt>	<

Other Keys

<Esc>	<Help>	<Null>	<Undo>
-------	--------	--------	--------

Termcap Entries

On UNIX systems, the Termcap or Terminfo database contains a description of the terminal, including function keys. The special key <t_xx> represents the key defined by **xx** Termcap entry.

See your UNIX documentation for a complete list of keys. One way to get a list (for most systems) is to execute the following command:

```
$ man terminfo
```

Mouse Actions

<LeftMouse>	<RightMouse>
<LeftRelease>	<RightRelease>
<LeftDrag>	<RightDrag>
<MiddleDrag>	<MiddleRelease>
<MouseUp>	<MouseDown>
<MiddleMouse>	<Mouse>

Modifiers

M	Meta (Alt)
C	Control
S	Shift
D	Macintosh command key

Mouse Modifiers

<Blank>	Mouse button one
2	Mouse button two
3	Mouse button three
4	Mouse button four

The Vim Tutorial and Reference

Note: If you want to find the name of a key on the keyboard, you can go into Insert mode and press **CTRL-K** key. The <> name of the key will be inserted. This works for the function keys and many other keys.

Appendix C: Normal-Mode Commands

[count]<BS>	Move count characters to the left. See the ' backspace ' option to change this to delete rather than backspace. (Same as: <Left>, CTRL-H , CTRL-K , h . See page 31, 32, 38, 49, 268.)
[count]<C-End>	Move to the end of line count. If no count is specified, go to the end of the file. (Same as: G .) (See pages 557.)
[count]<C-Home>	Move to the start (first non-blank character) of line [count] . Default is the beginning of the file. (Same as: gg .) (See page 226.)
[count]<C-Left>	Move count WORDS backward. (Same as: B .) (See page 263.)
<C-LeftMouse>	Jump to the location of the tag whose name is under the cursor. (Same as: CTRL-] , g<LeftMouse> .) (See pages 131, 132, 134, 240.)
[count] <C-PageDown>	Go to the next tab. If a [count] is specified, go to the given tab. (Same as : tabn : tabnext , gt .) (See page 97.)
[count] <C-PageUp>	Go to the previous tab. If a [count] is specified, go to the given tab. (Same as : tabN : tabNext , : tabp , : tabprevious , gT .) (See page 97.)
<C-Right>	Move count WORDS forward. (Same as: w .) (See page 263.)
[count]<C-RightMouse>	Jump to a previous entry in the tag stack. (Same as: CTRL-T , g<RightMouse> .) (See pages 133, 134.)
[count]<CR>	Move down count lines. Cursor is positioned on the first nonblank character on the line. (Same as: < ENTER >, CTRL-M , and +.) (See page 265.)
["{register}"] [count]	Delete characters. If a " {register} " is present, the deleted text is stored in it. (Same as: x .) (See pages 33, 36, 69, 232, 276.)
[count]<Down>	Move [count] lines down. If the ' keymodel ' is set to startselect , start a selection. (Same as: < NL >, CTRL-J , CTRL-N , j .) (See pages 330, 467.)
[count]<End>	Move the cursor to the end of the line. If a [count] is present, move to the end of the count line down from the current one. (Same as: < kEnd >, \$.) (See pages 43, 329.)
[count]<Enter>	Move down [count] lines. (Default = 1.) Cursor is

The Vim Tutorial and Reference

	positioned on the first nonblank character on the line. (Same as: <CR>, CTRL-M , +.) (See page 265.)
< F1 >	Go to the initial help screen. (Same as: < Help >, : h , and : help .) (See pages 40, 130, 131, 227.)
< F8 >	Toggle between left-to-right and right-to-left modes. (See page 252.)
< F9 >	Toggles the encoding between ISIR-3342 standard and <i>Vim</i> extended ISIR-3342 (supported only in right-to-left mode when ' fkmap ' [Farsi] is enabled). (See page 255.)
< Help >	Go to the initial help screen. (Same as: < F1 >, : h , : help .) (See pages 40, 130, 131, 227.)
< Home >	Move to the first character of the line. (Same as: < kHome >.) (See page 44, 329.)
[count]< Insert > text < Esc >	Insert text. If [count] is present, the text will be inserted count times. (Same as: i .) (See pages 30, 32, 56.)
[count]< kEnd >	Move the cursor to the end of the line. If a count is present, move to the end of the count line down from the current one. (Same as: < End >, \$.) (See page 43, 329.)
< kHome >	Move to the first character of the line. (Same as: < Home >. See page 44, 329.)
[count]< Left >	Move left count characters. If the ' keymodel ' is set to startselect , start a selection. (Same as: < BS >, CTRL-H , CTRL-K , h .) (See page 268, 467.)
< LeftMouse >	Move the text cursor to the location of the mouse cursor. (See pages 465.)
["register"] < MiddleMouse >	Insert the text in register at the location of the mouse cursor. (Same as: P .) (See pages 235, 236, 236, 465.)
< MouseDown >	Scroll three lines down. (See page 465.)
< MouseUp >	Scroll three lines up. (See page 465.)
[count]< NL >	Move count lines down. (Same as CTRL-N .) (See page 330.)
[count]< PageDown >	Scroll count pages forward. If the ' keymodel ' is set to startselect , stop a selection. (Same as: < S-Down >, CTRL-F .) (See page 271, 467.)
[count]< PageUp >	Scroll the window count pages backward. If the ' keymodel ' is set to startselect , stop a selection. (Same as: < S-Up >, CTRL-B .) (See page 270, 467.)
[count]< Right >	Move right count characters. If the ' keymodel ' is set to

The Vim Tutorial and Reference

	startselect , stop a selection. (Same as: <Space>, 1.) (See page 268, 467.)
<RightMouse>	Start select mode with the text cursor to the mouse cursor highlighted. (See pages 465.)
[count]<S-Down>	Scroll [count] pages forward. If you are running Windows GUI version, <S-Down> enters visual mode and selects down. (Same as: <PageDown>, CTRL-F .) (See page 271.)
[count]<S-Left>	Move left count words. (Same as: b .) (See page 42.)
[count]<S-LeftMouse>	Find the next occurrence of the word under the cursor. (See page 294.)
<S-MouseDown>	Scroll a full page up. (See page 465.)
<S-MouseUp>	Scroll a full page down three lines up. (See page 465.)
[count]<S-Right>	Move count words forward. (Same as: w .) (See pages 42, 49, 49, 51.)
[count]<S-RightMouse>	Search backward for the word under the cursor. (See page 295.)
[count]<S-Up>	Scroll count pages up. (Same as: <PageUp>, CTRL-B .) (See page 270.)
[count]<Space>	Move count spaces to the right. (Same as: <Right>, l . See page 268.)
[count]<Tab>	Go to the count position in the jump list. (Same as: CTRL-I . See page 266.)
[count]<Undo>	Undo the last count changes. (Same as: u .) (See page 34.)
[count]<Up>	Move count lines up. If the ' keymodel ' is set to startselect , start a selection. (Same as: CTRL-P , k . See page 122, 467.)
CTRL-\ CTRL-N	Enter normal mode from any other mode. (See page 102.)
CTRL-]	Jump to the function whose name is under the cursor. (In the help system, jump to the subject indicated by a hyperlink.) (Same as: <C-LeftMouse>, g<LeftMouse> .) (See pages 38, 131, 132, 134, 240.)
[count]CTRL-^	If a [count] is specified, edit the [count] file on the command line. If no [count] is present, edit the previously edited file. Thus repeated CTRL-^ can be used to toggle rapidly between two files. (See page 81.)
CTRL-_	Switch between English and a foreign language keyboard. (See pages 255, 256, 340.)
[count]CTRL-A	Add [count] to the number under the cursor. If no

The Vim Tutorial and Reference

	<p>[<i>count</i>] is specified, increment the number. (See pages 277, 552.)</p>
[<i>count</i>] CTRL-B	<p>Move back [<i>count</i>] screens. (Default = 1.) (Same as: <PageUp>, <S-Up>. See pages 270.)</p>
CTRL-BREAK	<p>Interrupt search (Same as CTRL-C). (See page 294.)</p>
CTRL-C	<p>Interrupt search. (Same as CTRL-BREAK). (See page 294.)</p>
[<i>count</i>] CTRL-D	<p>Move down the number of lines specified by the 'scroll' option. If a [<i>count</i>] is specified, set the 'scroll' option to [<i>count</i>] and then move down. (See pages 48, 271.)</p>
[<i>count</i>] CTRL-E	<p>Move down [<i>count</i>] lines. (See page 271, 272.)</p>
[<i>count</i>] CTRL-F	<p>Scroll the window [<i>count</i>] pages forward. (Same as: <PageDown>, <S-Down>.) (See page 271.)</p>
CTRL-G	<p>Display the current file and location of the cursor within that file. (Same as: :file.) (See pages 47, 268.)</p>
1 CTRL-G	<p>Same as CTRL-G, but include the full path in the filename. (See pages 268.)</p>
2 CTRL-G	<p>Same as 1 CTRL-G, but adds a buffer number. (See pages 268.)</p>
[<i>count</i>] CTRL-H	<p>Move [<i>count</i>] characters to the left. See the 'backspace' option to change this to delete rather than backspace. (Same as: <BS>, <Left>, CTRL-K, h.) (See page 31, 32, 38, 49, 268.)</p>
[<i>count</i>] CTRL-I	<p>Jump to the [<i>count</i>] next item in the jump list. (Same as: <Tab>. See page 266.)</p>
[<i>count</i>] CTRL-J	<p>Move down [<i>count</i>] lines. (Same as: <Down>, <NL>, CTRL-J, j.) (See pages 330.)</p>
[<i>count</i>] CTRL-K	<p>Move [<i>count</i>] characters to the left. (Same as: <BS>, <Left>, CTRL-H, h.) (See page 31, 32, 38, 49, 268.)</p>
CTRL-L	<p>Redraw screen. (See page 227.)</p>
CTRL-L	<p>Leave insert mode if 'insertmode' is set. (See page 258.)</p>
CTRL-M	<p>Move down count lines. Cursor is positioned on the first nonblank character on the line. (Same as: <ENTER>, <CR>, and +.) (See page 265.)</p>
[<i>count</i>] CTRL-N	<p>Move count lines down. (Same as: <Down>, <NL>, CTRL-J, j.) (See pages 330.)</p>
[<i>count</i>] CTRL-O	<p>Jump to the count previous item in the jump list. (See page 266.)</p>
[<i>count</i>] CTRL-P	<p>Move count lines upward. (Same as: <Up>, k.) (See pages 31, 32, 38, 41, 122.)</p>

The Vim Tutorial and Reference

CTRL-Q	Not a <i>Vim</i> command. Used by some terminals to start output after it was stopped by CTRL-S . (See page 227.)
CTRL-R	Redo the last change that was undone. (See page 34, 286.)
CTRL-S	Not a <i>Vim</i> command. Used by some terminals to stop output. (See page 227.)
[count] CTRL-T	Go back [count] tags. If the current buffer has been modified, this command fails unless the force (!) option is present. When using the help system, this command returns to the location you were at before making the last hyperlink jump. (Same as: <C-RightMouse> , g<RightMouse> .) (See pages 39, 133, 134.)
[count] CTRL-U	Move up the number of lines specified by the ' scroll ' option. If a [count] is specified, set the ' scroll ' option to [count] and then scroll up. (See pages 48, 269.)
CTRL-V	Start visual block mode. (See pages 101, 103 and 105.)
[count] CTRL-W<Down>	Move down a window. If a [count] is specified, move to window number [count] . (Same as: CTRL-W CTRL-J , CTRL-Wj . See page 84, 335, 464.)
[count] CTRL-W<Left>	Go left [count] windows. (Same as: CTRL-W CTRL-h , CTRL-W<Left> . See page 85.)
[count] CTRL-W<Right>	Go right [count] windows. (Same as: CTRL-W CTRL-l , CTRL-W<Right> . See page 85.)
[count] CTRL-W<Up>	Move up a window. If a [count] is specified, move to window number [count] . (Same as: CTRL-W CTRL-k , CTRL-Wk . See page 84, 335, 464.)
[count] CTRL-W CTRL-]	Split the current window and jump to the function whose name is under the cursor. If a [count] is specified, it is the height of the new window. (Same as: CTRL-W] .) (See page 134.)
CTRL-W CTRL-^	Split the window and edit the alternate file. If a [count] is specified, split the window and edit the [count] file on the command line. (Same as: CTRL-W^ .) (See page 340.)
[count] CTRL-W CTRL_-	Set the height of the current window to [count] . (Same as: CTRL-W_, :resize .) (See page 89.)
CTRL-W CTRL-B	Move to the bottom window. (Same as: CTRL-Wb .) (See page 335.)

The Vim Tutorial and Reference

CTRL-W CTRL-C	Cancel any pending window command. (See page 84.)
CTRL-W CTRL-D	Split the window and find the definition of the word under the cursor. If the definition cannot be found, do not split the window. (Same as: CTRL-Wd.) (See page 341.)
CTRL-W CTRL-F	Split the window and edit the file whose name is under the cursor. Looks for the file in the current directory, and then all the directories specified by the ' path ' option. (Same as: CTRL-Wf.) (See page 341.)
[count] CTRL-W CTRL-G CTRL-]	: split followed a CTRL-] . If a [count] is specified, make the new window [count] lines high. (Same as: CTRL-Wg CTRL-] , CTRL-Wg.) (See page 341.)
[count] CTRL-W CTRL-G }	Do a : ptjump on the word under the cursor. If a [count] is specified, make the new window [count] lines high. (Same as: CTRL-W CTRL-G{} .) (See page 393.)
[count] CTRL-W CTRL-H	Go left [count] windows. (Same as: CTRL-W h , CTRL-W<Left> .) (See page 85.)
[count] CTRL-W CTRL-I	Split the window and search for the [count] occurrence of the word under the cursor. Start the search at the beginning of the file. (Same as: CTRL-Wi . See page 340.)
[count] CTRL-W CTRL-J	Move down a window. If a count is specified, move to window number count. (Same as: CTRL-W<Down> , CTRL-Wj.) (See page 84, 335, 464.)
[count] CTRL-W CTRL-K	Move count windows up. (Same as: CTRL-W<Up> , CTRL-Wk.) (See page 84, 335, 464.)
[count] CTRL-W CTRL-L	Go right [count] windows. (Same as: CTRL-W l , CTRL-W<Right> . See page 85.)
CTRL-W CTRL-N	Split the window like : split . The window is started on a blank file. (Same as: CTRL-Wn . See page 86.)
CTRL-W CTRL-O	Make the current window the only one. (Same as: CTRL-Wo , :on , :only .) (See page 339.)
CTRL-W CTRL-P	Move to the previous window. (Same as: CTRL-Wp. (See pages 235, 236, 335.)
CTRL-W CTRL-Q	Close a window. If this is the last window, exit <i>Vim</i> . The command fails if this is the last window for a modified

The Vim Tutorial and Reference

	file, unless the force (!) option is present. (Same as: CTRL-W q , :q , :quit .) (See pages 35, 84, 165, 288, 338.)
[count] CTRL-W CTRL-R	Rotate windows downward. (Same as: CTRL-Wr .) (See page 336.)
[count] CTRL-W CTRL-S	Split the current window. (Make the new window count lines high.) (Same as: CTRL-Ws , CTRL-WS , :sp , :split . See pages 235, 236.)
CTRL-W CTRL-T	Move the top window. (Same as: CTRL-Wt . See page 335.)
CTRL-W CTRL-W	Move to the next window down. (if in vertical mode move to the next window to the right.) If there is no next window, move to the first one. If a [count] is specified, move to window number [count] . (Same as: CTRL-Ww .) (See pages 84, 85, 336.)
[count] CTRL-W CTRL-X	Exchange the current window with the next one. If there is no next one, exchange the last window with the first. If a [count] is specified, exchange the current window with window number [count] . (Same as: CTRL-Wx .) (See page 338.)
CTRL-W CTRL-Z	Close the preview window. Discard any changes if the force (!) option is present. (Same as: CTRL-Wz , :pc , :pclose .) (See page 393.)
[count] CTRL-W +	Increase the size of the current window by [count] . (Default = 1.) (Same as: :res + , :resize + .) (See page 89.)
[count] CTRL-W - (CTRL-W <dash>)	Decrease the size of the current window by [count] . (Default = 1.) (Same as :res - , :resize -) (See page 89.)
CTRL-W=	Make all windows the same size (or as close as possible). (See page 89.)
[count] CTRL-W]	Split the current window and jump to the function whose name is under the cursor. If a [count] is specified, it is the height of the new window. (Same as: CTRL-W CTRL-] .) (See page 134.)
[count] CTRL-W ^	Split the window and edit the alternate file. If a [count] is specified, split the window and edit the [count] file on the command line. (Same as: CTRL-W CTRL-^ .) (See page 340.)

The Vim Tutorial and Reference

[count]CTRL-W _	Set the current window to be [count] lines high. If no count is specified, make the window as big as possible. (Same as: CTRL-W CTRL-_ , :res , :resize .) (See page 89.)
CTRL-W }	Do a :ptag on the word under the cursor. (See page 393.)
CTRL-W b	Move to the bottom window. (Same as: CTRL-W CTRL-B .) (See page 335.)
CTRL-W c	Close the current window. (Same as: :clo , :close .) (See page 84.)
CTRL-W d	Split the window and find the definition of the word under the cursor. If the definition cannot be found, do not split the window. (Same as: CTRL-W CTRL-D .) (See page 341.)
CTRL-W f	Split the window and edit the file whose name is under the cursor. Looks for the file in the current directory, then all the directories specified by the ' path ' option. (Same as: CTRL-W CTRL-F .) (See page 341.)
CTRL-W g CTRL-]	:split followed a CTRL-] . (Same as: CTRL-W CTRL-G , CTRL-Wg] .) (See page 341.)
CTRL-W g]	:split followed a CTRL-] . (Same as: CTRL-W g CTRL-] , CTRL-W CTRL-G] .) (See page 341.)
CTRL-W g }	Do a :ptjump on the word under the cursor. (Same as: CTRL-W CTRL-G} . (See page 393.)
CTRL-Wgf	Open a new tab and do a :find on the file who's name is under the cursor. (Similar to :tabfind , :tabf) (See page 98.)
CTRL-WgF	Open a new tab and do a :find on the file who's name is under the cursor. Position the cursor on the line who's number appears after the file name. (Similar to :tabfind , :tabf) (See page 98.)
[count]CTRL-W h	Go left [count] windows. (Same as: CTRL-W CTRL-h , CTRL-W<Left> .) (See page 85.)
[count]CTRL-W i	Split the window and search for the [count] occurrence of the word under the cursor. Start the search at the beginning of the file. (Same as: CTRL-W CTRL-I .) (See page 340.)
[count]CTRL-W j	Move down a window. If a [count] is specified, move to window number [count] . (Same as: CTRL-W CTRL-J , CTRL-W<Down> .) (See page 84, 335, 464)
[count]CTRL-W k	Go up [count] windows. (Same as: CTRL-W CTRL-K , CTRL-W<Up> .) (See page 84, 335, 464.)

The Vim Tutorial and Reference

[count]CTRL-W 1	Go right [count] windows. (Same as: CTRL-W CTRL-L , CTRL-W<Right> .) (See page 85.)
CTRL-W n	Split the window like :split . The window is started on a blank file. (Same as: CTRL-W CTRL-N .) (See page 86.)
CTRL-W o	Make the current window the only one. If ! is specified, modified files whose windows are closed will have their contents discarded. (Same as: CTRL-W CTRL-O , :on , :only .) (See page 339.)
CTRL-W p	Move to the previous window. (Same as: CTRL-W CTRL-P .) (See pages 235, 236, 335.)
CTRL-W q	Close a window. If this is the last window, exit <i>Vim</i> . The command fails if this is the last window for a modified file, unless the force (!) option is present. (Same as: CTRL-W CTRL-Q , :q , :quit .) (See pages 35, 84, 165, 288, 338.)
[count]CTRL-W r	Rotate windows downward. (Same as: CTRL-W CTRL-R , CTRL-WR .) (See page 336.)
[count]CTRL-W R	Rotate windows upward. (See page 336.)
[count]CTRL-W s	Split the current window. Make the new window [count] lines high (same as [count]CTRL-WS). (Same as: CTRL-W CTRL-S , CTRL-WS , :sp , :split .) (See pages 235, 236.)
[count]CTRL-W S	Split the current window. Make the new window count lines high (same as [count]CTRL-Ws). (Same as: CTRL-W CTRL-S , CTRL-Ws , :sp , :split .) (See page 235, 236.)
CTRL-W t	Move the top window. (Same as: CTRL-W CTRL-T .) (See page 335.)
[count]CTRL-W w	Move to the next window down (to the right in vertical mode). If there is no next window, move to the first one. If a [count] is specified, move to window number [count] . (Same as: CTRL-W CTRL-W .) (See pages 84, 85, 336.)
[count]CTRL-W W	Move to the previous window. If at the top window, go to the bottom one. >If a [count] is specified, move to window number [count] . (See page 336.)
[count]CTRL-W x	Exchange the current window with window number count . (Same as: CTRL-W CTRL-X .) (See page 338.)
CTRL-W z	Close the preview window. (Same as: CTRL-W CTRL-Z , :pc , :pclose .) (See page 393.)
[count]CTRL-X	Subtract [count] from the number under the cursor. If no [count] is specified, decrement the number. (See

The Vim Tutorial and Reference

	pages 278, 552.)
[count]CTRL-Y	Move up [count] lines. (See pages 270, 272.)
CTRL-Z	Suspend the editor (Unix only). (Same as CTRL-Z , :suspend , :sus , :st , and :stop .) (See page 227.)
! {motion} {command}	Filter the block of text represented by {motion} through the an external {command} command. (See pages 76, 139, 184, 239.)
[count]!!{command}	Filter the current line (or [count] lines} through the an external command. (See page 76, 77.)
[count]#	Search for the word under the cursor, backward. (Same as: f , <S-RightMouse> .) (See page 295.)
[count]\$	Move the cursor to the end of the line. If a [count] is present, move to the end of the [count] line down from the current one. (Same as: <End> , <kEnd> .) (See pages 43, 50, 56, 328, .)
%	Find the matching (,) , { , } , #ifdef/#else/#endif . (See pages 113, 114, 120, 122, 129, 373.)
[count]%	Jump to the line whose [count] percent of the way through the file. (See page 46.)
&	Synonym for :s//~/ Repeat last substitution. (See page 435.)
' {letter} (single quote)	Go to the line containing mark named {letter} . (See pages 239, 319.)
[count](Move backward [count] sentences. (See page 189.)
[count])	Move forward [count] sentences. (See page 184, 189.)
[count]*	Search for the word under the cursor, forward. (See page 294.)
[count]+	Move down [count] lines. (Default = 1.) Cursor is positioned on the first nonblank character on the line. (Same as: <CR> , + , CTRL-M .) (See page 265.)
[count],	Reverse the direction of the last single character and perform the search [count] times. (See page 264.)
[count]-	Move up [count] lines. (Default = 1.) Cursor is positioned on the first non-blank character on the line. (See page 264.)
[count] . (dot)	Repeat the last simple normal mode command that changed text. If [count] is specified use it instead of the [count] specified with the original command. (See page 53, 233, 185.)
[count]/	Repeat last search in the forward direction. (See pages 60, 60.)
[count]/{pattern}	Search forward. (See pages 59-68, 233, 319.)
[count]/{pattern}/{offset}	

The Vim Tutorial and Reference

	Search forward, position the cursor at {offset} from the search pattern. (See page 295.)
[count]//{offset}	Repeat last search in the forward direction with a new offset. (See page 297.)
[count];	Repeat the last single character search [count] times. (Default = 1.) (See page 264.)
[count]<< <{motion}	Shift count lines to the left. (See pages 115.) Shift lines from cursor to {motion} to the left. (See pages 115.)
[count]>> >{motion}	Shift [count] lines to the right. (See page 115.) Shift lines from cursor to {motion} to the right. (See pages 115, 129, 129.)
={motion}	Filter {motion} lines through the internal indentation program or if the ' equalprg ' option is defined, the external program given by ' equalprg '. (See page 119.)
[count]?	Repeat last search in the backward direction. (See page 64, 65.)
[count]?{pattern}	Search backward. (See page 297.)
[count]?{pattern}?{offset}	Search backward, position the cursor at {offset} from the search pattern. (See page 297.)
[count]??{offset}	Repeat last search in the backward direction with a new {offset} . (See page 297.)
[count]@{character}	Execute the macro in register {character} . (See page 55.)
["{register}]" <MiddleMouse>	Put the {register} in the buffer like the p command, but adjust the text to fit the indent of the current line. (Same as: [p, [P,]P.]) (See page 367.)
[count] [CTRL-D]	Find definition of the macro currently sitting under the cursor. Start the search from the beginning of the file. (See page 127.)
[count] [CTRL-I]	Search for the word under the cursor starting at the beginning of the file. (See pages 126, 402.)
[count] ["{register}<MiddleMouse>	Put the {register} in the buffer like the p command, but adjust the text to fit the indent of the current line. (See page 367.)
[#	Finds the previous unmatched #if/#else/#endif . (See page 395.)
[count] [*	Move backward to the beginning of the [count]

The Vim Tutorial and Reference

		comment from the cursor. (Same as: [/.) (See page 396.)
[count]	/	Same as: [*.. (See page 396.)
[count]	(Move backward to the [count] previous unmatched (in column 1. (See page 395.)
[count])	Move backward to the [count] previous unmatched). (See page 395.)
[count]	{	Move backward [count] sections or to the previous { in column 1. (See pages 190, 396.)
[count]	}	Move [count] sections backwards or to the previous } in column 1. (See pages 396.)
[count]	[}	Finds the [count] previous unmatched }. (See page 395.)
[count]	[c	Go to the [count] next change in the current diff. (See page 384.)
[count]	[d	List the definition of the macro. Start search at the beginning of the file. (See pages 127, 402.)
[count]	[D	List all definitions of the macro whose name is under the cursor. Start the list with the next first definition in the file. (See page 127.)
[f		Deprecated. Use gf instead. (Same as: gf ,]f.) (See page 399.)
[count]	[i	Display the [count] line that contains the keyword under the cursor. The search starts from the beginning of the file. (See page 403.)
[I		List all lines in the current and included files that contain the word under the cursor. (See page 403.)
[m		Search backward for the start of a method. (See page 396.)
[M		Search backward for the end of a method. (See page 396.)
["{register}"]	[p	Put the {register} in the buffer like the P command, but adjust the text to fit the indent of the current line. (Same as: [<MiddleMouse>, [P,]P.) (See page 367.)
["{register}"]	[P	Put the {register} in the buffer like the P command, but adjust the text to fit the indent of the current line. (Same as: [<MiddleMouse>, [p, [P.) (See page 367..)
[count]	[s	Search for the next misspelled word starting at the current location. A [count] cause the command to repeat [count] times. (See page 185.)
[count]	[S	Search for the next misspelled word starting at the current location, but do not stop at bad words or uncommon words. A [count] cause the command to

The Vim Tutorial and Reference

		repeat [count] times. (See page 185.)
[z		Move to the start of the current open fold. (See page 390.)
[" {register}]]<MiddleMouse>	Put the {register} in the buffer like the p command, but adjust the text to fit the indent of the current line. (Same as:]p .) (See page 367.)
[count]]CTRL-D	Find definition of the macro currently sitting under the cursor. Start the search from the beginning current location. (See page 127, 439.)
[count]]CTRL-I	Search for the word under the cursor starting at the current cursor location. (See pages 126,-402.)
[count]]#	Finds the next unmatched #if/#else/#endif . (See page 395.)
[count]]()	Move forward to the [count] next unmatched) . (See page 396.)
[count]] /	-or -
[count]] *	Move forward to the end of the [count] comment from the cursor. (See page 396.)
[count]] (Move forward to the count next unmatched (. (See page 395.)
[count]] [Move count sections forward or to the next } in column 1. (See pages 396.)
[count]]]	Move count sections forward or to the next { in column 1. (See pages 190, 396.)
[count]] {	Finds the count previous unmatched { .(See page 395.)
[count]] }	Finds the count previous unmatched } . (See page 395.)
[count]] c	Go to the [count] previous change in the current diff. (See page 384.)
[count]] d	List the definition of the macro. Start search at the current cursor position. (See pages 127, 263, 402.)
[count]] D	List all definitions of the macro whose name is under the cursor. Start the list with the first next definition. (See page 127.)
] f		Deprecated. Use gf instead. (Same as: gf , and depreciated command [f . See page 399.)
[count]] i	Display the [count] line that contains the keyword under the cursor. The search starts from the current cursor position. (See page 403.)
] I		List all lines in the current and included files that contain the word under the cursor starting at the current location. (See page 403.)
] m		Search forward for the start of a method. (See page

The Vim Tutorial and Reference

	396.)
]M	Search forward for the end of a method. (See page 396.)
["{register}"]]p	Put the {register} in the buffer like the P command, but adjust the text to fit the indent of the current line. (Same as:]<MiddleMouse>.) (See page 367.)
["{register}"]]P	Put the {register} in the buffer like the P command, but adjust the text to fit the indent of the current line. (Same as]<MiddleMouse>, [p, [P. See page 367.)
[count]]s	Search for the next misspelled word starting at the current location. A [count] cause the command to repeat [count] times. (See page 185.)
[count]]S	Search for the next misspelled word starting at the current location, but do not stop at bad words or uncommon words. A [count] cause the command to repeat [count] times. (See page 185.)
]z	Move to the end of the current open fold. (See page 390.)
[count]_	Move to the first printing character of the [count]-1 line below the cursor. (See page 265.)
`{mark}	Go to the mark named mark. Cursor is positioned exactly on the mark. (See page 72.)
`{mark} (backtick)	Go to the line containing mark. Position the cursor at the first non-blank character on the line. (See page 72)
[count] {	Move backward [count] paragraphs. (See page 189.)
[count]	Move to the column [count] on the current line. (See page 330.)
[count] }	Move forward [count] paragraphs. (See page 177, 189, 234.)
~{motion}	Change the case of the indicated characters. (This version of the command depends on the ' 'tildeop' option being on. (The default is off.) (See page 55.)
[count]~	Change the case of count characters. (This version of the command depends on the ' 'tildeop' option being off (the default). (See pages 268.)
[count]£	Search for the word under the cursor, backward. (Same as #, <S-RightMouse>.) (See page 295.)
0 (zero)	Move to the first character on the line. (See pages 44, 263, 329.)
[count]a{text}<Esc>	Insert text starting after the character under the cursor. If a [count] is specified, the text is inserted count times. (See page 36, 41, 56.)

The Vim Tutorial and Reference

[count]A{text}<Esc>	Append the text on to the end of the line. (See page 277.)
[count]b	Move backward count words. (Same as: < S-Left >.) (See page 42.)
[count]B	Move count WORDS backward. (Same as: < C-Left >. See page 263.)
c{motion}	Delete from the cursor to the location specified by the {motion} then enter insert mode. (See pages 51, 103, 233.)
[count]C	Delete from the cursor to the end of the current line and [count] -1 more lines, and then enter insert mode. (See page 52, 103, 275, 276.)
[count]cc	Delete [count] entire lines (default = 1) and enter insert mode. (See page 52.)
["{register}"] d{motion}	Delete from the cursor location to where {motion} goes. (See pages 49, 50, 53, 69, 70, 73, 234, 275.)
do	Copy the current diff from the other window to this one. (See page 122.)
dp	Copy the current diff from the this window to the other one. (See page 122.)
[count]D	Delete from the cursor to the end of the line. If a [count] is specified, delete an additional [count] -1 lines. (See pages 50, 275.)
["{register}"] [count]dd	Delete [count] lines. (See pages 36, 49, 313.)
[count]e	Move [count] words forward, stop at the end of the word. (See page 260.)
[count]E	Move [count] WORDS forward to the end of the WORD. (See page 263.)
[count]f{char}	Search forward for character {char} on the current line. Stop on the character. (See pages 44, 53, 264, 282.)
[count]F{char}	Search backward for character {char} on the current line. Stop on the character. (See page 44.)
[count]G	Go to the line [count] . If no line is specified, go to the last line in the file. (Same as: < C-End >. See pages 46, 226, 233, 266, 557.)
[count]g<Down>	Move down one line on the screen. (Same as: gj .) (See page 330.)
g<End>	Move to the rightmost character on the screen. (Same as: g\$.) (See page 329.)

The Vim Tutorial and Reference

g<Home>	Move to the leftmost character on the screen. (In other words, move to column 1.) (Same as: g0 . See page 329.)
g<LeftMouse>	Jump to the location of the tag whose name is under the cursor. (Same as: <C-LeftMouse>, CTRL-J .) (See page 131, 132, 134, 240.)
[count]g<RightMouse>	Jump to a previous entry in the tag stack. (Same as: <C-RightMouse>, CTRL-T .) (See page 133, 134.)
[count]g<Up>	Move up lines in one the screen space. (Same as: gk .) (See page 330.)
g CTRL-[Do a :tjump on the word under the cursor. (See page 136.)
g CTRL-G	Display detailed information about where you are in the file. (See page 226.)
g CTRL-H	Start select block mode. (See page 360.)
g@{Motion}	Call the function specified by the ' operatorfunc ' to process the text. (See page 510.)
[count]g-	Go to the [count] older text state. (See page 287.)
[count]g+	Go to the [count] newer text state. (See page 287.)
[count]g,	Go [count] forward cursor positions in the change list (Seepage 267.)
[count]g;	Go [count] backward cursor positions in the change list (Seepage 267.)
[count]gf	Search for the word under the cursor, backward. Unlike f , this finds partial words. (Same as: g#.) (See page 295.)
g\$	Move to the rightmost character on the screen. (Same as: g<End> .) (See page 329.)
[count]g*	Search for the word under the cursor, forward. Unlike * , this finds partial words. (See page 295.)
g?{motion}	Encrypt the text from the current cursor location to where {motion} takes you using rot13 encryption. (See page 191.)
[count]g??	Encrypt the lines using the rot13 encryption. (Same as: g?g? . See page 191.)
[count]g?g?	Encrypt the lines using the rot13 encryption. (Same as: g??. See page 191.)
g#	Search for the word under the cursor, backwards. Unlike # , this finds partial words. (Same as: gf . See page 295.)
g]	Do a :tselect on the word under the cursor. (See page 136.)

The Vim Tutorial and Reference

g^	Move to the leftmost printing character visible on the current line. (See page 329.)
[count]g;	Go [count] cursor positions back in the change list. (See page 267.)
g~{motion}	Reverse the case of the text from the cursor to {motion} . (See pages 283.)
[count]g~g~	-or -
[count]g~~	Reverse the case of the entire line. If a [count] is specified, change the case of [count] lines. (See page 283.)
g0 (zero)	Move to the leftmost character on the screen. (In other words, move to column 1.) (Same as: g<Home> .) (See page 329.)
gf	Search for the word under the cursor, backwards. Unlike #, this finds partial words. (Same as: g# . See page 295.)
ga	Print the ASCII value of the character under the cursor. (Same as: :as , :ascii . See page 226.)
gd	Find the local definition of the variable under the cursor. (See pages 126.)
gD	Find the global definition of the variable under the cursor. (See pages 126.)
[count]ge	Move [count] words backward stopping on the end of the word. (See page 260.)
[count]gE	Move [count] WORDS backward to the end of the WORD. (See page 263.)
gf	Edit the file whose name is under the cursor. If the file is not in the current directory, search the directory list specified by the ' path ' option. (Same as: [f,]f . See page 263.)
[count]gg	Move to line count. Default is the first line. (Same as: <C-Home> . See page 226.)
gh	Start select mode characterwise. (See page 360.)
gH	Start select mode linewise. (See page 360.)
[count]gI{text}<Esc>	Insert text in column 1, [count] times. (See page 277.)
[count]gj	Move down one line on the screen. (Same as: g<Down> .) (See page 330.)
[count]gJ	Join lines. No spaces are put between the assembled parts. If a [count] is specified, [count] lines are joined (minimum of two lines). (See page 278.)
[count]] gk	Move up lines in the screen space. (Same as: g<Up> .) (See page 330.)

The Vim Tutorial and Reference

gm	Move to the middle of the screen. (See page 329.)
[count]go	Go to [count] byte of the file. (Same as: :go , :goto . See page 226.)
[" " {register}] gp	Paste the text before the cursor, but do not move the cursor. (See page 309.)
[" " {register}] gP	Paste the text after the cursor, but do not move the cursor. (See page 309.)
gq{motion}	Format the text from the line the cursor is on to the line where {motion} takes you. (See pages 156, 177, 181, 183, 374.)
gqq	Format the current line. (Same as: gqqq . See page 177.)
gqqq	Format the current line. (Same as: gqq . See page 177.)
qΩ	Enter :ex mode. (See page 159.)
[count]gr{character}	Replace the virtual character under the cursor with {character} . (See pages 281.)
[count]gR{string}<Esc>	Enter virtual replace mode until <Esc> is pressed. (See page 281.)
[count] gs	Sleep for the specified number of seconds. (Same as: :s1 , :sleep .) (See page 227.)
[count] gt	Go to the next tab. If a [count] is specified, go to the given tab. (Same as :tabn :tabnext , <C-PageDown> .) (See page 97.)
[count] gT	Go to the previous tab. If a [count] is specified, go to the given tab. (Same as :tabN :tabNext , :tabp , :tabprevious , <C-PageUp> .) (See page 97.)
gu{motion}	Lowercase the text from the cursor to {motion} . (See page 283.)
gU{motion}	Uppercase the text from the cursor to {motion} . (See page 283.)
[count]gugu	-or -
[count]guu	Lowercase the entire line. If a [count] is specified change the case of [count] lines. (See page 283.)
[count]gUGU	-or -
[count]gUU	Uppercase the entire line. If a [count] is specified change the case of [count] lines. (See page 283.)
gv	Repeat the last visual-mode selection. (See page Error: Reference source not found352.)
gv	Do not automatically reselect the selected text. (See page 362.)
[count]h	Move cursor left. (Same as: <BS> , <Left> , CTRL-H ,

The Vim Tutorial and Reference

	CTRL-K.) (See page 31, 32, 38, 49, 268.)
[count]H	Move to the cursor to the top of the screen. If a [count] is specified, move to the count line from the top. (See page 266, 557.)
[count]i{text}<Esc>	Insert text starting before the character under the cursor. If a count is specified, the text is inserted count times. (Same as: < Insert >.) (See pages 30, 32, 56.)
[count]I{text}<Esc>	Insert the text at the beginning of the line. (See page 277.)
[count]j	Down. (Same as: < Down >, < NL >, CTRL-J , CTRL-N . See pages 31, 32, 38, 330.)
[count]J	Join lines. Spaces are put between the assembled parts. If a [count] is specified, [count] lines are joined (minimum of 2 lines). (See pages 54, 179, 278.)
[count]k	Move cursor up. (Same as: < Up >, CTRL-P .) (See pages 31, 32, 38, 41, 122.)
[count]K	Run the <i>man</i> command on the word under the cursor. If a [count] is specified, use [count] as the section number. On Microsoft Windows, by default, this command performs a : help on the word under the cursor. (See page 130, Error: Reference source not found131.)
[count]l	Right. (Same as: < Right >, < Space >. See page 31, 32, 38, 268, 328.)
[count]L	Move the cursor to the bottom of the screen. If a count is specified, move to the [count] line from the bottom. (See page 266, 557.)
m{letter}	Mark the current text with the name {letter} . If {letter} is lowercase, the mark is local to the buffer being edited. In other words, just the location in the file is marked, and you have a different set of marks for each file.
	If an uppercase letter is specified, the mark is global. Both the file and the location within are marked. If you execute a "go to mark(`)" command to jump to a global mark, you may switch files. (See pages 72, 75, 139, 161, 234, 235, 239, 319.)
M	Move to the cursor to the middle of the screen. (See page 266, 557.)
[count]n	Repeat last search. Search in the same direction. (See

The Vim Tutorial and Reference

	pages 64, 65, 233.)
[count]N	Repeat last search. Search in the reverse direction. (See pages 65, 65.)
[count]o	Open a new line below the cursor and put the editor into insert mode. (See page 37, 181, 375.)
[count]O	Open a new line above the cursor and put the editor into insert mode. (See page 37, 181, 375.)
["{register}"] p	Paste the text in the unnamed register ("") after the cursor. (If the register contains complete lines, the text will be placed after the current line.) (See pages 69, 70, 71, 75, 232, 234, 309, 367.)
["{register}"] P	Paste the text in the {register} before the cursor. If no {register} is specified, the unnamed register is used. (Same as: <MiddleMouse>. See pages 71, 235, 236, 236, 309, 313, 367.)
q/	Open a command window and allow the user to browse through the forward search history. (See page 61.)
q:	Open a command window and allow the user to browse through the command history. (See page 427.)
q?	Open a command window and allow the user to browse through the reverse search history. (See page 64.)
q{character}	Begin recording keys in register {character} (character is a-z). Stop recording with a q command. (See page 55.)
Q	Enter ex mode. (See page 159.)
[count]r{char}	Replace count characters with the given character. (See pages 54, 55, 280.)
[count]R{text}<Esc>	Enter replace mode and replace each character in the file with a character from {text}. If a [count] is specified, repeat the command [count] times. (See page 279.)
[count]s	Delete [count] characters and enter insert mode. (See page 275.)
[count]S	Delete [count] lines and enter insert mode. (See page 276.)
[count]t{char}	Search forward for character {char} on the current line. Stop one before the character. (See page 45.)
[count]T{char}	Search backward for character {char} on the current line. Stop one after the character. (See page 45.)
u	Undo the last change. (Same as: <Undo>.) (See page 34, 286.)
U	Undo all the changes on the last line edited. (A second

The Vim Tutorial and Reference

	u redoes the edits.) (See page 34, 313.)
v	Start visual character mode. (See pages 49, 51, 53, 71, 76, 99, 100, 103, 103, 130, 373.)
v	Start visual line mode. (See pages 71, 74, 100, 122, 140, 236, 236, 236, 237, 365.)
[count]w	Move count words forward. (Same as: < S-Right >.) (See pages 42, 49, 49, 51.)
[count]W	Move count WORDS forward. (Same as: < C-Right >.) (See page 263.)
["[register}]" "[count]x	Delete [count] characters. (Default = 1.) Deleted text goes into {register} or the unnamed register if no register specification is present. (Same as: < Del >.) (See pages 33 , 36, 69, 232, 276.)
["{register}]" [count]X	Delete the characters before the cursor. (See pages 276.)
xp	Exchange the character under the cursor with the next one. Useful for turning "teh" into "the". (See pages 70, 232.)
["{register}]" y{motion}	Yank the text from the current location to {motion} into the register named {register} . Lowercase register specifications cause the register to be overwritten by the yanked text. Uppercase register specifications append to the contents of the register. (See pages 74, 75, 103, 235.)
["{register}]" [count]Y	-or -
["{register}]" [count]yy	-or -
[count] ["{register}]" yy	Yank [count] lines into the register named {register} . Lowercase register specifications cause the register to be overwritten by the yanked text. Uppercase register specifications append to the contents of the register. (See pages 74, 76, 103, 311, 313.)
z{height}<CR>	Make the window {height} lines high. (See page 228.) (Same as: z<Enter> , see next entry..)
[count]z<CR>	
[count]z<Enter>	Position the line count at the top of the screen. If no [count] is specified, the current line is used. Cursor is positioned on the first nonblank character after this command. (Same as: z:<CR> . See page 272.)
[count]z<Left>	Scroll the screen [count] characters to the right. (Same as: zh . See page 241.)

The Vim Tutorial and Reference

[count]z<Right>	Scroll the screen [count] characters to the left. (Same as: z1 . See page 330.)
[count]z-	Position the line [count] at the bottom of the screen. If no [count] is specified, the current line is used. Cursor is positioned on the first nonblank character after this command. (See page 273.)
[count]z.	Position the line [count] at the middle of the screen. If no [count] is specified, the current line is used. Cursor is positioned on the first nonblank character after this command. (See pages 274.)
[count]z=	Display a list of suggested corrections for the misspelled word under the cursor. If [count] is present, automatically the [count] entry from the list. (See page 184.)
za	Open a fold if it's closed. Close it if it's open. Works on a single level of folding. (See page 390.)
zA	Open a fold if it's closed. Close it if it's open. Works on all levels of folding. (See page 390.)
[count]zb	Position the line [count] at the bottom of the screen. If no [count] is specified, the current line is used. Cursor is positioned on the same column after this command. (See page 273.)
zc	Close one fold at the cursor. (See page 124, 389.)
zC	Close all folds at the cursor. (See page 389.)
zd	Delete the current fold. (See page 389.)
zD	Delete the current fold and all nested folds. (See page 389.)
ze	Open all the folds currently visible in the window. (See page 389.)
zf{motion}	Create a fold. (See page 122, 387.)
[count]zF	Create a fold starting at the cursor and containing [count] lines. (See page 389.)
[count]zg	Add the word under the cursor to the list of good words. If a [count] is specified, add it to the [count] word file. (See page 186.)
zG	Add the word under the cursor to the internal list of good words. This list is temporary and is not stored between sessions. (See page 186.)
[count]zh	Scroll the screen [count] character to the right. (Same as z<Left> . See page 330.)
zi	Invert ' foldenable '. If this option is set, folding is allowed, if not all text is displayed. (See page 390.)
[count] zj	Move down to the start of the next fold. The [count]

The Vim Tutorial and Reference

	parameter tells <i>Vim</i> how many folds to move down. (See page 390.)
[count] zk	Move up to the end of the previous fold. The [count] parameter tells <i>Vim</i> how many folds to move up. (See page 390.)
[count] zl	Scroll the screen count character to the left. (Same as z<Right> . See page 330.)
zm	Increase the folding by one level, but subtracting one to ' foldlevel '. (See page 124.)
zM	Open all folds by setting the ' foldlevel ' to 0. (See page 125.)
zn	Turn off ' foldenable ' opening all folds. (See page 390.)
zN	Turn on ' foldenable ' causing all folds to return to the current open / closed state. (See page 390.)
zo	Open a fold at the cursor. (See page 123, 124, 389, 390.)
zO	Open all folds under the cursor. (See page 389.)
zL	Move the screen $\frac{1}{2}$ screen-full left. (See page 330.)
zQ	Abandon the file discarding all edits. (Same as : quit! .) (See page 288.)
zr	Decrease the folding by one level, but adding one to ' foldlevel '. (See page 124.)
zR	Remove all folding by setting ' foldlevel ' to the highest fold level. (See page 125.)
[count] zt	Position the line count at the top of the screen. If no [count] is specified, the current line is used. Cursor is positioned on the same column after this command. (See pages 273.)
[count] zug	Remove the word under the cursor from the good list. If a [count] is specified, remove it from the [count] word file. (See page 186.)
zuG	Remove the word under the cursor from the internal good list. (See page 186.)
[count] zuw	Remove the word under the cursor from the wrong list. If a [count] is specified, remove it from the [count] word file. (See page 186.)
zuW	Remove the word under the cursor from the internal wrong list. (See page 186.)
zv	Undo enough folds to make the line the cursor is on visible. (See page 124.)
[count] zw	Add the word under the cursor to the list of wrong words. If a [count] is specified, add it to the [count]

The Vim Tutorial and Reference

	word file. (See page 186.)
zw	Add the word under the cursor to the internal list of good words. This list is temporary and is not stored between sessions. (See page 186.)
zx	Clear all manually opened and closed folds. (See page 124.)
zz	Write file and exit. (See pages 35, 38, 96, 224, 243.)

Motion Commands

[count]a"	Select matching "" pair, including the "".
[count]a'	Select matching " " pair, including the ' '.
[count]a(From with text enclosed in (), select the text up to and including the () . (See page 354.)
[count]a)	Select matching) pair, including the). (See page 354.)
[count]a<]	Select matching <> pair, include the <>.
[count]a>	Select matching > pair, including the >.
[count]a[Select matching [] pair, include the [].
[count]a]	Select matching] pair, including the].
[count]a`	Select matching `` pair, including the ``.
[count]a{	Select matching { } pair, including the { }.
[count]a}	Select matching } pair, including the }.
[count]ab	From with text enclosed in (), select the text up to and including the () . (See page 354.)
[count]aB	Select matching } pair, including the }.
[count]ap	Select a paragraph and the following space.
[count]as	Select a sentence (and spaces after it).
[count]aw	Select a word and the space after it. (Word is defined by the ' iskeyword ' option . (See page 353, 354.)
[count]aw	Select a word and the space after it. (Word is defined to be any series of printable characters.) (See page 354.)
[count]at	Select the enclosing XML tag block (<foo> ... </foo>) including the tags. (See 354.)
[count]i"	Select matching "" pair, not including the "".
[count]i'	Select matching " " pair, not including the ' '.
[count]i(From with text enclosed in (), select the text up to but not including the () .
[count]i)	Select matching) pair, not including the).
[count]i<	Select matching <> pair, excluding the <>.
[count]i>	Select matching > pair, not including the >.
[count]i[Select matching [] pair, excluding the [].
[count]i]	Select matching] pair, not including the].
[count]i`	Select matching `` pair, not including the ``.

The Vim Tutorial and Reference

[count]i{	Select matching {} pair, excluding the {}. (See page 123, 130.)
[count]ib	From with text enclosed in (), select the text up to but not including the (). (See page 354.)
[count]iB	Select matching {} pair, excluding the {}. (See page 354.)
[count]ip	Select a paragraph only. (See page 177, 354.)
[count]is	Select the sentence only. Do no select whitespace after a sentence. (See page 354.)
[count]it	Select the enclosing XML tag block (<foo> ... </foo>) excluding the tags. (See page 354.)
[count]iw	Select inner word (the word only). (Word is defined by the ' iskeyword ' option.) (See page Error: Reference source not found353, 354.)
[count]iw	Select inner word (the word only). (Word is defined to be any series of printable characters.) (See page 354.)

Appendix D: Command-Mode Commands

:!	: ! {cmd}
Execute shell command. (See page 446.)	
:!!	: !!
Repeat last :! {cmd}. (See page 446.)	
:#	: [range] #
Print the lines with line numbers. (See page 433.)	
:&	: [count] &
Repeat the last :substitute command on the next [count] lines. (Default = 1.) (See page 435.)	
:& cmd-zm(7-1)	: [range] & [flags] [count]
Repeat the last substitution with a different [range] and [flags]. (See page 435.)	
:*	[line] *{register}
Execute the contents of the {register} as an ex-mode command. (Same as: :@.) (See page 444.)	
:<	: [line] < {count}
Shift lines left. (See page 445.)	
:=	:=
Print line number. (See page 443.)	
:>	: [line] > {count}
Shift lines right. (See page 445.)	
@	: [line] @{register}
Go to line and execute {register} as a command. (Same as: :*.) (See page 444.)	

The Vim Tutorial and Reference

:@:	:[line] @:
Repeat the last command-mode command. (See page 444.)	
:@@	:[line] @@
Repeat the last :@{register} command. (See page 444.)	
:~	: [range]~ {flags} {count}
Repeat the last substitution, but the last search string as the {from} pattern instead of the {from} from the last substitution. (See page 437.)	
:a	: [line] a
Insert text after the specified line. (Default = current.) (Same as: :append.) (See page 432.)	
:ab	:ab
List all abbreviations. (Same as: :abbreviate.) (See pages 149, 419.)	
:ab	:ab {lhs} {rhs}
Define an abbreviation. When {lhs} is entered, put {rhs} in the text. (Same as :abbreviate.) (See page 148, 157, 241, 418.)	
:abbreviate	:abbreviate
List all abbreviations. (Same as: :ab.) (See pages 149, 419.)	
:abbreviate	:abbreviate {lhs} {rhs}
Define an abbreviation. When {lhs} is entered, put {rhs} in the text. (Same as :ab.) (See pages 148, 157, 241, 418.)	
:abc	:abc
:abclear	:abclear
Remove all abbreviations. (See page 418.)	
:abo	:abo {cmd}
:aboveleft	:aboveleft {cmd}
Execute the Vim command {cmd}. If the command splits a window, the window is opened above the current one or to its left (overriding	

The Vim Tutorial and Reference

any any split options that are currently set.) (Same as :**lefta**, :**leftabove**)

(See page 346.)

:**a1**

:**[count] al**

:**all**

:**[count] all**

Open a window for all the files being edited. When a **[count]** is specified, open up to **[count]** windows. (Note that the **[count]** can be specified after the command, such as ":**all [count]**.) (Same as :**s1**, :**sall**.)

(See page 339.)

:**am**

:**{priority} am {menu-item} {command-string}**

:**amenu**

:**{priority} amenu {menu-item} {command-string}**

Define a menu item that's valid for all modes .

(See page 468.)

The following characters are automatically inserted for some modes:

Mode	Prefix Character Inserted	Meaning
Normal	(Nothing)	?N/A
Visual	<Esc>	Exit visual mode
Insert	CTRL-O	Execute one normal command.
Command Line	CTRL-C	Exit command-line mode
Operator pending mode	<ESC>	End operator-Pending

:**an**

:**{priority} an {menu-item} {command-string}**

:**anoremenu**

:**{priority} anoremenu {menu-item} {command-string}**

Perform a :**amenu** command in which the **{command-string}** is not remapped.

(See page 473.)

:**append**

:**[line] append**

Insert text after the specified line. (Default = current). (Same as :**a.**)

(See page 432.)

:**ar**

:**ar**

List the files being edited. The name of the current file is enclosed in square brackets ([]). (Same as :**args**.)

The Vim Tutorial and Reference

(See page 78, 246, 316, 317.)

:arg *[count]arga {file}*
:argadd *[count]argadd {file}*

Add the **{file}** to the argument list. If **[count]** is given, the **{file}** will be added after the **[count]** parameter. If no **[count]** is specified, the **{file}** will be added after the current file.

(See page 318.)

:argd *:argd {pattern}*
:argdelete *:argadelete {pattern}*

Delete all the files which match **{pattern}** from the argument list.

(See page 318.)

:argd *:{range}argd*
:argdelete: *:{range}argadelete*

Delete all the files specified **{range}** from the argument list.

(See page 318.)

:argdo *:argdo[!] {cmd}*
Execute the command **{cmd}** for each argument on the argument list.

(See page 78.)

:arge *:arge[!] [++opt] [+cmd] {file-name}*
:argedit *:argedit[!] [++opt] [+cmd] {file-name}*

A combination of **:argadd** and **:edit**. Add a file to the argument list (if it's not already there and then edit it.)

(See page 318.)

:argg *:argg[!] [++opt] [+cmd] {file-list}*
:argglobal *:argglobal[!] [++opt] [+cmd] {file-list}*

Define a new global argument list and use it for this window. If a **{file-list}** is not specified, use the existing global argument list.

(See page 319.)

:arg *:argl[!] [++opt] [+cmd] {file-list}*
:arglocal *:arglocal[!] [++opt] [+cmd] {file-list}*

Define a new local argument list and use it for this window. If a **{file-list}** is not specified, use the existing global argument list, but make a local copy.

The Vim Tutorial and Reference

(See page 319.)

:args

:args

List the files being edited. The name of the current file is enclosed in square brackets ([]). (Same as: **:ar.**)

(See pages 79, 317.)

:args

:args {file-list}

Change the list of files to **{file-list}** and start editing the first one. (Same as: **:ar.**)

(See page 78, 246, 316, 317.)

:argu

:argu {number}

:arguement

:argument {number}

Edit the **{number}** file in the file list. _

(See page 316.)

:as

:as

:ascii

:ascii

Print the number of the character under the cursor. (Same as: **ga.**)

(See page 226.)

:au

:au

List all the autocommands. (Same as: **:autocmd.**)

(See page 208.)

:au

:au {group} {event} {pattern}

Lists the autocommands that match the given specification. If * is used for the event, all events will match.

(See pages 156.)

:au

:au {group} {events} {file_pattern} nested {command}

Define an autocommand to be executed when one of the **{events}** happens on any files that match **{pattern}**. The group parameters enables you to put this command in a named group for easier management. The nested flag allows for nested events. (Same as **:autocmd.**)

(See page 117, 203, 203, 208, 596.)

:au

:au !

Delete all the autocommands. (Same as **:autocmd!**)

(See page 209.)

The Vim Tutorial and Reference

:au	:au! {group} {event} {pattern} nested {command}
	Remove any matching autocmds and replace them with a new version. (See page 209.)
:aug	:aug {name}
:augroup	:augroup {name}
	Start an autocmd group. The group ends with a :augroup END statement. (See page 203.)
:aun	:aun {menu-item}
:aunmenu	:aunmenu {menu-item}
	Remove the menu item named {menu-item} that was defined with an :amenu command. The wildcard * will match all menu items. (Same as: :aun .) (See page 473.)
:autocmd	:autocmd
	List all the autocmds. (Same as: :au .) (See page 208.)
:autocmd	:autocmd {group} {event} {pattern}
	Lists the autocmds that match the given specification. If * is used for the event, all events will match. (Same as: :au .) (See pages 156, 203, 203.)
:autocmd	:autocmd {group} {events} {file_pattern} [nested] {command}
	Define an autocmd to be executed when one of the {events} happens on any files that match {pattern} . The group parameters enables you to put this command in a named group for easier management. The nested flag allows for nested events. (Same as :au .) (See pages 117, 147, 208, 596.)
:autocmd	:autocmd !
	Delete all the autocmds. (Same as :au!) (See page 209.)
:autocmd	:autocmd! {group} {event} {pattern}
	Remove the specified autocmds. (Same as :au!) (See page 209.)

The Vim Tutorial and Reference

:autocmd **:autocmd!** {group} {event} {pattern} [nested] {command}

Remove any matching autocommands and replace them with a new version.

(See page 209.)

:b

: [count] b[!]

Switch the current window to buffer number count. (If a count is not specified, the current buffer is used.) If ! is specified, if the switch abandons a file, any changes might be discarded.

(An alternative version of this command has count at the end--for example, **:buffer 5**.) (Same as: **:buffer**.)

(See page 92.)

:b

:b[!] {file-name}

Switch the current window to the buffer containing {file-name}. If ! is specified, if the switch abandons a file, any changes might be discarded. (Same as: **:buffer**.)

(See page 92.)

:ba

: [count] ba

Open a window for each buffer. If a count is specified, open at most count windows. (Same as: **:ball**, **:sba**, **:sball**.)

(See page 343.)

:bad

:bad [+line] {file}

:badd

:badd [+line] {file}

Add the file to the buffer list. If a +line is specified, the cursor will be positioned on that line when editing starts.

(See page 342.)

:ball

: [count] ball

Open a window for each buffer. If a count is specified, open at most count windows. (Same as: **:ba**, **:sba**, **:sball**.)

(See page 343.)

The Vim Tutorial and Reference

```
:bd                               :bd[!] {file}
:bdelete                          :bdelete[!] {file}
                                :[n] bd[!]
                                :[n] bdelete[!]
                                :[n,m] bd[!]
                                :[n,m] bdelete[!]
                                :bd[!] [n]
                                :bdelete[!] [n]
```

Delete the specified buffer, but leave it on the buffer list. (Reclaims all the memory allocated to the buffer and closes all windows associated with.) If the override option (!) is specified, any changes made are discarded. If **{file}** is specified, the buffer for that file is deleted. A buffer number **[n]** or a range of buffer numbers **[n,m]** can be specified as well.

(See page 342.)

```
:be                               :be {mode}
:befile                           :bemouse {mode}
```

Sets the behavior of the mouse. The **{mode}** is either **xterm** for X Windows System-style mouse usage or **mswin** for Microsoft Windows-style usage.

(See page 169.)

```
:bel                             :bel {cmd}
:belowright                      :belowright {cmd}
```

Execute the *Vim* command **{cmd}**. If the command splits a window, the window is opened below the current one or to its right (overriding any any split options that are currently set.)

(See page 346.)

```
:bf                               :bf[!]
:bfirst                          :bfirst[!]
```

Go to the first buffer in the list. (Same as: **:brewind**, **:br**.)

(See page 93.)

```
:bl                               :bl[!]
:blast                          :blast[!]
```

Go to the last buffer in the list. (Same as: **:bl**.)

(See page 93.)

The Vim Tutorial and Reference

:bm	:bm [count]
:bmodified	:bmodified [count]
	Go to count-modified buffer. (Same as: :bm .)
	(See page 93.)
:bn	: [count] bn[!]
	Go to the next buffer. If ! is specified, if the switch abandons a file, any changes might be discarded. If a [count] is specified, go to the [count] next buffer. (Same as: :bnext .)
	(See page 93.)
:bN	: [count] bN[!]
	Go to previous buffer. If a [count] is specified, go to the [count] previous buffer. (Same as: :bNext , :bp , :bprevious .)
	(See page 93.)
:bnext	: [count] bnext[!]
	Go to the next buffer. If ! is specified, if the switch abandons a file, any changes might be discarded. If a [count] is specified, go to the [count] next buffer.
	(See page 93.)
:bNext	: [count] bNext[!]
	Go to previous buffer. If a [count] is specified, go to the [count] previous buffer. (Same as: :bN , :bNext , :bp , :bprevious .)
	(See page 93.)
:bo	:bo {cmd}
:botright	:botright {cmd}
	Execute the <i>Vim</i> command {cmd} . If the command splits a window horizontally, a new full width window is created at the bottom of the screen. Commands that cause a vertical split create a full height window at the right of the screen. This overrides any any split options that are currently set.
	(See page 346.)
:bp	: [count] bp
:bprevious	: [count] bprevious
	Go to previous buffer. If a [count] is specified, go to the [count] previous buffer. (Same as: :bN , :bNext , :bf , :bfirst .)
	(See page 93.)

The Vim Tutorial and Reference

:br	:br[!]
Go to the first buffer in the list. (Same as: :brewind , :bf , :bfirst .)	
(See page 93.)	
:brea	:brea
:break	:break
Break out of a loop.	
(See page 496.)	
:breaka file	:breaka file [line] {file-name}
:breakadd file	:breakadd file [line] {file-name}
Set a breakpoint that will be triggered when the specified file is read by a :source command. If [line] is specified, stop on that line, otherwise stop on the first line.	
(See page 505.)	
:breaka func	:breaka func [line] {function-name}
:breakadd func	:breakadd func [line] {function-name}
Create a breakpoint in the specified {function-name} . If no [line] is specified, the breakpoint will be set at the start of the function, otherwise it is set at the specified line.	
(See page 503, 504.)	
:breaka here	:breaka here
:breakadd here	:breakadd here
Add a breakpoint at the current file and line.	
(See page 505.)	
:breakd	:breaka {number}
:breakdel	:breakadd {number}
Delete a breakpoint specified by number. The special number * (star) deletes all breakpoints.	
(See page 505.)	
:breakd file	:breakd file [line] {file-name}
:breakdel file	:breakdel file [line] {file-name}
Delete a file type breakpoint.	
(See page 505.)	

The Vim Tutorial and Reference

:breakd func	:breakd func [line] {function-name}
:breakdel func	:breakdel func [line] {function-name}
Delete function breakpoint. (See page 505.)	
:breakd here	:breakd here
:breakdel here	:breakdel here
Delete a breakpoint at the current location. (See page 505.)	
:breakl	:breakl
:breaklist	:breaklist
List breakpoints. (See page 504.)	
:brewind	:brewind[!]
Go to the first buffer in the list. (Same as: :br , :bf , :bfirst .) (See page 93.)	
:bro	:bro {command}
Open a file browser window and then run {command} on the chosen file. (Same as: :browse .) (See page 475.)	
:bro set	:bro set
Enter an option browsing window that enables you to view and set all the options. (Same as: :browse set , :opt , :options .) (See page 479.)	
:browse	:browse {command}
Open a file browser window and then run {command} on the chosen file. (Same as: :bro .) (See page 475.)	
:browse set	:browse set
Enter an option browsing window that enables you to view and set all the options. (Same as: :bro set , :opt , :options .) (See page 479.)	
:buffer	: [count] buffer[!]
Switch the current window to buffer number [count] . (If a [count] is not specified, the current buffer is used.) If ! is specified, if the switch abandons a file, any changes might be discarded. (An	

The Vim Tutorial and Reference

alternative version of this command has **{count}** at the end--for example, **:buffer 5.** (Same as: **:b.**)

(See page 92.)

:buffer **:buffer[!] {filename}**

Switch the current window to the buffer containing **{filename}**. If ! is specified, if the switch abandons a file, any changes might be discarded. (See page 92.)

:buffers

List all the specified buffers. (Same as: **:files**, **:ls**.)

(See pages 90.)

:bun **:bun[!] {file}**

:bun **:[n]bun[!]**

:bun **:[n,m]bun[!]**

:bun **:bun[!] [n]-**

:bunload **:bunload[!] {file}**

:bunload **:[n]bunload[!]**

:bunload **:[n,m]bunload[!]**

:bunload **:bunload[!] [n]**

Unload the specified buffer. If the override option is specified, if there are any changes, discard them.

(See page 342.)

:bw **:bw[!] {file}**

:bw **:[n]bw[!]**

:bw **:[n,m]bw[!]**

:bw **:bw[!] [n]-**

:bwipeout **:bwipeout[!] {file}**

:bwipeout **:[n]bwipeout[!]**

:bwipeout **:[n,m]bwipeout[!]**

:bwipeout **:bwipeout[!] [n]**

Wipeout the the specified buffer. If the override option is specified, if there are any changes, discard them. This is similar to unloading a buffer, but everything about the buffer disappears.

(See page 343.)

:c **:[range] c**

Delete the specified lines, and then do a **:insert**. (Same as:

The Vim Tutorial and Reference

:change, :t.)

(See page 431, 445.)

:ca

:ca {lhs} {rhs}

:cabbrrev

:cabbrrev {lhs} {rhs}

Define an abbreviation for command-mode only.

(See page 419.)

:cabc

:cabc

:cabclear

:cabclear

Remove all for command mode.

(See page 419.)

:cad

:cad [buffer-number]

:caddbuffer

:caddbuffer [buffer-number]

Adds the contents of the buffer to the quick fix list. If not buffer is specified the current one is used.

(See page 404, 405.)

:cadde

:cadde[!] {expression}

:caddeexpr

:caddeexpr[!] {expression}

Add the results of **{expression}** to the quick fix buffer.

(See page 404, 405.)

:caddf

:caddf[!] {file}

:caddfile

:caddfile[!] {file}

Add the contents of **{file}** to the quick fix buffer.

(See page 403, 404, 405.)

:cal

:cal [range] cal {name}({argument list})

:call

:call [range] call {name}({argument list})

Call a function.

(See page 500.)

:cat

:cat /{pattern}/

:catch

:catch /{pattern}/

Catch an exception. If no **/pattern/** is specified, all exceptions will be caught.

(See page 497.)

The Vim Tutorial and Reference

:cb	:cb [buffer-number]
:cbuffer	:cbuffer [buffer-number]
Replace the contents of the quick fix list with the current buffer. If not buffer is specified the current one is used.	
(See page 404, 405.)	
:cc	:cc[!] [number]
Display error number. If the [number] is omitted, display the current error. Position the cursor on the line that caused it.	
(See page 404.)	
:ccl	:ccl
:cclose	:cclose
Close the quick fix window.	
(See page 406.)	
:cd	:cd [path]
Change the directory to the specified path. If path is - , change the previous path. If no path is specified, on UNIX go to the home directory. On Microsoft Windows, print the current directory. (Same :chd, :chdir.)	
(See page 441.)	
:ce	: [range] ce [width]
:center	: [range] center [width]
Center the specified lines. If the width of a line is not specified, use the value of the ' textwidth '. (If ' textwidth ' is 0, 80 is used.)	
(See page 177.)	
:cex	:cex[!] {expr}
:cexpr	:cexpr[!] {expr}
Create a quick fix list from {expr} .	
(See page 404, 405.)	
:cf	:cf[!] [errorfile]
:cfile	:cfile[!] [errorfile]
Read an error list from file. (Default = the file specified by the ' errorfile ' option.) Go to the first error. If the override option is specified and a file switch is made, any unsaved changes might be lost.	
(See page 403, 403, 405.)	

The Vim Tutorial and Reference

:cfir	:cfir[!] [number]
:cfirst	:cfirst[!] [number]
Go the first error in the list. If a number is specified, display that error. (Same as :cr, :crewind.)	
(See page 142, 145, 405.)	
:cg	:cg[!] [<i>error-file</i>]
Create a quick fix list from the contents of <i>[error-file]</i> but do not jump to the first error. (Sames as :cgetfile)	
(See page 404, 405.)	
:cgetb	:cgetb [<i>buffer-number</i>]
:cgetbuffer	:cgetbuffer [<i>buffer-number</i>]
Create a quick fix list from <i>[buffer-number]</i> but do not jump to the first error. If no <i>[buffer-number]</i> is specified, the current one is used.	
(See page 404, 405.)	
:cgete	:cgetx {<i>expr</i>}
:cgetexpr	:cgetexpr {<i>expr</i>}
Create a quick fix list from <i>{expr}</i> but do not jump to the first error.	
(See page 404, 405, 405.)	
:cgetfile	:cgetfile[!] [<i>error-file</i>]
Create a quick fix list from the contents of <i>[error-file]</i> but do not jump to the first error. (Sames as :cgetfile)	
(See page 404.)	
:change	:[range] change
Delete the specified lines, and then do an :insert. (Same as: :c.)	
(See page 445.)	
:changes	:changes
Print the change list.	
(See page 267.)	

The Vim Tutorial and Reference

:chd	:chd [path]
:chdir	:chdir [path]
Change the directory to the specified path. If path is -, change the previous path. If no path specified, on UNIX go to the home directory. On Microsoft Windows, print the current directory. (Same as :cd)	
(See page 441.)	
:che	:che[!]
:checkpath	:checkpath[!]
Check all the #include directives and make sure that all the files listed can be found. If the override option (!) is present, list all the files. If this option is not present, only the missing files are listed.	
(See page 400, 402.)	
:cl	:cl[!] [from], [to]
List out the specified error messages. If the override option is present, list out all the errors. (Same as: :clist .)	
(See page 145, 406.)	
:cla	:cla [number]
:clast	:clast [number]
Go the last error in the list. If a number is specified, display that error.	
(See page 142, 145, 405.)	
:clist	:clist[!] [from], [to]
List out the specified error messages. If the override option is present, list out all the errors. (Same as: :cl .)	
(See page 145, 406.)	
:clo	:clo[!]
:close	:close[!]
Close a window. If this is the last window, exit <i>Vim</i> . The command fails if this is the last window for a modified file, unless the force (!) option is present. (Same as: CTRL-Wc .)	
(See page 84.)	
:cm	:cm
Listing all the mappings for command-line mode maps. (Same as: :cmap .)	
(See page 421, 423, 425.)	

The Vim Tutorial and Reference

:cm	:cm {lhs}
	List the command-line mapping of {lhs} . (Same as: :cmap .) (See page 421, 425.)
:cm	:cm {lhs} {rhs}
	Define a keyboard mapping for command-line mode. (Same as: :cmap .) (See page 421, 425.)
:cmap	:cmap
	Listing all the command-line mode mappings. (Same as: :cm .) (See page 421, 423, 425.)
:cmap	:cmap {lhs}
	List the command-line mode mapping of {lsh} . (Same as: :cm .) (See page 421, 425.)
:cmap	:cmap {lhs} {rhs}
	Define a keyboard mapping for command-line mode. (Same as: :cm .) (See page 421, 425:d(25-1).)
:cmapc	:cmapc-
:cmapclear	:cmapclear
	Clear all the command-mode mappings. (See page 425.)
:cme	: [priority] cme {menu-item} {command-string}
:cmenu	: [priority] cmenu {menu-item} {command-string}
	Define a menu item that is available for command-line mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is {menu-item} , and when the command is selected, the command {command-string} is executed. (See page 467.)
:cn	: [count] cn[!]
	Go to the [count] next error. (Same as: :cnext .) (See pages 246, 405.)
:cN	: [count] cN[!]
	Go the previous error in the error list. (Same as: :cNext , :cp , :cprevious .)

The Vim Tutorial and Reference

(See page 246, 405.)

:cnew

:cnew [count]

:cnewer

:cnewer [count]

Go to the **[count]** newer error list.

(See page 403, 406.)

:cnext

: [count] cnext[!]

Go to the **[count]** next error. (Same as: **:cn.**)

(See pages 246, 405.)

:cNext

: [count] cNext[!]

Go the previous error in the error list. (Same as: **:cN**, **:cp**,

:cprevious.)

(See page 246, 405.)

:cnf

: [count] cnf[!]

Go the first error in the next file. If the override option (!) is present, if there are any unsaved changes, they will be lost.

(See page 405.)

:cNf

: [count] cNf[!]

Go the first last in the previous file. If the override option (!) is present, if there are any unsaved changes, they will be lost.

(See page 405.)

:cnfile

: [count] cnfile[!]

Go the first error in the next file. If the override option (!) is present, if there are any unsaved changes, they will be lost.

(See page 405.)

:cNfile

: [count] cNfile[!]

Go the last error in the previous file. If the override option (!) is present, if there are any unsaved changes, they will be lost.

(See page 405.)

:cno

:cno {lhs} {rhs}

Same as **:cmap**, but does not allow remapping of the **{rhs}**. (Same as: **:cnoremap**.)

(See page 425.)

The Vim Tutorial and Reference

:cnorea	:cnoreabbr	:cnoreab {lhs} {rhs} :cnoreabbr {lhs} {rhs}
Do a :noreabbrev that works in command-mode only. (See page 419, 425.)		
:cnoremap		
Same as :cmap , but does not allow remapping of the {rhs} . (Same as: :cno . (See page 425.)		
:cnoreme	:[priority] cnoreme {menu-item} {command-string}	
:cnoremenu	:[priority] cnoremenu {menu-item} {command-string}	
Like :cmenu , except the {command-string} is not remapped. (See page 473.)		
:co	:[range] co {address}	
Copy the range of lines below {address} . (Same as: :copy , :t). (See page 431.)		
:col	:col {count}	
:colder	:colder {count}	
Go to the {count} older error list. (See page 403, 406.)		
:colo	:colo {name}	
:colorscheme	:colorscheme {name}	
Load the color scheme {name} .		
:com	:com	
List the user-defined commands. (Same as: :command). (See page 509.)		
:com	:com {definition}	
Define a user-defined command. (Same as: :command). (See pages 508.)		
:comc	:comc	
:comclear	:comclear	
Clear all user-defined commands. (See page 509.)		

The Vim Tutorial and Reference

:command	:command
List the user-defined commands. (Same as: :com .) (See page 509.)	
:command	:command {definition}
Define a user-defined command. (Same as: :com .) (See page 508.)	
:con	:con {command}
Start a loop over. (Same as: :continue .) (See page 496.)	
:conf	:conf {command}
:confirm	:confirm {command}
Execute the {command} . If this command would result in the loss of data, display a dialog box to confirm the command. (See page 478.)	
:continue	:continue
Start a loop over. (Same as: :con .) (See page 496.)	
:cope	:cope [height]
:copen	:copen [height]
Open a window for the quick fix list. If [height] is specified, use it as the window height. (See page 146, 406.)	
:copy	: [range] copy {address}
Copy the range of lines below {address} . (Same as: :co , :t .) (See page 431.)	
:cp	: [count] cp[!]
Go to the [count] previous error. (Same as: :cprevious , :cN , :cNext .) (See pages 246, 405.)	
:cpf	: [count] cpf[!]
:cpfile	: [count] cpfile[!]
Go to the last error in the previous file. If a [count] is specified, go back that many files. (See page 405.)	

The Vim Tutorial and Reference

:cprevious	: [count] cprevious[!]
Go to the [count] previous error. (Same as: :cp , :cN , :cNext.)	
(See pages 246 405.)	
:cq	:cq
:cquit	:cquit
Exit <i>Vim</i> with an error code. (This is useful in integrating <i>Vim</i> into an IDE.)	
(See page 144.)	
:cr	:cr[!] [number]
:crewind	:crewind[!] [number]
Go the first error in the list. If a number is specified, display that error. (Same as :cfir , :cfirst .)	
(See page 142, 145, 405.)	
:cs	:cs {arguments}
:cscope	:cscope {argument}
Handle various activities associated with the <i>CScope</i> program.	
(See page 247.)	
:cst	:cst {procedure}
:cstag	:cstag {procedure}
Go to the tag in the <i>CScope</i> database named {procedure} .	
(See page 247.)	
:cu	:cu {lhs}
Remove a command-mode mapping. (Same as: :cunmap .)	
(See page 425.)	
:cuna	:cuna {lhs}
:cunabbreviate	:cunabbreviate {lhs}
Remove the command-line mode abbreviation.	
(See page 419, 425.)	
:cunmap	:cunmap {lhs}
Remove a command-mode mapping. (Same as: :cu .)	
(See page 425.)	

The Vim Tutorial and Reference

:cunm	:cunm
:cunmenu	:cunmenu {menu-item}
Remove the command-mode menu item named {menu-item} . The wildcard * will match all menu items.	
(See page 473.)	
:cw	:cw [height]
:cwindow	:cwindow [height]
If there are errors, open a quick fix window of the specified height. If there are no errors, close the quick fix window.	
(See page 406.)	
:d	: [range] d {register} [count]
Delete text. (Same as: :delete .)	
(See page 427.)	
:deb	:deb {command}
:debug	:debug {command}
Execute {command} in debug mode.	
(See page 503.)	
:debugg	: [0]debugg
:debuggreedy	: [0]debuggreedy
The :debugg command tells Vim to read debug commands from the standard command stream instead of forcing all debug input to come from the user. This is useful for starting a debug session with a script. The :0debugg command cancels this feature and all future debug input must come from the user.	
(See page 506.)	
:delc	:delc {command}
:delcommand	:delcommand {command}
Delete a user-defined command.	
(See page 509.)	
:delete	: [range] delete {register} [count]
Delete text. (Same as: :d .)	
(See page 427.)	
:def	:def {name}
:function	:function {name}
Delete the function named {name} .	

The Vim Tutorial and Reference

(See page 502.)

:delm

:delm {marks}

Delete the specified **{marks}**. (Same as **:delmarks**.)

(See page 311.)

:delm

:delm!

Delete all marks. (Same as **:delmarks**.)

(See page 311.)

:delmarks

:delmarks {marks}

Delete the specified **{marks}**. (Same as **:delm**.)

(See page 311.)

:delmarks

:delmarks!

Delete all marks. (Same as **:delm**.)

(See page 311.)

:di

:di {list}

Display the registers. (Same as: **:display**, **:reg**, **:registers**.)

(See page 312.)

:dif

:dif

When in diff mode, update the list of differences between the files.

(Same as **:diffupdate**.)

(See page 384.)

:diffg

: [range]diffg [buffer-spec]

:diffget

: [range]diffget [buffer-spec]

Get a different from the other buffer. In other words, take the change from the other buffer and put it in the current file.

(See page 384.)

:diffo

:diffo[!]

:diffoff

:diffoff[!]

Turn off diff mode for the current window. If the override (!) is specified, turn off diff mode for all windows in the current tab.

(See page 384.)

:diffp

:diffp {patchfile}

:diffpatch

:diffpatch {patchfile}

Apply the diffs in the **{patchfile}** to the current buffer.

The Vim Tutorial and Reference

(See page 384.)

:diffpu : [range]diffpu [buffer-spec]
:diffput : [range]diffput [buffer-spec]

Put a different from the current buffer to the other one.

(See page 384.)

:diffs :diffs {file-name}
:diffsplit :
 diffsplit {file-name}

Open a new window for the file {file-name}. Both windows will be part of a difference set.

(See page 383.)

:difft :difft
:diffthis :diffthis

Make this window part of the set of files being diffed.

(See page 384.)

:diffupdate :diffupdate
When in diff mode, update the list of differences between the files.
(Same as :dif.)

(See page 384.)

:dig :dig
List all the digraph definitions. (Same as: :digraphs.)
(See page 57.)

:dig :dig {character1}{character2} {number}
Define a digraph. When is pressed, inset character whose number **CTRL-K{character1}{character2}** is {number}.
(See page 282.)

:digraphs :digraphs
List all the digraph definitions. (Same as: :dig.)
(See page 57.)

:digraphs :digraphs {character1}{character2} {number}
Define a digraph. When **CTRL-K{character1}{character2}** is pressed, insert character whose number is {number}.
(See page 282.)

The Vim Tutorial and Reference

:display	:display [arg]
Display the contents of the registers. (Same as :registers, :di.)	
(See page 312.)	
:dj	:range dj [count] /{pattern}/
:djump	:range djump [count] /{pattern}/
Search the range (default = whole file) for the definition of the macro named {pattern} and jump to it. If a [count] is specified, jump to the [count] definition. If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is the full name of the macro.	
(See page 439.)	
:dl	:range dl /{pattern}/
:dlist	:range dlist /{pattern}/
List the all the definitions of the macro named {pattern} in the range. (Default = the whole file.) If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is the full name of the macro.	
(See page 440.)	
:do	:do {group} {event} [file_name]
Execute a set of autocommands pretending that {event} has just happened. If a group is specified, execute only the commands for that group. If a filename is given, pretend that the filename is file_name rather than the current file during the execution of this command. (Same as: :doautocmd.)	
(See page 204, 204, 206.)	
:doautoa	:doautoa {group} {event} [file_name]
:doautoall	:doautoall {group} {event} [file_name]
Like :doautocmd, but repeated for every buffer.	
(See page 204.)	
:doautocmd	:doautocmd {group} {event} [file_name]
Execute a set of autocommands, pretending that {event} has just happened. If a group is specified, execute only the commands for that group. If a [file_name] is given, pretend that the filename is file_name rather than the current file during the execution of this command. (Same as: :do.)	
(See page 204, 204, 206.)	

The Vim Tutorial and Reference

:dr :dr {file} [file] ...
:drop :drop {file} [file] ...

Edit the first file in the list. If the file is already in a window, then switch to that window. If not attempt to open the file in the current window. If switching files would cause changes in the current window to be lost, split the current window and then edit the file.
(See page 319.)

:ds : [range] ds /{pattern}/
:dsearch : [range] dsearch /{pattern}/

List the first definition of the macro named **{pattern}** in the range. (Default = the whole file.) If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is the full name of the macro.
(See page 440.)

:dsp : [range] dsp [count] /{pattern}/
:dsplit : [range] dsplit [count] /{pattern}/

Do a **:split** and a **:djump**.

(See page 440.)

:e :e [+cmd] [file]
Close the current file and start editing the named file. If no file is specified, re-edit the current file. If **[+cmd]** is specified, execute it as the first editing command. (Same as: **:edit**.)
(See page 77.)

:ea :ea {time}
:earlier :earlier {time}

Go to an earlier text state. If **{time}** is specified as a count, then count changes are undone. Time also can be specified as **{n}s**, then the state will go back that many seconds. Time can also be specified as **{n}m** for minutes, and **{n}h** for hours.

(See page 284.)

:ec :ec {arguments}
:echo :echo {arguments}

Print the arguments.

(See pages 487, 504.)

:echoe :echoe {arguments}
:echoerr :echoerr {arguments}

Print the arguments as an error message and save it in the message

The Vim Tutorial and Reference

history.

(See page 494, 498.)

:echoh

:echoh {name}

:echohl

:echohl {name}

Change the color of future echoes to be in the color of highlight group **{name}**.

(See page 494.)

:echom

:echom {arguments}

:echomsg

:echomsg {arguments}

Print the arguments as a message and save it in the message history.

(See page 495.)

:echon

:echon {arguments}

Echo the arguments without a newline.

(See page 494.)

:edit

:edit [+cmd] [file]

Close the current file and start editing the named file. If no **file** is specified, re-edit the current file. If **+cmd** is specified, execute it as the first editing command. (Same as: **:e**.)

(See page 77.)

:el

:el

:else

:else

Reverse the condition of an **:if**.

(See page 495.)

:elsei

:elsei

:elseif

:elseif

A combination of **:else** and **:if**.

(See page 495.)

:em

:em {menu-item}

:emenu

:emenu {menu-item}

Execute the given **{menu-item}** as if the user had selected it.

(See page 473.)

:en

:en

End an **:if** statement. (Same as: **:endif**.)

(See page 495.)

The Vim Tutorial and Reference

:endif	:endiff
End a function. (Same as :endfunction .) (See page 499.)	
:endfo	:endfo
:endfor	:endfor
End a :for loop. (See page 496.)	
:endfunction	:endfunction
End a function. (Same as :endif .) (See page 499.)	
:endif	:endif
End an :if statement. (Same as: :en .) (See page 495.)	
:endt	:endt
:endtry	:endtry
End a :try block. (See page 498.)	
:endw	:endw
:endwhile	:endwhile
End a :while loop. (See page 496.)	
:ene	:ene[!]
:enew	:enew[!]
Start editing a new buffer. (See page 77.)	
:ex	:ex[!] [+command] [filename]
Enter ex mode. If a filename is specified, edit that file; otherwise, use the current file. The +command argument is a single command that will be executed before any editing begins. If the override option (!) is specified, switching files will discard any changes that have been made. (See page Error: Reference source not found.)	
:ex	:[range] ex[!] [file]
If the buffer has been modified, write the file and exit. If a range is	

The Vim Tutorial and Reference

specified, write only the specified lines. If a **file** is specified, write the data to that file. When the override option (!) is present, attempt to overwrite existing files or read-only files.

(See page Error: Reference source not found.)

:exe	:exe {string}
:execute	:execute {string}

Execute a **string** as a command.

(See page 497.)

:exi	:[range] exi[!] [file]
:exit	:[range] exit[!] [file]

If the buffer has been modified, write the file and exit. If a **range** is specified, write only the specified lines. If a **file** is specified, write the data to that file. When the override option (!) is present, attempt to overwrite existing files or read-only files. (Same as **:xit**, **:x**.)

(See page 450.)

:exu	:exu
:exusage	:exusage

Provides help on the *ex* mode commands. Included for compatibility with *Nvi*. The **:help** command is much better for getting information than this one.

(See page 228.)

:f	:f[!] [file]
:file	:file[!] [file]

Print the current filename. The override (!) option causes the short version of the message to be printed.

If a file is specified, set the name of the current file to file. (Same as: **CTRL-G**.)

(See pages 204, 442.)

:files	
	List all the specified buffers. (Same as: :buffers , :ls .)
	(See page 90.)

:filet	:filet {on off}
:filetype	:filetype {on off}

Tell *Vim* to turn on or off the file type detection logic.

(See page 117.)

The Vim Tutorial and Reference

:fin	:fin[!] {+command} {file}
	Edit a file like the :vi command, but searches for the file in the directories specified by the path option. (Same as :find .) (See page 398.)
:fina	:fina
:finally	:finally
	This command starts the part of a try/catch block which is executed after all other code. (See page 498.)
:find	:find[!] {+command} {file}
	Edit a file like the :vi command, but searches for the file in the directories specified by the path option. (Same as :fin .) (See page 398.)
:fini	:fini
:finish	:finish
	Stop sourcing a script. (See page 498.)
:fir	:fir
:first	:first
	Edit the first file in the list. (Same as :rew , :rewind .) (See pages 80, 245, 317.)
:fix	:fix
:fixdel	:fixdel
	Make the <Delete> key do the right thing on UNIX systems. (See page 151.)
:fo	:{range}fo
:fold	:{range}fold
	Manually create a fold. (See page 389.)
:foldc	:{range}foldc[!]
:foldclose	:{range}foldclose[!]
	Close a fold. One level of folding is closed unless the override (!) option is present, then all levels are closed. (See page 389.)

The Vim Tutorial and Reference

:fold	:[range] fold [cmd]
	(Fold Do Open) Execute the {cmd} for each line in the [range] which is not folded. Much like :global only for open folds. (Same as :folddoopen .)
	(See page 390.)
:folddoc	:[range] folddoc [cmd]
:folddochlosed	:[range] folddochclosed [cmd]
	(Fold Do Closed) Execute the {cmd} for each line in the [range] which is folded. Much like :global only for closed folds.
	(See page 390.)
:folddoopen	:[range] folddoopen [cmd]
	(Fold Do Open) Execute the {cmd} for each line in the [range] which is not folded. Much like :global only for open folds. (Same as :fold .)
	(See page 390.)
:foldo	:[range] foldo[!]
:foldopen	:[range] foldopen[!]
	Open one level of folding for all lines within the given [range] . If the override option (!) is present, open all levels.
	(See page 389.)
:for	:for {var} in {list}
:for	:for {var1}, {var2} ... in {list}
	Start a loop. For scripting.
	(See page 496.)
:fu	:fu
	List all functions. (Same as: :function .)
	(See page 501.)
:fu	:fu {name}
	List the contents of function {name} . (Same as :function .)
	(See page 501.)
:fu	:fu {function definition}
	Start a function definition.
	(See page 499.)

The Vim Tutorial and Reference

:function	:function
List all functions. (Same as: :fu .)	
(See page 501.)	
:function	:function {name}
List the contents of function {name} .	
(See page 501.)	
:function	:function {function definition}
Start a function definition.	
(See page 499.)	
:g	: [range] g /{pattern}/ {command}
Perform {command} on all lines that have {pattern} in them in the given range. (Same as: :global .)	
(See page 438.)	
:g!	: [range] g! /{pattern}/ {command}
Perform {command} on all lines that do not have {pattern} in them in the given range. (Same as :global! , :v , :vglobal .)	
(See page 438.)	
:global	: [range] global /{pattern}/ {command}
Perform {command} on all lines that have {pattern} in them in the given range. (Same as: :g .)	
(See page 438.)	
:global!	: [range] global! /{pattern}/ {command}
Perform {command} on all lines that do not have {pattern} in them in the given range. (Same as :g! , :v , :vglobal .)	
(See page 438.)	
:go	:go [count]
:goto	:goto [count]
Go to [count] byte of the file. If no [count] is specified, go to the first byte of the file.	
(See page 226.)	
:gr	:gr {arguments}
:grep	:grep {arguments}
Run the <i>grep</i> program with the given {arguments} and capture the output so that the :cc , :cnext , and other commands will work on it. (Like :make , but with <i>grep</i> rather than <i>make</i> .)	

The Vim Tutorial and Reference

(See pages 403, 411, 406.)

:grep

:grepa {arguments}

:grepadd

:grepadd {arguments}

Like **:grep**, but add to the quick fix list instead of replacing it.

(See page 406.)

:gu

:gu [+command] [-f|-b] [files...]

:gui

:gui [+command] [-f|-b] [files...]

:gv

:gv [+command] [-f|-b] [files...]

:gvim

:gvim [+command] [-f|-b] [files...]

Start GUI mode. If a **+command** is specified, execute that after loading the files. If the **-b** flag is specified, execute the command in the background (the default). The **-f** flag tells *Vim* to run in the foreground. If a list of files is specified, they will be edited; otherwise, the current file is edited.

(See page 453.)

:h

:h [topic]

Display help on the given topic. If no topic is specified, display general help. (Same as: **:help**, **<F1>**, **<Help>**)

(See pages 37, 39, 41, 130, 131, 227.)

:ha

:[range]ha[!] [arguemets]

:hardcopy

:[range]hardcopy[!] [arguments]

Send the lines in **[range]** to the printer. The **[arguments]**, if specified will be given to the print command. The override (!) option will cause printing on Microsoft Windows to skip the printer selection dialog and go directly to the default printer.

(See page 166.)

:ha

:[range]ha[!] >{file-name}

:hardcopy

:[range]hardcopy[!] >{file-name}

Send the lines in **[range]** to the a printable (PostScript) file. The **[arguments]**, if specified will be given to the print command. This command does not work on Microsoft Windows, use the print to file feature in the printer dialog instead.

(See page 451.)

The Vim Tutorial and Reference

:help	:help [topic]
Display help on the given topic. If no topic is specified, display general help. (Same as: :h , <F1>, <Help>)	
(See pages 37, 39, 41, 110, 130, 131, 227.)	
:help!	:help!
:help 42	:help 42
:help holy-grail	:help hold-grail
These commands do something interesting. Not useful, but interesting.	
:helpf	:helpf
:helpfind	:helpfind
Open a dialog box that enables you to type in a help subject.	
(See page 478.)	
:helpg	:helpg {pattern} {@lang}
:helpgrep	:helpgrep {pattern} {@lang}
Search the help text for the given {pattern} and put the results in the quick fix list. If @lang is included, limit results to that language.	
(See page 228.)	
:helpt	:helpt {dir}
:helptags	:helptags {dir}
Generate the help tags. Useful for those of you rewriting the help file. But if you're rewriting the help files you probably don't need this book.	
(See page 228.)	
:hi	:hi
List all highlight groups. (Same as: :highlight .)	
(See page 81, 494.)	
:hi	:hi {options}
Customize the syntax coloration.	
(See page 412.)	
:hi link	:hi link {new-group} {old-group}
Highlight the {new-group} the same as {old-group} .	
(See page 587.)	

The Vim Tutorial and Reference

:hid	:hid
:hide	:hide
Hide the current buffer. (See page 90, 94.)	
:highlight	:highlight
List all highlight groups. (See page 81, 494.)	
:highlight	:highlight {options}
Customize the syntax coloration. (Same as: :hi .) (See pages 412.)	
:highlight link	:highlight link {new-group} {old-group}
Highlight the {new-group} the same as {old-group} . (See page 587.)	
:his	:his {code} [first] ,[last]
:history	:history {code} [first] ,[last]
Print the last few commands or search strings (depending on the code). The code parameter defaults to cmd for command-mode command history. The first parameter defaults to the first entry in the list and last defaults to the last. (See page 447.)	
:i	:[line] i
Start inserting text before line. Insert ends with a line consisting of just .. (Same as: :insert .) (See page 432.)	
:ia	:ia {lhs} {rhs}
:iabbrev	:iabbrev {lhs} {rhs}
Define an abbreviation for insert mode only. (See page 419.)	
:iabc	:iabc
:iabclear	:iabclear
Remove all for insert mode. (See page 419.)	

The Vim Tutorial and Reference

:if	:if {expression}
Start a conditional statement. (See page 495.)	
:ij	:[range] ij [count] /{pattern}/
:ijump	:[range] ijmp [count] /{pattern}/
Search the range (default = whole file) for the {pattern} and jump to it. If a [count] is specified, jump to the [count] occurrence. If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is just a string. (See page 438.)	
:il	:[range] il /{pattern}/
:ilist	:[range] ilist /{pattern}/
List all the occurrences {pattern} in the range. (Default = the whole file.) If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is a string. (See page 438.)	
:im	:im
List all the insert-mode mappings. (Same as: :imap .) (See page 421, 423, 425.)	
:im	:im {lhs}
List the insert-mode mapping of {lhs} . (Same as: :imap .) (See page 421, 425.)	
:im	:im {lhs} {rhs}
Define a keyboard mapping for insert mode. (Same as: :imap .) (See page 421, 425.)	
:imap	:imap
List all the insert-mode mappings. (Same as: :im .) (See page 421, 423, 425.)	
:imap	:imap {lhs}
List the insert-mode mapping of {lhs} . (Same as: :im .) (See page 421, 425.)	
:imap	:imap {lhs} {rhs}
Define a keyboard mapping for insert mode. (Same as: :im .) (See page 421, 425.)	

The Vim Tutorial and Reference

:imapc	:imapc
:imapclear	:imapclear
Clear all the insert-mode mappings. (See page 425.)	
:ime	
:imenu	
Define a menu item that is available for insert mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is {menu-item} , and when the command is selected, the command {command-string} is executed. (See page 468.)	
:ino	:ino {lhs} {rhs}
Same as :imap , but does not allow remapping of the {rhs} . (Same as: :inoremap .) (See page 425.)	
:inorea	:inorea {lhs} {rhs}
:noreabbrev	:noreabbrev {lhs} {rhs}
Do a :nreabbrev that works in insert mode only. (See page 419, 425.)	
:inoremap	:inoremap {lhs} {rhs}
Same as :imap , but does not allow remapping of the {rhs} . (Same as: :ino .) (See page 425.)	
:inoreme	:inoreme {menu-item} {command-string}
:inoremenu	:inoremenu {menu-item} {command-string}
Like :imenu , except the {command-string} is not remapped. (See page 473.)	
:insert	:[line] insert
Start inserting text before line. Insert ends with a line consisting of just .. (Same as: :i .) (See page 432.)	
:int	:int
:intro	:intro
Display the introductory screen. (See page 230.)	

The Vim Tutorial and Reference

:is *: [range] is /{pattern}/*
:isearch *: [range] isearch /{pattern}/*

List the first occurrence *{pattern}* in the range. (Default = the whole file.) If the pattern is enclosed in slashes (/), it is a regular expression; otherwise, it is a string.

(See page 439.)

:isp *: [range] isp [count] /{pattern}/*
:isplit *: [range] isplit [count] /{pattern}/*

Combination of **:split** and **:ijump**. Split the window and jump to the first occurrence of the *{pattern}*. Things that look like comments are ignored.

(See page 439.)

:iu *: iu {lhs}*
Remove an insert-mode mapping. (Same as: **:iunmap**.)
(See page 425.)

:iuna *: iuna {lhs}*
:iunabbreviate *: iunabbreviate {lhs}*
Remove the insert line-mode abbreviation.
(See page 419, 425.)

:iunmap *: iunmap {lhs}*
Remove an insert-mode mapping. (Same as **:iu**.)
(See page 425.)

:iunme *: iunme {menu-item}*
:iunmenu *: iunmenu {menu-item}*
Remove the insert-mode menu item named *{menu-item}*. The wildcard * will match all menu items.
(See page 473.)

:j *: [range] j[!]*
:join *: [range] join[!]*
Join the lines in range into one line. Spaces are used to separate the parts unless the ! is specified.

(See page 445.)

:ju *: ju*
:jumps *: jumps*
List out the jump list.

The Vim Tutorial and Reference

(See page 266.)

:k

:*[line]* k*{letter}*

Place mark *{letter}* on the indicated line. (Same as: :mar, :mark.)

(See page 446.)

:ke

:ke *{command}*

Execute *{command}* and attempt to preserve the marks inside the text affected by the command. Currently only filter (*[range]!*) commands are supported. The number of lines after filtering must be greater or equal to the number of lines before for this to work.
(Same as :keepmarks.)

(See page 311.)

:keepa

:keepa *{command}*

:keepalt

:keepalt *{command}*

Execute *{command}* but do not change the name of the alternate file.
(Even if the *{command}* would normally cause it to change.)

(See page 228.)

:keepj

:keepj *{command}*

:keepjumps

:keepjump *{command}*

Execute a command which might normally move or destroy the jump list or the marks ' ', '.', or '^', but do not destroy the marks.

(See page 228.)

:keepmarks

:keepmarks *{command}*

Execute *{command}* and attempt to preserve the marks inside the text affected by the command. Currently only filter (*[range]!*) commands are supported. The number of lines after filtering must be greater or equal to the number of lines before for this to work.
(Same as :ke.)

(See page 311.)

:l

:*[range]* l *[count]*

Like :print, but assumes that the 'list' option is on.

(See page 433.)

:la

:la *[+command]*

Edit the last file in the list. (Same as :last.)

(See page 80, 317.)

The Vim Tutorial and Reference

:lad	:lad[!] {expr}
	Add the results of {expr} to the location list. (Same as :ladexpr .) (See page 405.)
:laddir	:laddir {buffer}
:laddirbuffer	:laddirbuffer {buffer}
	Add the contents of {buffer} to the location list. (See page 405.)
:ladexpr	:ladexpr[!] {expr}
	Add the results of {expr} to the location list. (Same as :lad .) (See page 405.)
:laddir	:laddir {file}
:laddirfile	:laddirfile {file}
	Add the contents of {file} to the location list. (See page 405.)
:last	:last [+command]
	Edit the last file in the list. (Same as :la .) (See page 80, 317.)
:lat	:lat {time}
:later	:later {time}
	Go to an later text state. If {time} is specified as a count, then count changes are redone. Time also can be specified as {n}s , then the state will go forward that many seconds. Time can also be specified as {n}m for minutes, and {n}h for hours. (See page 284.)
:lb	:lb[!] [buffer-number]
:lbuffer	:lbuffer[!] [buffer-number]
	Read the contents of the buffer into the location list and jump to the first location. (See page 405.)
:lc	:lc[!] [path]
:lcd	:lcd[!] [path]
:lch	:lch[!] [path]
:lchdir	:lchdir[!] [path]
	Change the directory like :cd , but only for the current window.

The Vim Tutorial and Reference

(See page 442.)

:lcl :lcl
:lclose :lclose

Close the window with the location list in it.

(See page 406.)

:lcs :lcs {arguments}
:lcscope :lcscope {argument}

Handle various activities associated with the *CScope* program sending the results to the location list.

(See page 247.)

:le :[range] le [margin]
:left :[range] left [margin]

Left justify the text putting each line **[margin]** characters from the left margin. (Default = 0.)

(See page 178.)

:lefta :lefta {cmd}
:leftabove :leftabove {cmd}

Execute the *Vim* command **{cmd}**. If the command splits a window, the window is opened above the current one or to its left (overriding any any split options that are currently set.) (Same as **:abo**, **:aboveleft**)

(See page 346.)

:let :let {variable} = {expression}
Assign a **{variable}** a value.
(See page 487.)

:lex :lex[!] {expr}
:lexpr :lexpr[!] {expr}

Evaluate **{expr}** and use it to create a location list.

(See page 405.)

:lf :lf[!] {file}
:lfile :lfile[!] {file}

Create a location list from the contents of **{file}**.

(See page 405.)

The Vim Tutorial and Reference

:lfir	:lfir[!] [number]
:lfirst	:lfirst[!] [number]
Go to the first location in the location list. If a number is specified go to that error number. (Same as :lr , :lrewind .)	
(See page 405.)	
:lg	:lg[!] {file}
Read the file into the location list but do not jump to the first location. (Same as :lgetfile .)	
(See page 405.)	
:lgetb	:lgetb {buffer}
:lgetbuffer	:lgetbuffer {buffer}
Read the {buffer} into the location list, but do not jump to the first location.	
(See page 405.)	
:lgete	:lgete[!] {expr}
:lgetexpr	:lgetexpr[!] {expr}
Evaluate the expression and turn it into a location list. Do not jump to the first location.	
(See page 405.)	
:lgetfile	:lgetfile[!] {file}
Read the file into the location list but do not jump to the first location. (Same as :lg .)	
(See page 405.)	
:lgr	:lgr[!] [arguemets]
:lgrep	:lgrep[!] [arguemets]
Execute a <i>grep</i> command and use the results to create a location list. (See :grep .)	
(See page 406.)	
:lgrepa	:lgrepa[!] [arguemets]
:lgrepadd	:lgrepadd[!] [arguemets]
Execute a <i>grep</i> command and add the results to the location list. (See :grep .)	
(See page 406.)	

The Vim Tutorial and Reference

:lh	:lh {pattern} {@lang}
:lhelpgrep	:lhelpgrep {pattern} {@lang}
	Search the help text for the given {pattern} and put the results in the location list. If @lang is included, limit results to that language.
	(See page 228.)
:list	:[range] list [count]
	Like :print , but assumes that the ' list ' option is on. (Same as: :1.)
	(See page 433.)
:ll	:ll[!] [number]
	Move to the current location in the location list. If a number is specified, move to the indicated entry in the location list.
	(See page 404.)
:lla	:lla[!] [number]
:llast	:llast[!] [number]
	Move to the last location in the location list (or a [number] of entries from the end if specified.)
	(See page 405.)
:lli	:lli[!] [from] [,to]
:llist	:llist[!] [from] [,to]
	List all the locations in the list that match the given range (the default being all). If the override (!) option is given, output is limited to those locations valid for the current window.
	(See page 406.)
:lm	:lm {lhs} {rhs}
	Define a language dependent mapping that's will be used in command mode, insert mode, and when entering a language dependent argument. (Same as :lmap .)
	(See page 425.)
:lmak	:lmak[!] [arguements]
:lmake	:lmake[!] [arguments]
	Execute the <i>make</i> command and capture the results in the location list. Position the editor on the first error.
	(See page 406.)

The Vim Tutorial and Reference

:lmap	:lmap {lhs} {rhs}
	Define a language dependent mapping that's will be used in command mode, insert mode, and when entering a language dependent argument. (Same as :lm.)
	(See page 425.)
:lmapc	:lmapc
:lmapclear	:lmapclear
	Clear all language dependent mappings.
	(See page 425.)
:ln	:[count]ln [!]
	Go to the previous location in the location list. (Same as :lNext , :lp , :lprevious.)
	(See page 405.)
:ln	:ln {lhs} {rhs}
	Create a language dependent mapping which is used for command mode, insert mode, and language input which does not remap {rhs} . (Same as :lnoremap .)
	(See page 425.)
:lne	:[count]lne[!]
	Go to the next location in the location list. (Same as :lnext.)
	(See page 405.)
:lnew	:lnew [count]
:lnewer	:lnewer [count]
	Go to a newer version of the location list.
	(See page 406.)
:lNext	:[count]lNext [!]
	Go to the previous location in the location list. (Same as :ln , :lp , :lprevious.)
	(See page 405.)
:lnext	:[count]lnext[!]
	Go to the next location in the location list. (Same as :lne.)
	(See page 405.)

The Vim Tutorial and Reference

:lnf	:lnf [count]!
:lnfile	:lnfile [count]!
	Go to the last location in the previous file. (Same as :lpf , :lpfile .) (See page 405.)
:lnf	:lnf [count]!
:lnfile	:lnfile [count]!
	Go to the first location in the next file in the location list. (See page 405.)
:lnoremap	:lnoremap {lhs} {rhs}
	Create a language dependent mapping which is used for command mode, insert mode, and language input which does not remap {rhs} . (Same as :ln .) (See page 425.)
:lo	:lo [number]
	Load a view which has been made with :mkview for the current file. (Same as :loadview .) (See page 350.)
:loadk	:loadk {file}
:loadkeymap	:loadkeymap {file}
	Load a keymap file. (See page 561.)
:loadview	:loadview [number]
	Load a view which has been made with :mkview for the current file. (Same as :lo .) (See page 350.)
:loc	:loc {cmd}
:lockmarks	:lockmarks {cmd}
	Execute {cmd} without changing the location of any marks. (See page 311.)
:lockv	:lockv[!] [depth] {name}
:lockvar	:lockvar[!] [depth] {name}
	Lock a variable so it can not be changed. The [depth] if specified is a code which indicates how to lock dictionaries and list. A value of 1 locks the size of the dictionary or list, but lets you change values.

The Vim Tutorial and Reference

The 2 code indicates that the top level values can not be changed. A level of 3 indicates that the array, the values in the array, and their values can not be changed. The override option (!) tell *Vim* that nothing can be changed.

(See page 491.)

:lol	:lol [count]
:lolder	:lolder [count]

Use an older version of the location list.

(See page 406.)

:lop	:lop [height]
:lopen	:lopen [height]

Open a window containing the location list.

(See page 406.)

:lp	: [count]lp [!]
------------	------------------------

Go to the previous location in the location list. (Same as :**ln**,
:lNext, **:lprevious**.)

(See page 405.)

:lpf	: [count]lpf[!]
:lpfile	: [count]lpfile[!]

Go to the last location in the previous file. (Same as :**lnf**, **:lnfile**.)

(See page 405.)

:lprevious	: [count]lprevious [!]
-------------------	-------------------------------

Go to the previous location in the location list. (Same as :**ln**,
:lNext.)

(See page 405.)

:lr	:lr[!] [number]
:lrewind	:lrewind[!] [number]

Go to the first location in the location list. If a number is specified go to that error number. (Same as :**lfir**, **:lfirst**.)

(See page 405.)

:ls	:ls
------------	------------

List all the buffers. (Same as: **:buffers**, **:files**.)

(See page 90.)

The Vim Tutorial and Reference

:lt	:lt[!] {name}
:ltag	:ltag[!] {name}
Jump to the first tag {name} and create a location list for this window containing all matching tags.	
(See page 406.)	
:lu	:lu {lhs}
:lunmap	:lunmap {lhs}
Remove a language entry mode mapping from the system.	
(See page 425.)	
:lv	:lv[!] /{pattern}/[g][j] {file-list}
:lvimgrep	:lvimgrep[!] /{pattern}/[g][j] {file-list}
Perform a search using the <i>Vim</i> internal <i>grep</i> command and store the results in a new location list for the current window.	
(See page 406.)	
:lvimgrepa	:lvimgrepa[!] /{pattern}/[g][j] {file-list}
:lvimgrepadd	:lvimgrepadd[!] /{pattern}/[g][j] {file-list}
Perform a search using the <i>Vim</i> internal <i>grep</i> command and add the results to the existing location list for the current window.	
(See page 406.)	
:lw	:lw [height]
:lwindow	:lwindow [height]
Open the location window if it contains data. If it is open and empty, close it.	
(See page 406.)	
:m	: [range] m {address}
Move the range of lines from their current location to below {address} . (Same as: :move .)	
(See pages 234, 431.)	
:ma	: [line] ma{letter}
Mark the current line with mark {letter} . (Same as: :k , :mark .)	
(See page 446.)	
:mak	:mak {arguments}
:make	:make {arguments}
Run the external <i>make</i> program, giving it the arguments indicated.	

The Vim Tutorial and Reference

Capture the output in a file so that error-finding commands such as **:cc** and **:cnext** can be used.

(See pages 145, 403, 406.)

:map

:map[!]

List all the mappings. Note: Only **:map** and **:map!** list the mappings for all modes. The other mode-dependent versions of these commands list the mapping for their modes only. Normally this command lists the mappings for the normal, visual, select, and operating pending modes. With the override (!) it lists the mappings for the Insert, Command Line, and Language argument modes.

(See pages 150, 421, 423, 425.)

:map

:map {lhs}

List the mapping of **{lhs}**.

(See page 421, 423, 425.)

:map!

:map! {lhs}

List the mapping of **{lhs}** for insert and command mode..

(See page 421, 425.)

:map

:map {lhs} {rhs}

Define a keyboard mapping. When the **{lhs}** is typed in normal mode, pretend that **{rhs}** was typed.

(See page 149, 420, 421, 425, 539.)

:map!

:map! {lhs} {rhs}

Define a keyboard mapping. When the **{lhs}** is typed in insert or command mode, pretend that **{rhs}** was typed.

(See page 149, 420, 421, 425.)

:mapc

:{mode} mapc[!]

:mapclear

:{mode} mapclear[!]

Clear all the mappings. Normally this command clears the mappings for the normal, visual, select, and operating pending modes. With the override (!) it clears the mappings for the Insert, Command Line, and Language argument modes.

(See page 423, 425.)

:mark

:{line} mark {letter}

Mark the given line with mark **{letter}**. (Same as: **:k**.)

(See page 446.)

The Vim Tutorial and Reference

:marks	List all the marks. (See page 73, 74.)	:marks
:marks	List the marks specified by the character list: {chars} . (See page 73, 74.)	:marks {chars}
:mat		:[number]mat {group} /{pattern}/
:match		:[number]match {group} /{pattern}/
	Display text that matches {pattern} using the given highlight {group} . Multiple matches may be displayed at the same time by using the numbers 1-3. (See page 81.)	
:me		:[priority][mode] me {menu-item} {command-string}
:menu		:[priority][mode] menu {menu-item} {command-string}
	Define a menu item. The priority determines its placement in a menu. Higher numbers come first. The mode parameter defines which <i>Vim</i> mode the item works in. The name of the menu item is {menu-item} , and when the command is selected, the command {command-string} is executed. (See page 467.)	
:menut		:menut {english} {lang}
:menut		:menut clear
:menutranslate		:menutranslate {english} {lang}
:menutranslate		:menutranslate clear
	Create a menu translation from English to another language. If the keyword clear is use, clear all translations. (See page 474.)	
:mes		:mes
:messages		:messages
	View previous messages. (See page 449, 495.)	
:mk		:mk[!] [file]
:mkexrc		:mkexrc[!] [file]
	Like :mkvimrc , except the [file] defaults to <i>.exrc</i> . This command has been superseded by the :mkvimrc command. (Same as: :mk .)	

The Vim Tutorial and Reference

(See page 155.)

:mks

:mks[!] {file}

:mkssession

:mkssession[!] {file}

Create a session file and save the current settings. If the override option (!) is specified, overwrite any existing session file.

(See page 348, 488.)

:mksp

:mksp[!] [-ascii] {outname} [inname]

:mkspell

:mkspell[!] [-ascii] {outname} [inname]

Create a spelling word list file.

(See page 332.)

:mkv

:mkv[!] {file}

Write out setting to **{file}** in a manner suitable for including in a *.vimrc* file. In fact, if you do not specify **{file}**, it defaults to *.vimrc*. If the file exists, it will be overwritten if the override option (!) is used. (Same as **:mkvimrc**.)

(See page 152.)

:mkvie

:mkvie[!] {file}

:mkview

:mkview[!] {file}

Create a view file which stores the view information about the current window. See **:loadview**.

(See page 350.)

:mkvimrc

:mkvimrc[!] {file}

Write out setting to **{file}** in a manner suitable for including in a *.vimrc* file. In fact, if you do not specify **{file}**, it defaults to *.vimrc*. If the file exists, it will be overwritten if the override option (!) is used. (Same as **:mkv**.)

(See page 152.)

:mod

:mod {mode}

:mode

:mode {mode}

Set the screen mode for an MS-DOS editing session.

(See page 486.)

:move

:{range} move {address}

Move the range of lines from their current location to below **{address}**. (Same as: **:m**.)

(See pages 234, 431.)

The Vim Tutorial and Reference

:mz	:[range]mz {statement}
	Execute the <i>MzScheme</i> statement {statement} . (Same as : mzscheme .)
	(See page 248.)
:mz	:[range]mz << {marker}
	Execute the <i>MzScheme</i> statements that follow up to {marker} . (Same as : mzscheme .)
	(See page 248.)
:mzf	:[range]mzf {file}
:mzfile	:[range]mzfile {file}
	Execute the <i>MzScheme</i> statements in {file} .
	(See page 248.)
:mzscheme	:[range]mzscheme {statement}
	Execute the <i>MzScheme</i> statement {statement} . (Same as : mz .)
	(See page 248.)
:mzscheme	:[range]mzscheme << {marker}
	Execute the <i>MzScheme</i> statements that follow up to {marker} . (Same as : mz .)
	(See page 248.)
:n	:[count] n [+cmd] {file-list}
	When editing multiple files, go to the next one. If [count] is specified, go to the [count] next file. (If no {file-list} , same as: : next . If {file-list} same as: : args , : ar , : next .)
	(See pages 78, 245, 246, 317, 319.)
:N	:[count] N [+cmd] {file-list}
	When editing multiple files, go to the previous one. If a [count] is specified, go to the [count] previous file. (Same as: : Next , : prev , : previous .)
	(See page 80, 317.)
:nb	:nb {key}
:nbkey	:nbkey {key}
	Used to send ean:q the given key for when <i>Vim</i> is working inside of <i>Netbeans</i> .
	(See page 249.)

The Vim Tutorial and Reference

:new	:new[!] [+command] [file-name]
	Split the window like :split . The only difference is that if no filename is specified, a new window is started on a blank file. (Same as: CTRL-W CTRL-N, CTRL-Wn.)
	(See page 88.)
:next	:[count] next [+cmd] [file-list]
	When editing multiple files, go to the next one. If count is specified, go to the count next file. (If no {file-list} , same as: :n . If {file-list} same as: :args, :ar, :n .)
	(See pages 78, 245, 246, 317, 319.)
:Next	:[count] Next [+cmd] [file-list]
	When editing multiple files, go to the previous one. If [count] is specified, go to the [count] previous file. (Same as: :N, :prev, :previous .)
	(See page 80, 317.)
:nm	:nm
	Listing all the mappings for normal-mode maps. (Same as: :nmap .)
	(See page 421, 423, 425.)
:nm	:nm {lhs}
	List the normal mapping of {lhs} . (Same as :nmap .)
	(See page 421, 425.)
:nm	:nm {lhs} {rhs}
	Define a keyboard mapping for normal mode. (Same as :nmap .)
	(See page 421, 425.)
:nmap	:nmap
	Listing all the normal-mode mappings. (Same as: :nm .)
	(See page 421, 423, 425.)
:nmap	:nmap {lhs}
	List the normal-mode mapping of {lhs} . (Same as :nm .)
	(See page 421, 425.)
:nmap	:nmap {lhs} {rhs}
	Define a keyboard mapping for normal mode. (Same as :nm .)
	(See page 421, 425.)

The Vim Tutorial and Reference

:nmapc	:nmapc
:nmapclear	:nmapclear
Clear all the normal mappings.	
(See page 425.)	
:nme	:[priority]nme {menu-item} {command-string}
:nmenu	:[priority]nmenu {menu-item} {command-string}
Define a menu item that is available for normal mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is {menu-item} , and when the command is selected, the command {command-string} is executed.	
(See page 468.)	
:nn	:nn {lhs} {rhs}
:nnoremap	:nnoremap {lhs} {rhs}
Same as :nmap , but does not allow remapping of the {rhs} .	
(See page 425.)	
:nnoreme	:[priority] nnoreme {menu-item} {command-string}
:nnoremenu	:[priority] nnoremenu {menu-item} {command-string}
Like :nmenu , but the {command-string} is not remapped.	
(See page 473.)	
:no	:no {lhs} {rhs}
Same as :map , but does not allow remapping of the {rhs} . (Same as: :noremap .)	
(See pages 419, 421, 425.)	
:noh	:noh
:nohlsearch	:nohlsearch
Turn off the search highlighting. (It will be turned on by the next search. To turn it off permanently, use the :set nohlsearch command.)	
(See page 326.)	
:norea	:norea {lhs} {rhs}
:noreabbrev	:noreabbrev {lhs} {rhs}
Define an abbreviation, but do not allow remapping of the right side.	
(See page 424, 425.)	

The Vim Tutorial and Reference

:noremap	:noremap {lhs} {rhs}
	Same as :map , but does not allow remapping of the {rhs} . (Same as: :no .)
	(See pages 419, 425.)
:norem	:[priority][mode] norem{!} {menu-item} {command-string}
:noremenu	:[priority][mode] noremenu {!} {menu-item} {command-string}
	Define a menu item like defined with :menu , but do not allow remapping of the {command-string} .
	(See page 473.)
:norm	:norm{!} {commands}
:normal	:normal{!} {commands}
	Execute the commands in normal mode. If the override option (!) is present, mappings will not be used.
	(See page 450.)
:nu	:[range] nu
:number	:[range] number
	Print the lines with line numbers.
	(See page 433.)
:nun	:nun {lhs}
:nunmap	:nunmap {lhs}
	Remove a normal mapping.
	(See page 425.)
:nunme	:nunme {menu-item}
:nunmenu	:nunmenu {menu-item}
	Remove the normal menu item named {menu-item} . The wildcard * will match all menu items. (Same as: :nunme .)
	(See page 468.)
:o	:o
	The one command that <i>Vi</i> has that <i>Vim</i> does not. (In <i>Vi</i> , this command puts the editor into "open" mode, a mode that no sane persons ever use if they can avoid it.) (Same as: :open .)
	(See page 231.)
:om	:om
	List all the mappings for operator-pending-mode maps. (Same as:

The Vim Tutorial and Reference

:omap.)

(See page 421, 423, 425.)

:om

:om {lhs}

List the operator-pending mapping of **{lhs}**. (Same as **:omap.**)

(See page 421, 425.)

:om

:om {lhs} {rhs}

Define a keyboard mapping for operator-pending mode. (Same as

:omap.)

(See page 421, 425.)

:omap

:omap

List all the operator-pending-mode mappings. (Same as: **:om.**)

(See page 421, 423, 425.)

:omap

:omap {lhs}

List the operator-pending-mode mapping of **{lhs}**. (Same as **:om.**)

(See page 421, 425.)

:omap

:omap {lhs} {rhs}

Define a keyboard mapping for operator-pending mode. (Same as

:om.)

(See page 421, 425.)

:omapc

:omapc

:omapclear

:omapclear

Clear all the operator-pending-mode mappings.

(See page 425.)

:ome

:ome [priority] ome {menu-item} {command-string}

:omenu

:omenu [priority] omenu {menu-item} {command-string}

Define a menu item that is available for operator-pending mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is **{menu-item}**, and when the command is selected, the command **{command-string}** is executed.

(See page 468.)

:on

:on[!]

:only

:only[!]

Make the current window the only one. If ! is specified, modified files

The Vim Tutorial and Reference

whose windows are closed will have their contents discarded. (Same as: **:CTRL-W CTRL-O, CTRL-Wo.**)

(See page 339.)

:ono

:ono {lhs} {rhs}

:onoremap

:onoremap {lhs} {rhs}

Same as **:omap**, but does not allow remapping of the **{rhs}**.

(See page 425.)

:onoreme

:[priority] onoreme {menu-item} {command-string}

:onoremenu

:[priority] onoremenu {menu-item} {command-string}

Like **:omenu**, but the the **{command-string}** is not remapped.

(See page 473.)

:open

:open

The one command that *Vi* has that *Vim* does not. (In *Vi*, this command puts the editor into "open" mode, a mode that no sane persons ever use if they can avoid it.) (Same as: **:o**. See page 231.)

:opt

:opt

:options

:options

Enter an option-browsing window that enables you to view and set all the options. (Same as: **:bro set, :browse set.**)

(See page 479.)

:ou

:ou {lhs}

:ounmap

:ounmap {lhs}

Remove an operator-pending-mode mapping.

(See page 422, 425.)

:ounme

:ounme {menu-item}

:ounmenu

:ounmenu {menu-item}

Remove the command-mode menu item named **{menu-item}**. The wildcard * will match all menu items.

(See page 473.)

:p

: [range] p

:p

: [range] P

Print the specified lines. (Same as: **:print, :Print.**)

(See page 160, 160, 161.)

The Vim Tutorial and Reference

:pc	:pc[!]
:pclose	:pclose[!]
Close the preview window. Discard any changes if the force (!) option is present. (Same as: CTRL-W CTRL-Z , CTRL-Wz .)	
(See page 393.)	
:pe	:pe {command}
Execute a single Perl command. Requires <i>Vim</i> be compiled with Perl support (not on by default). (Same as :perl .)	
(See page 249.)	
:pe	:pe << pattern
Execute Perl commands until a line containing only <i>pattern</i> is seen.. Requires <i>Vim</i> be compiled with Perl support (not on by default).	
(Same as :perl .)	
(See page 249.)	
:ped	:ped[!] [++opt] [+cmd] {file}
:pedit	:pedit[!] [++opt] [+cmd] {file}
Edit a file in the preview window.	
(See page 393.)	
:perl	:perl {command}
Execute a single Perl command. Requires <i>Vim</i> be compiled with Perl support (not on by default). (Same as :pe .)	
(See page 249.)	
:perl	:perl << pattern
Execute Perl commands until a line containing only <i>pattern</i> is seen.. Requires <i>Vim</i> be compiled with Perl support (not on by default).	
(Same as :pe .)	
(See page 249.)	
:perld	: [range] perld {command}
:perldo	: [range] perldo {command}
Execute a Perl command on a range of lines. The Perl variable <code>\$_</code> is set to each line in range.	
(See page 249.)	
:po	: [count] po[!]
:pop	: [count] pop[!]
Go back [count] tags. If the current buffer has been modified, this	

The Vim Tutorial and Reference

command will fail unless the force (!) option is present.

(See page 393.)

:pp

: [count]pp[!]

:ppop

: [count] ppop[!]

Do a **:pop** command in the preview window. If the force option (!) is specified, discard any changes made on the file in the preview window. If a **[count]** is specified, pop that many tags.

(See page 393.)

:pre

:pre

:preserve

:preserve

Write out entire file to the swap file. This means that you can recover the edit session from just the swap file alone.

(See pages 224, 446.)

:prev

: [count] prev {[+cmd]} {[file-list]}

:previous

: [count] previous {[+cmd]} {[file-list]}

Edit the previous file in the file list. (Same as **:N**, **:Next**.)

(See page 80, 317.)

:print

: [range]print

:Print

: [range]Print

Print the specified lines. (Same as **:p**.)

(See pages 160, 160, 161.)

:pro

:pro

Open a Find dialog box. (Same as **:promptfind**.)

(See page 477.)

:prof

:prof continue

Continue profiling after a profiling pause. (Same as **:profile**.)

(See page 508.)

:prof

:prof[!] file {pattern}

Profile all files which match **{pattern}**. If the override (!) operator is used, profile the functions inside these files as well. (Same as **:profile**.)

(See page 508.)

:prof

:prof func {pattern}

Profile all functions that match **{pattern}**. (Same as **:profile**.)

The Vim Tutorial and Reference

(See page 506.)

:prof	:prof pause
	Stop profiling until a :profile continue command is executed. (Same as :profile .)
	(See page 508.)
:prof	:prof start {file}
	Start profiling. Write the results to {file} when done. (Same as :profile .)
	(See page 506.)
:prof	:prof *
:prof	:prof {number}
:prof	:prof func {line} {function-name}
:prof	:prof file {line} {file-name}
:prof	:prof here
:prof	:prof del *
:prof	:prof del {number}
:prof	:prof del func {line} {function-name}
:prof	:prof del file {line} {file-name}
:prof	:prof del here
	Stop profiling the indicated item. The item specification can be * (star) which deletes all profiles, or a profile {number} . A specific line in a function or file can be used. Finally the profiling of the current location can be turned off by using the here keyword.
	(See page 508.)
:profile	:profile continue
	Continue profiling after a profiling pause. (Same as :prof .)
	(See page 508.)
:profile	:profile[!] file {pattern}
	Profile all files which match {pattern} . If the override (!) operator is used, profile the functions inside these files as well. (Same as :prof .)
	(See page 508.)
:profile	:profile func {pattern}
	Profile all functions that match {pattern} . (Same as :prof .)
	(See page 506.)

The Vim Tutorial and Reference

:profile	:profile pause
Stop profiling until a :profile continue command is executed. (Same as :prof.)	
(See page 508.)	
:profile	:profile start {file}
Start profiling. Write the results to {file} when done. (Same as :prof.)	
(See page 506.)	
:promptfind	:promptfind
Open a Find dialog box. (Same as :pro.)	
(See page 477.)	
:promptr	:promptr
:promptrepl	:promptrepl
Open a Replace dialog box.	
(See page 477.)	
:ps	: [range]ps[!] [count] /{pattern}/
:psearch	: [range]psearch[!] [count] /{pattern}/
Search for the given pattern and show the first match in the preview window. If the override option (!) is present, things that look like comments are searched. If the pattern is enclosed in slashes (/) then it's a regular expression. Without the slashes (/) it's considered a set of independent words.	
:pt	:pt[!] {identifier}
:ptag	:ptag[!] {identifier}
Open a preview window and do a :tag . Discard any changes in the preview window if the override (!) option is present.	
(See page 392, 393.)	
:ptf	: [count] ptf[!]
:ptfirst	: [count] ptfirst[!]
Do a :trewind in the preview window. Discard any changes in the preview window if the override (!) option is present. (Same as :ptr , :ptrewind .)	
(See page 393.)	

The Vim Tutorial and Reference

:ptj	:ptj[!] {identifier}
:ptjump	:ptjump[!] {identifier}
Open a preview window and do a :tjump . Discard any changes in the preview window if the override (!) option is present.	
(See page 393.)	
:ptl	:ptl[!]
:ptlast	:ptlast[!]
Do a :tlast in the preview window. Discard any changes in the preview window if the override (!) option is present.	
(See page 393.)	
:ptn	: [count] ptn[!]
Open a preview window and do a : [count] tnext . Discard any changes in the preview window if the override (!) option is present. (Same as: :ptnext .)	
(See page 393.)	
:ptN	: [count] ptN[!]
Open a preview window and do a : [count] tprevious . Discard any changes in the preview window if the override (!) option is present. (Same as: :ptNext , :ptp , :ptprevious .)	
(See page 393.)	
:ptnext	: [count] ptnext[!]
Open a preview window and do a : [count] tnext[!] . Discard any changes in the preview window if the override (!) option is present. (Same as: :ptn .) (See page 393.)	
:ptNext	: [count] ptNext[!]
Same as : [count] ptnext! . (Same as: :ptNext , :ptprevious .)	
(See page 393.)	
:ptp	: [count] ptp[!]
:ptprevious	: [count] ptprevious[!]
Do a :tprevious in the preview window. Discard any changes in the preview window if the override (!) option is present. (Same as: :ptN , :ptNext .)	
(See page 393.)	

The Vim Tutorial and Reference

:ptr	: [count] ptr[!]
:ptrewind	: [count] ptrewind[!]
Do a :trewind in the preview window. Discard any changes in the preview window if the override (!) option is present. (Same as :ptf , :ptfirst .)	
(See page 393.)	
:pts	:pts[!] {identifier}
:ptselect	:ptselect[!] {identifier}
Open a preview window and do a :tselect . Discard any changes in the preview window if the override (!) option is present.	
(See page 393.)	
:pu	: [line] pu[!] register
:put	: [line] put[!] register
Put the text in the register after (before ! is specified) the specified line. If a register is not specified, it defaults to the unnamed register.	
(See page 445.)	
:pw	:pw
:pwd	:pwd
Print current working directory.	
(See page 441.)	
:py	: [range] py {statement}
Execute a single Python {statement} . (Same as: :python .)	
(See page 250.)	
:pyf	: [range] pyf {file}
:pyfile	: [range] pyfile {file}
Executes the Python program contained in {file} .	
(See page 250.)	
:python	: [range] python {statement}
Execute a single Python {statement} . This works only if Python support was compiled into <i>Vim</i> ; it does not work by default. (Same as: :py .)	
(See page 250.)	
:q	:q[!]
Close a window. If this is the last window, exit <i>Vim</i> . The command fails if this is the last window for a modified file, unless the force (!)	

The Vim Tutorial and Reference

option is present. (Same as: **CTRL-W CTRL-Q**, **CTRL-Wq**, **:quit.**)
(See pages 35, 84, 165, 288, 338, 96.)

:qa :qa[!]
:qall :qall[!]

Close all windows. If the force option is present, any modifications that have not been saved will be discarded. (Same is **:quita**, **:quitall**.)

(See page 338.)

:quit :quit[!]

Close a window. If this is the last window, exit *Vim*. The command fails if this is the last window for a modified file, unless the force (!) option is present. (Same as: **CTRL-W CTRL-Q**, **CTRL-Wq**, **:q.**)

(See pages 35, 84, 165, 288, 338, 9696.)

:quita :quita[!]
:quitall :quitall[!]

Close all windows. If the force option is present, any modifications that have not been saved will be discarded. (Same is **:qa**, **:qall**.)

(See page 338.)

:r :[line] r {file}

Read the specified file (default = current file) and insert it after the given line (default = current line). (Same as **:read**.)

(See page 165, 444.)

:r :[line] r !{command}

Run the given command, capture the output, and insert it after the given line (default = current line). (Same as **:read**.)

(See page 444.)

:read :[line] read {file}

Read the specified file (default = current file) and insert it after the given line (default = current line). (Same as **:r**.)

(See page 444.)

:read :[line] read !{command}

Run the given command, capture the output, and insert it after the given line. (Default = current line.) (Same as **:r**.)

(See page 444.)

The Vim Tutorial and Reference

:rec **:recover** **:rec[!] {file}**
:recover[!] {file}

Recover the editing session from the specified file. If no file is specified, the current file is used. If changes have been made to the file, this command will result in an error. If the force (!) option is present, attempting to recover a file changed in the current session will discard the changes and start recovery.

(See page 224.)

:red **:red**

Redo the last edit. (Same as: **:redo**.)

(See page 446.)

:redi **:redi[!] {>|>>} {file}**
:redir **:redir[!] {>|>>} {file}**

Copy messages to the file as they appear on the screen. If the override option (!) is present, the command will overwrite an existing file. The flag > tells the command to write the file; the >> indicates append mode. To close the output file, use the command

:redir END.

(See page 449.)

:redo **:redo**

Redo the last edit. (Same as **:red**.)

(See page 446.)

:redr **:redr[!]**
:redraw **:redraw[!]**

Redraw the screen. If the override option (!) is present, clear, then redraw the screen. This command is useful to show progress in scripts and mappings.

(See page 506.)

:redraws **:redraws[!]**
:redrawstatus **:redrawstatus[!]**

Redraw the status line of the current window. If the override option (!) is present, clear, then redraw the status line of all windows. This command is useful to show progress in scripts and mappings.

(See page 506.)

The Vim Tutorial and Reference

:reg	:reg {list}
:registers	:registers {list}
Show the registers in list. If no list is specified, list all registers. (Same as :di , :display .)	
(See page 312.)	
:res	:res [count]
Change the size of the current window to [count] . If no [count] is specified, make the window as large as possible. (Similar to: CTRL-W CTRL-_ , CTRL-W+ , CTRL-W- , CTRL-W_ , :resize .)	
(See page 89.)	
:res	:res +[count]
Increase the size of the current window by [count] . (Default = 1.) (Similar to: CTRL-W CTRL-_ , CTRL-W+ , CTRL-W- , CTRL-W_ , :resize-)	
(See page 89.)	
:res	:res -[count]
Decrease the size of the current window by [count] . (Default = 1.) (Similar to: CTRL-W CTRL-_ , CTRL-W+ , CTRL-W- , CTRL-W_ , :resize-)	
(See page 89.)	
:resize	:resize [count]
Change the size of the current window to [count] . If no [count] is specified, make the window as large as possible. (Similar to CTRL-W CTRL-_ , CTRL-W+ , CTRL-W- , CTRL-W_ , :res .)	
(See page 89)	
:resize	:resize +[count]
Increase the size of the current window by [count] . (Default = 1.) (Similar to: CTRL-W+ , :res +)	
(See page 89.)	
:resize	:resize -[count]
Decrease the size of the current window by [count] . (Default = 1.) (Similar to: CTRL-W- , :res -)	
(See page 89.)	
:ret	:{range} ret [!] {tabstop}
:retab	:{range} retab [!] {tabstop}
Replace tabs at the current tab stop with tabs with the tab stops set at {tabstop} . If the ' expandtab ' option is set, replace all tabs with	

The Vim Tutorial and Reference

space. If the force option (!) is present, multiple spaces will be changed into tabs where appropriate.

(See page 370.)

:retu	:retu {expression}
:return	:return {expression}

Return a value from a function.

(See page 499.)

:rew	:rew {file-list}
:rewind	:rewind {file-list}

Edit the first file in the list. (Same as **:fir**, **:first**.)

(See pages 80, 245, 317, 319.)

:ri	:[range] ri {width}
:right	:[range] right {width}

Right-justify the specified lines. If the width of a line is not specified, use the value of the '**textwidth**'. (If '**textwidth**' is 0, 80 is used.)

(See page 178.)

:rightb	:rightb {cmd}
:rightbelow	:rightbelow {cmd}

Execute **{cmd}** and if it owns a new window, open the window to the below or right of the current window overriding all other windowing options. (Same as **:bel**, **:belowright**.)

(See page 346.)

:ru	:ru[!] {file-spec}
------------	---------------------------

Read the command in the first file on the '**runtimopath**' that matches **{file-spec}**. If the override options (!) is present, read all the files. If this option is not present, only the first file is read. No error message is issued if no file is found. (Same as **:runtime**.)

(See page 157.)

:rub	:[range]rub {statement}
-------------	--------------------------------

Execute the Ruby statement **{statement}**. (Same as **:ruby**.)

(See page 250.)

:rub	:[range]rub << {marker}
-------------	--------------------------------------

Execute the Ruby statements that follow up to **{marker}**. (Same as **:ruby**.)

(See page 250.)

The Vim Tutorial and Reference

:ruby	: [range] ruby {statement}
	Execute the Ruby statement {statement} . (Same as :rub.) (See page 250.)
:ruby	: [range] ruby << {marker}
	Execute the Ruby statements that follow up to {marker} . (Same as :rub.) (See page 250.)
:rubyd	: [range] rubyd {command}
:rubydo	: [range] rubydo {command}
	Execute a Ruby command on a range of lines. The Ruby variable \$_ is set to each line in range. (See page 250.)
:rubyf	: [range] rubyzf {file}
:rubyfile	: [range] rubyfile {file}
	Execute the Ruby statements in {file} . (See page 250.)
:runtime	: runtime[!] {file-spec}
	Read the command in the first file on the ' runtimepath ' that matches {file-spec} . If the override options (!) is present, read all the files. If this option is not present, only the first file is read. No error message is issued if no file is found. (Same as :ru.) (See page 157.)
:rv	: rv[!] {file}
:rviminfo	: rviminfo[!] {file}
	Read the .viminfo file specified. If the override option is present (!), settings in the file override the current settings. (See page 326.)
:s	: [range] s /{from}/{to}/{flags}
	Change the regular expression {from} to the string {to} . See :substitute for a list of flags. (Same as: :substitute.) (See pages 162, 164, 232, 234, 242, 243, 434, 559.) See section <i>Substitute flags</i> on page 437 for a list of {flags}
:sa	: [count] sa[!] {number}
	Do a : [count] split followed by :argument[!] number . (Same as: :sargument.)

The Vim Tutorial and Reference

(See page 341.)

:sal

: [count] sal

:sall

: [count] sall

Open a window for all the files being edited. When a **count** is specified, open up to **count** windows. (Note that the **count** can be specified after the command--for example, **:all count**.) (Same as: **:all**.

(See page 339.)

:san

:san {cmd}

:sandbox

:sandbox {cmd}

Evaluate **{cmd}** in a sandbox. The sandbox prevents the command from modifying a number of things like the text in the buffer and is designed for the secure execution of the command.

(See page 502.)

:sargument

: [count] sargument[!] {number}

Do a **: [count] split** followed by **:argument[!] number**. (Same as: **:sa**.)

(See page 341.)

:sav

:sav[!] {file-name}

:saveas

:saveas[!] {file-name}

Save the file under a new name and change the name of the file to this name.

(See page 165.)

:sb

:sb [number]

Shorthand for **:split** and **:buffer [number]**.

(See page 92, 93.)

:sba

: [count] sba

:sball

: [count] sball

Open a window for each buffer. If a **[count]** is specified, open at most **[count]** windows. (Same as: **:ba**, **:ball**, **:sba**.)

(See page 343.)

:sbf

:sbf[!]

:sbfirst

:sbfirst[!]

Shorthand for **:split** and **:bfirst**. (Same as **:sbr**, **:sbrweind**.)

(See page 93.)

The Vim Tutorial and Reference

:sbl	:sbl[!]
:blast	:blast[!]
Shorthand for :split and :blast . (See page 93.)	
:sbm	:sbm [count]
:sbmodified	:sbmodified [count]
Shorthand for :split and :bmodified . (See page 94.)	
:sbn	: [count] sbn
Shorthand for :split followed by : [count] bnext . (Same as: :sbnext .) (See page 93.)	
:sbN	: [count] sbN
Shorthand for :split and : [count] bprevious . (Same as: :sbNext , :sbp , :sbprevious .) (See page 93.)	
:sbnext	: [count] sbnnext
Shorthand for :split followed by : [count] bnext . (Same as: :sbn .) (See page 93.)	
:sbNext	: [count] sbNext
:sbp	: [count] sbp
:sbprevious	: [count] sbprevious
Shorthand for :split and : [count] bprevious . (Same as: :sbN .) (See page 93.)	
:sbr	:sbr[!]
:brewind	:sbrewind[!]
Shorthand for :split and :brewind . (Same as :sbf , :sbfirst .) (See page 93.)	
:sbuffer	:sbuffer {number}
Shorthand for :split and :buffer {number} . (Same as: :sb .) (See page 92, 93.)	
:scr	:scr
List the name of all the sourced scripts in the order they were	

The Vim Tutorial and Reference

sourced. (Same as :scriptnames.)	
(See page 157.)	
:scripte	:scripte [encoding]
:scriptencoding	:scriptencoding [encoding]
Specify the encoding that's used for a script.	
(See page 158.)	
:scriptnames	:scriptnames
List the name of all the sourced scripts in the order they were	
sourced. (Same as :scr.)	
(See page 157.)	
:scs	:scs {arguments}
:scscope	:scscope {argument}
Split the window and handle various activities associated with the	
<i>CScope</i> program.	
(See page 247.)	
:se	:se
List all options that are not set to the default. (Same as: :set.)	
(See page 536.)	
:se	:se {option}
Set Boolean option. Deprecated. For all other types of options, show	
the value of the option.	
(See page 159, 533, 535.)	
:se	:se {option}:{value}
:se	:se {option}={value}
Set an {option} to a {value} .	
(See page 533, 535.)	
:se	:se {option}^={number}
:se	:se {option}[]!
Multiple an option by the given value.	
(See page 534.)	
:se	:se {option}&
Set the option to the default value.	
(See page 533.)	

The Vim Tutorial and Reference

:se	:se {option}+={value}
	Add a number to a numeric option. For a string option, append the {value} to the string. (See page 534, 534.)
:se	:se {option}-={number}
	Subtract a number to a numeric option. For a string option, remove the {value} from the string. (See page 534, 534.)
:se	:se {option}?
	List the value of an option. (See page 533, 535.)
:se	:se {option}^={number}
	Multiply a number to a numeric option. Prepend string to the beginning of the option. (See page 398, 534, 534.)
:se	:se all
	List all options. (See page 536.)
:se	:se all&
	Set all options to their default values. (See page 537.)
:se	:se inv{option}
	Invert a Boolean option. (See page 533.)
:se	:se no{option}
	Clear a Boolean option. (See page 533.)
:set	:set
	List all options not set to the default. (Same as: :se .) (See page 536.)
:set	:set {option}
	Set Boolean option. Deprecated. For all other types of options, show the value of the option. Deprecated: show all others. (See page 159, 533, 535.)

The Vim Tutorial and Reference

:set	:set {option}:{value}
:set	:set {option}={value}
Set an option. (See page 533, 534, 535.)	
:set	:set {option}[]
Invert a Boolean option. (See page 533.)	
:set	:set {option}&
Set the option to the default value. (See page 533, 534.)	
:set	:set {option}+={value}
Add a number to a numeric option. Append a string to a string option. (See page 534, 535.)	
:set	:set {option}-={value}
Subtract a number from a numeric option. Remove a string from a string option. (See page 534, 534, 534.)	
:set	:set {option}?
List the value of an option. (See page 533, 535.)	
:set	:set {option}^={number}
Multiply a number to a numeric option. Prepend string to the beginning of the option. (See page 398, 534, 534.)	
:set	:set all
List all options. (See page 536.)	
:set	:set all&
Set all options to their default values. (See page 537.)	
:set	:set inv{option}
Invert a Boolean option. (See page 533.)	

The Vim Tutorial and Reference

:set	:set no{option}
Clear a Boolean option. (See page 533.)	
:setf	:setf {file-type}
:set filetype	:set filetype {file-type}
Set the ' filetype ' option if it has not already been set by in this series of autocmds. (See page 117.)	
:sf	: [count] sf[!] [+command] {file}
:sfind	: [count] sfind[!] [+command] {file}
A combination of : [count] split and :find . (See page 398.)	
:sfir	:sfir[!]
:sfirst	:sfir[!]
:split followed by :rewind . If ! is specified, modified files whose windows are closed will have their contents discarded. (Same as :sr , :srewind .) (See page 341.)	
:sh	:sh
:shell	:shell
Suspend the editor and enter command mode (a.k.a. run a shell). (See pages 165, 225, 486.)	
:si	:si {char}
Simulate the pressing of Alt-{char} . (Same as :simalt .) (See page 483.)	
:sig	:sig [arguments]
:sign	:sign [arguments]
Handle the placement of "signs", that is markers in the text. This is designed for integration with other programs such as debuggers which need to annotate the code. (See page 228.)	
:sil	:sil[!] {cmd}
:silent	:silent[!] {cmd}
Execute {cmd} silently. Normal messages will not be output. With the override option (!), error message disappear as well.	

The Vim Tutorial and Reference

(See page 561.)

:simalt	:simalt {char}
	Simulate the pressing of Alt-{char} . (Same as :si.)
	(See page 483.)
:sl	:sl {seconds}
:s1	:s1 {milliseconds}m
	Sleep the specified number of seconds or milliseconds. (Same as: gs, :sleep.)
	(See page 227.)
:sla	:sla[!]!
:slast	:slast![!!]
	:split followed by :last. If ! is specified, modified files whose windows are closed will have their contents discarded.
	(See page 341.)
:sleep	:sleep {seconds}
:sleep	:sleep {milliseconds}m
	Sleep the specified number of seconds or milliseconds. (Same as: gs, :s1.)
	(See page 227.)
:sm	: [range] sm /{from}/{to}/[flags]
:smagic	: [range] smagic /{from}/{to}/[flags]
	Substitute the pattern {to} for the pattern {from} for the given range assuming that the 'magic' option is set for the duration of the command.
	(See page 435.)
:smap	:smap {lhs} {rhs}
	Define a mapping that works only in select mode.
	(See page 421.)
:smapc	:smapc
:smapclear	:smapclear
	Clear all select mode mappings.
	(See page 425.)

The Vim Tutorial and Reference

:sme :[priority] sme {menu-item} {command-string}
:smenu :[priority] smenu {menu-item} {command-string}

Define a menu item for select mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is {menu-item}, and when the command is selected, the command {command-string} is executed.

(See pages 468.)

:sn :[count] sn[!] [file-list]

:split followed by :[count] next. If ! is specified, discard any changes to buffers that have been modified, but not written. If file-list is specified, change the arguments to that list. (Same as: :snext.)

(See page 341.)

:sN :[count] sN[!]

:split followed by :[count] previous. If ! is specified, discard any changes to buffers that have been modified, but not written. (Note: The [count] parameter can be specified after the command--for example, :sN [count].) (Same as: :sNext, :spr, :sprevious.)

(See page 341.)

:snext :[count] snext[!] [file-list]

:split followed by :[count] next. If ! is specified, discard any changes to buffers that have been modified, but not written. If [file-list] is specified, change the arguments to that list. (Same as: :sn.)

(See page 341.)

:sNext :[count] sNext[!]

:split followed by :[count] previous. If ! is specified, discard any changes to buffers that have been modified, but not written. (Note: The [count] parameter can be specified after the command--for example, :sN [count].) (Same as: :sN, :spr, :sprevious.)

(See page 341.)

:sni :sni {command}

:sniff :sniff {command}

Perform a command using the interface to Sniff+. If no command is present, list out information on the current connection. Sniff+

The Vim Tutorial and Reference

support has to be compiled in for this to work (not on by default).
(See page 250.)

:sno :[range] sno /{from}/{to}/[flags]
:snomagic :[range] snomagic /{from}/{to}/[flags]

Substitute the pattern **{to}** for the pattern **{from}** for the given range assuming that the '**nomagic**' option is set.

(See page 435.)

:snoreme :[priority] snoreme {menu-item} {command-string}
:snoremenu :[priority] snoremenu {menu-item} {command-string}

Define a menu item like defined with **:smenu**, but do not allow remapping of the **{command-string}**.

(See page 473.)

:so :so {file}

Read in a session file. (Actually read in a whole set of commands.)
(Same as **:source**.)

(See pages 152, 157 348, 350, 416, 531, 560, 593.)

:sor :[range] sor [flags] [/{}pattern//]
:sort :[range] sort [flags] [/{}pattern//]

Sort a **[range]** of lines (default = entire file).

Flags are:

- ! -- Sort in reverse
- i -- Ignore case
- n -- Sort on the first decimal number
- o -- Sort on the first octal number
- r -- Sort on the pattern
- u -- Output only unique lines
- x -- Sort on the first hexadecimal number

The **{pattern}** denotes a pattern to be sorted on (**r** flag present) or text to be ignored (no **r** flag.)

(See page 237.)

:source :source {file}

Read in a session file. (Actually read in a whole set of commands.)
(Same as **:so**.)

(See pages 152, 157, 348, 350, 416, 531, 560, 593.)

The Vim Tutorial and Reference

:sp	: [count] sp [+cmd] [file-name]
	Split the current window. If a [count] is specified, make the new window [count] lines high. If a filename is present, put that file in the new window. (Otherwise, use the current file.) (Same as: :split , CTRL-W CTRL-S , CTRL-Ws , CTRL-WS .) (See pages 83, 86, 86, 134, 138, 148, 235, 236.)
:spe	: [count] spe {word}
	Add the {word} to the list of good words in the spelling list. The optional [count] select which entry in the ' spellfile ' list is used. (Same as :spellgood .) (See page 187.)
:spelld	:spelld[!]
:spelldump	:spelldump[!]
	Open a new window and dump the list of spelling words into it. If the override (!) option is used include the word count. (See page 333.)
:spellgood	: [count] spellgood {word}
	Add the {word} to the list of good words in the spelling list. The optional [count] select which entry in the ' spellfile ' list is used. (Same as :spellg .) (See page 187.)
:spelli	:spelli
:spellinfo	:spellinfo
	Display current list of spell files. (See page 187.)
:spellr	:spellr
:spellrepall	:spellrepall
	Repeat the replace done with the last z= command for all matching words. (See page 185.)
:spellu	: [count] spellu[!] {word}
:spellundo	: [count] spellundo[!] {word}
	Remove the {word} from the list of wrong wrong. If the override option (!) is present the word in removed from the internal word list.

The Vim Tutorial and Reference

If **[count]** is specified, then [count] file in '**spellfile**' is used.
(See page 187.)

:spellw :[count]spellw[!] {word}
:spellwrong :[count]spellwrong[!] {word}

Add {word} to the list of incorrect words. The override operator (!) causes it to be added to the internal word list. If a **[count]** is specified, the word is added to the **[count]** file in '**spellfile**'.

(See page 187.)

:split :[count] split [+cmd] [file-name]
Split the current window. If a **[count]** is specified, make the new window **[count]** lines high. If a filename is present, put that file in the new window. (Otherwise, use the current file.) (Same as: **:sp**, **CTRL-W** **CTRL-S**, **CTRL-Ws**, **CTRL-WS**.)
(See pages 83, 86, 86, 134, 138, 235, 236.)

:spr :[count] spr[!]
:sprevious :[count] sprevious[!]
:split followed by **:[count] previous**. If ! is specified, discard any changes to buffers that have been modified, but not written.
(Note:The **[count]** parameter can be specified after the command--for example, **:sN [count]**.) (Same as: **:sN**, **:sNext**.)
(See page 341.)

:sr :sr[!]
:srewind :srewind[!]
:split followed by **:rewind**. If ! is specified, modified files whose windows are closed will have their contents discarded. (Same as **:sfir**, **:sfirst**.)
(See page 341.)

:st :st[!]
Suspend the editor (UNIX terminal only). If the ! option is not present and '**autowrite**' is set, all changed files will be saved.
(Same as: **:stop**, **:sus**, **:suspend** and **CTRL-Z**).
(See page 227.)

:sta :[count] sta[!] {function}
:stag :[count] stag[!]! {function}
A combination of **:split** and **:tag**. If a **[count]** is specified it is the height of the new window.

The Vim Tutorial and Reference

(See page 134.)

:star

:star[!]

Begin insert mode as if a normal **i** command had been entered. If the ! is present, the insert starts at the end of line as if an **A** command had been issued. (Same as **:startinsert**.)

(See page 445.)

:startg

:startg[!]

:startgreplace

:startgreplace[!]

Begin virtual replace mode as if a normal **gR** command had been entered. If the ! is present, the insert starts at the end of line as if an **\$gR** command had been issued.

(See page 282.)

:startinsert

:startinsert[!]

Begin insert mode as if a normal **i** command had been entered. If the ! is present, the insert starts at the end of line as if an **A** command had been issued. (Same as **:star**.)

(See page 445.)

:startr

:startr[!]

:startreplace

:startreplace[!]

Begin replace mode as if a normal **R** command had been entered. If the ! is present, the insert starts at the end of line as if an **\$R** command had been issued.

(See page 433.)

:stj

:stj[!]! {ident}

:stjump

:stjump[!]! {ident}

Do a **:split** and a **:tjump**.

(See page 138.)

:stop

:stop[!]

Suspend the editor (UNIX terminal only). If the ! option is not present and '**autowrite**' is set, all changed files will be saved.

(Same as: **:st**, **:sus**, **:suspend** and **CTRL-Z**.)

(See page 227.)

:stopi

:stopi

:stopinsert

:stopinsert

Stop insert mode.

The Vim Tutorial and Reference

(See page 432.)

:sts	:sts[!] {ident}
:stselect	:stselect[!] {ident}
Do a :split and a :tselect .	
(See page 138.)	
:substitute	: [range] substitute /{[from]}/{[to]}/{[flags]}
Change the regular expression {from} to the string {to} . (Same as: :s.)	
(See pages 162, 164, 232, 234, 242, 243, 434, 559.)	
See section <i>Substitute flags</i> on page 437 for a list of {flags}	
:sun	:sun [count]
:sunhide	:sunhide [count]
Open a new window for all hidden buffer. Limit the number of window to count , if specified. (Same as: :unh , :unhide .)	
(See page 340.)	
:sus	:sus[!]
:suspend	:suspend[!]
Suspend the editor (UNIX terminal only). Actually, works in Win32 also.). If the ! option is not present and ' autowrite ' is set, all changed files will be saved. (Same as: :stop , :st , and CTRL-Z .)	
(See page 227.)	
:sv	:sv [+command] [filename]
:sview	:sview [+command] [file-name]
Split the window like :split . The only difference is that the file is opened for viewing.	
(See page 88.)	
:sw	:sw
:swapname	:swapname
List name of the current swap file.	
(See page 222.)	
:sy	:sy
List out all the syntax elements. (Same as: :syntax .)	
(See page 594.)	

The Vim Tutorial and Reference

:sy case match	:sy case match
Syntax definitions are case sensitive. In other words, the case of the letters must match.	
(See page 586.)	
:sy case ignore	:sy case ignore
Syntax definitions are case not sensitive. In other words, case differences are ignored.	
(See page 586.)	
:sy clear	:sy clear
Clear out any existing syntax definitions.	
(See page 586.)	
:sy cluster	:sy cluster {name} contains={groups} \ add={groups} remove={group}
Define a cluster of syntax groups.	
(See page 593.)	
:sy include	:sy include @{cluster} {file}
Read in a syntax file and put all the defined groups in the specified cluster. (See page 594.)	
:sy keyword	:sy keyword {group} \ {keyword} ... {keyword} {options}
Define a set of keywords for syntax highlighting. They will be highlighted according to {group-name} . The options may appear anywhere within the {keyword} list. Options can include 'contained', nextgroup , skipwhite , skipnl , ' skipempty ', and transparent . Keywords for abbreviations can be defined like abbreviation . This matches both abb and abbreviation .	
(See page 586.)	
:sy list	:sy list {group-name}
List out the named syntax groups.	
(See page 594.)	
:sy list	:sy list @{cluster-name}
List out the elements for syntax cluster.	
(See page 594.)	
:sy match	:sy match {group} excludenl {pattern} {options}
Define a regular expression that matches a syntax element. Options	

The Vim Tutorial and Reference

can be **contained**, **nextgroup**, **skipwhite**, **skipnl**, **skipempty**, **transparent**, and **contains**.

(See page 587.)

:sy off

:sy off

Turn off syntax highlighting. (Same as **:syntax off**.)

(See page 111.)

:sy on

:sy on

Turn on syntax highlighting. (Same as **:syntax on**.)

(See page 111, 156.)

:sy region

:sy region {options} matchgroup={group} keepend
 \ excludenl start={pattern}
 \ skip={pattern} end={pattern}

Define a syntax-matching region that starts and ends with the specified pattern. Options can be **contained**, **nextgroup**, **skipwhite**, **skipnl**, **skipempty**, **transparent**, '**contains**', and **oneline**.

(See page 587.)

:sy sync

:sy sync ccomment {group-name}
 \ minlines={min} maxlines={max}

Tell *Vim* to synchronize based on C-style comments. If a group name is specified, use that group for highlighting; otherwise, use the group name Comment. The '**minlines**' and '**maxlines**' options tell *Vim* how much to look backward through the file for a comment.

(See page 594.)

:sy sync clear

:sy sync clear

Remove all syntax synchronization directives.

(See page 596.)

:sy sync clear

:sy sync clear {sync-group-name}
 \ sync-group-name ...

Clear all the syntax synchronization commands for the named groups.

(See page 596.)

:sy sync match

:sy sync match {sync-group-name}
 \ grouphere {group-name} {pattern}

Define a synchronization command (in the group

The Vim Tutorial and Reference

{**sync-group-name**}) that tells Vim that when it sees {**pattern**} that the group {**group-name**} follows the match.

(See page 594.)

:sy sync match

:sy sync match {sync-group-name}

\ **groupthere {group-name} {pattern}**

Define a synchronization command (in the group {**sync-group-name**}) that tells Vim that when it sees {**pattern**} that the group {**group-name**} precedes the match.

(See page 594.)

:sy sync minlines

:sy sync minlines={min}

Define the minimum number of lines for a brute-force synchronization match.

(See page 594.)

:sy sync match

:sy sync match {match-specification}

Define a match or region to be skipped during synchronization.

(See page 594.)

:sync

:sync

:syncbind

:syncbind

Cause all scroll-bound windows to go to the same location.

(See page 387.)

:syntax

:syntax

List out all the syntax elements. (Same as: **:sy**.)

(See page 594.)

:syntax case match

:syntax case match

Syntax definitions are case sensitive. In other words, the case of the letters must match.

(See page 586.)

:syntax case ignore

:syntax case ignore

Syntax definitions are not case sensitive. In other word, case differences are ignored.

(See page 586.)

:syntax clear

:syntax clear

Clear out any existing syntax definitions.

(See page 586.)

The Vim Tutorial and Reference

:syntax cluster	:syntax cluster {name} contains={groups} \ add={groups} remove={group}
Define a cluster of syntax groups. (See page 593.)	
:syntax sync	
	:syntax sync ccomment {group-name} \ minlines={min} maxlines={max}
Tell <i>Vim</i> to synchronize based on C-style comments. If a group name is specified, use that group for highlighting; otherwise, use the group name Comment . The minlines and maxlines options tell <i>Vim</i> how much to look backward through the file for a comment. (See page 594.)	
:syntax include	:syntax include @{cluster} {file}
Read in a syntax file and put all the defined groups in the specified cluster. (See page 594.)	
:syntax keyword	:syntax keyword {group} \ {keyword} ... {keyword} options
Define a set of key words for syntax highlighting. They will be highlighted according to {group-name} . The options may appear anywhere within the {keyword} list. Options can include contained , nextgroup , skipwhite , skipnl , skipempty , and transparent . Keywords for abbreviations can be defined like abbreviation . This matches both abb and abbreviation . (See page 586.)	
:syntax list	:syntax list {group-name}
List out the named syntax groups. (See page 594.)	
:syntax list	:syntax list @{cluster-name}
List out the elements for syntax cluster. (See page 594.)	
:syntax match	:syntax match {group} excludenl {pattern} options
Define a regular expression that matches a syntax element. Options can be contained , nextgroup , skipwhite , skipnl , skipempty , transparent , and contains . (See page 587.)	

The Vim Tutorial and Reference

:syntax off	:syntax off
Turn off syntax highlighting. (Same as :sy off.) (See page 111, 156.)	
:syntax on	:syntax on
Turn on syntax highlighting. (Same as :sy on.) (See page 111.)	
:syntax region	:syntax region options matchgroup={group} \ keepend excludenl \ start={pattern} skip={pattern} end={pattern}
Define a syntax-matching region that starts and ends with the specified pattern. Options can be contained , nextgroup , skipwhite , skipnl , skipempty , transparent , contains , and oneline . (See page 587.)	
:syntax sync clear	:syntax sync clear
Remove all syntax synchronization directives. (See page 596.)	
:syntax sync clear	:syntax sync clear {sync-group-name} \ sync-group-name ...
Clear all the syntax synchronization commands for the named groups. (See page 594.)	
:syntax sync match	:syntax sync match {sync-group-name} \ grouphere {group-name} {pattern}
Define a synchronization command (in the group {sync-group-name}) that tells <i>Vim</i> that when it sees {pattern} that the group {group-name} follows the match. (See page 594.)	
:syntax sync match	:syntax sync match {sync-group-name} \ groupthere {group-name} {pattern}
Define a synchronization command (in the group {sync-group-name}) that tells <i>Vim</i> that when it sees {pattern} that the group {group-name} precedes the match. (See page 594.)	
:syntax sync minlines	:syntax sync minlines={min}
Define the minimum number of lines for a brute-force	

The Vim Tutorial and Reference

(See page 96.)

:tabf :tabf [++opt] [+cmd] {file}
:tabfind :tabfind [++opt] [+cmd] {file}

Like **:find**, but in a new tab.

(See page 96.)

:tabfir :tabfir
:tabfirst :tabfirst

Go to the first tab. (Same as **:tabr**, **:tabrewind**.)

(See page 97.)

:tabl :tabl
:tablast :tablast

Go to the last tab.

(See page 97.)

:tabm :tabm [position]
:tabmove :tabmove [position]

Move the current tab to just after **[position]**. If not specified, the tab will be moved to the last position.

(See page 346.)

:tabn :tabn [count]

Go to the next tab. If at the last tab, then wrap to the first. If a **[count]** is specified, go to the indicated tab. (Same as **:tabnext**, **<C-PageDown>**, **gt.**)

(See page 97.)

:tabN :tabN [count]

Go to the previous tab. If you are already on the first one, wrap to the last one. If a **[count]** is specified, go to the indicated tab.

(Same as **:tabNext**, **:tabp**, **:tabPrevious**, **<C-PageUp>**, **gT.**)

(See page 97.)

:tabnext :tabnext [count]

Go to the next tab. If at the last tab, then wrap to the first. If a **[count]** is specified, go to the indicated tab. (Same as **:tabn**, **<C-PageDown>**, **gt.**)

(See page 97.)

:tabNext :tabNext [count]

Go to the previous tab. If you are already on the first one, wrap to

The Vim Tutorial and Reference

the last one. If a *[count]* is specified, go to the indicated tab. (Same as :tabN, :tabp, :tabPrevious, <C-PageUp>, gT.)
(See page 97.)

:tabo :tabo[!]

:tabonly :tabonly[!]

Make this tab the only tab closing all others.

(See page 97.)

:tabp :tabp [count]

:tabprev :tabprev [count]

Go to the previous tab. If you are already on the first one, wrap to the last one. If a *[count]* is specified, go to the indicated tab. (Same as :tabN, :tabNext, <C-PageUp>, gT.)

(See page 97)

:tabr :tabr

:tabrewind :tabrewind

Go to the first tab. (Same as :tabfir, :tabfirst.)

(See page 97.)

:tabs :tabs

List each tab and the windows contained in them.

(See page 346.)

:tag :[count] tag!

Go forward *[count]* tags. (Same as: :ta.)

(See pages 131, 132, 133, 134.)

:tag :tag! /{pattern}

Search for all functions that match the regular expression defined by *{pattern}* and jump to the first one. (Same as: :ta.)

(See pages 135, 412.)

:tags :tags

List the tags.

(See page 132.)

:tc :tc {command}

:tcl :tcl {command}

Execute a single Tcl *{command}*.

(See page 251.)

The Vim Tutorial and Reference

:tcl	:[range] tcl {command}
:tcldo	:[range] tcldo {command}
Execute a Tcl {command} once for each line in the range. The variable " line " is set to the contents of the line.	
(See page 251.)	
:tclf	:tclf {file}
:tclfile	:tclfile {file}
Execute the Tcl script in the given {file}.	
(See page 251.)	
:te	:te {name}
:tearoff	:tearoff {name}
Tear off the named menu.	
(See page 474.)	
:th	:th {expr}
:throw	:throw {expr}
Throw an exception.	
(See page 446.)	
:tf	:tf [count]
:tfirst	:tfirst [count]
Go to the first tag. (Same as :tr, :trewind.)	
(See page 136, 393.)	
:tj	:tj [!] {ident}
:tjump	:tjump [!] {ident}
Like :tselect, but if there is only one tag, automatically pick it.	
(See page 136, 138, 393, 412.)	
:tl	:tl [count]
:tlast	:tlast [count]
Go to the last tag.	
(See page 136, 393.)	
:tm	:tm {menu-item} {tip}
:tmenu	:tmenu {menu-item} {tip}
Define the "tip" text that displays when the cursor is placed over an icon in the toolbar.	
(See page 471.)	

The Vim Tutorial and Reference

:tn	:[count] tn
Go to the next tag. (Same as: :tnext .) (See page 136, 393.)	
:tN	:[count] tN
Go to the previous tag. (Same as: :tNext , :tp , :tprevious .) (See page 136, 393.)	
:tnext	:[count] tnnext
Go to the next tag. (Same as: :tn .) (See page 136, 393.)	
:tNext	:[count] tNext
Go to the next tag. (Same as: :tN , :tp , :tprevious .) (See page 136, 393.)	
:to	:to {cmd}
:topleft	:topleft {cmd}
Execute a command which which opens a window and make that window to the top or left of the current window, overriding all other windowing options. (See page 346.)	
:tp	:[count] tp
:tprevious	:[count] tprevious
Go to the previous tag. (See page 136, 393.)	
:tr	:[count] tr
:trewind	:[count] trewind
Go to the first tag. (Same as :tf , :tfirst .) (See page 136, 393.)	
:try	:try
Begin a try / catch block of commands. (For scripting.) (See page 497.)	
:ts	:ts[!] [ident]
:tselect	:tselect[!] [ident]
List all the tags that match [ident] . If [ident] is not present, use the results of the last :tag command. After listing the tags, give the	

The Vim Tutorial and Reference

user a chance to select one and jump to it. (See page 135, 136, 137, 138, 393.)	
:tu	:tu {menu-item}
:tunmenu	:tunmenu {menu-item}
Remove a "tip" from an menu item. (See page 473.)	
:u	:u
Undo a change. (Same as: :undo .) (See page 286, 446.)	
:una	:una {lhs}
:unabbreviate	:unabbreviate {lhs}
Remove the abbreviation. (See page 418.)	
:undo	:undo
Undo a change. (Same as: :u .) (See page 286, 446.)	
:undoj	:undoj
:undojoin	:undojoin
Make any additional changes part of the previous undo block. The command set ends with the next user keypress. (See page 288.)	
:undol	:undol
:undolist	:undolist
List undo information. (See page 286.)	
:unh	:unh [count]
:unhide	:unhide [count]
Write the file in all windows. (Same as: :su , :sunhide .) (See page 340.)	
:unl	:unl[!] {variable}
:unlet	:unlet[!] {variable}
Remove the definition of the variable. If the force (!) option is present, do not issue an error message if the variable is not defined. (See page 491.)	

The Vim Tutorial and Reference

:unlo	:unlo[!] [depth] {name}
:unlockvar	:unlockvar[!] [depth] {name}

Unload a variable locked with **:lockvar**. The **[depth]** if specified is a code which indicates how to lock dictionaries and list. A value of 1 locks the size of the dictionary or list, but lets you change values.

The 2 code indicates th:unmenu(x1)at the top level values can not be changed. A level of 3 indicates that the array, the values in the array, and their values can not be changed. The override option (!) tell *Vim* that nothing can be changed.

(See page 491.)

:unm	:unm[!] {lhs}
:unmap	:unmap[!] {lhs}

Remove a mapping. The override option (!) is used to **:unmap** for a command in insert and command modes.

(See pages 422, 425.)

:unme	:{mode} unme {menu-item}
:unmenu	:{mode} unmenu {menu-item}

Remove the menu item named **{menu-item}**. The wildcard * will match all menu items.

(See page 473.)

:up	:{range} up[!] [file]
:up	:{range} up[!] >> [file]
:up	:{range} up !{command}
:update	:{range} update[!] [file]
:update	:{range} update[!] >> [file]
:update	:{range} update !{command}

Acts just like the **:write** command if the buffer is modified. Does absolutely nothing if it's it is not.

(See page 444.)

:v	:{range} v /{pattern}/ {command}
-----------	---

Perform **{command}** on all lines that do not have **{pattern}** in them in the given range. (Same as: **:vglobal**, **:global!**, **:g!.**)

(See page 438.)

The Vim Tutorial and Reference

:ve		:ve
	List version and configuration information, including the list of <i>.vimrc</i> files read in at startup. (Same as :version .)	
	(See page 153.)	
:verb		:[count]verb {cmd}
:verbose		:[count]verbose {cmd}
	Execute {cmd} with the ' verbose ' option set to [count] .	
	(See page 561.)	
:vert		:vert {cmd}
:vertical		:vertical {cmd}
	Execute {cmd} . If it opens a new window, perform a vertical split.	
	(See page 88, 92.)	
:version		:version
	List version and configuration information, including the list of <i>.vimrc</i> files read in at startup. (Same as :ve .)	
	(See page 153.)	
:vglobal	:[range] vglobal /{pattern}/ {command}	
	Perform {command} on all lines that do not have {pattern} in them in the given range. (Same as :v , :global! , :g!.)	
	(See page 438.)	
:vi		:vi [+cmd] {file}
	Close the current file and start editing the named file. If [+cmd] is specified, execute it as the first editing command. (Same as: :visual .)	
	(See page 159, 317, 241, 398.)	
:vie		:vie [+cmd] {file}
:view		:view [+cmd] {file}
	Like :vi , but open the file read-only.	
	(See page 77.)	
:vim	:vim[!] /{pattern}/[g][j] {file-list}	
:vimgrep	:vimgrep[!] /{pattern}/[g][j] {file-list}	
	Search the {file-list} for the given {pattern} and put the results in the quick fix list. Go to the location of the first occurrence. The g option causes lines which match {pattern} more than once to be added more than once. The j flag tells Vim to update the list but	

The Vim Tutorial and Reference

do not do the jump. The override (!) option will cause *Vim* to abandon the current buffer even if there are unsaved modifications in it.

(See page 246, 403, 406.)

:vimgrep :vimgrep[!] /{pattern}/[g][j] {file-list}
:vimgrepadd :vimgrepadd[!] /{pattern}/[g][j] {file-list}

Like **:vimgrep**, but add to the quick fix list instead of replacing it.
(See page 406.)

:visual :visual [+cmd] {file}
Close the current file and start editing the named file. If [+cmd] is specified, execute it as the first editing command. (Same as: **:vi**.)
(See page 159, 317.)

:viu :viu
:viusage :viusage
Show help on normal mode commands. This is for compatibility with Nvi. The **:help** command is much more useful for getting help.
(See page 228.)

:vm :vm
List all the mappings for visual-mode maps. (Same as: **:vmap**.)
(See pages 421, 423, 425.)

:vm :vm {lhs}
List the visual mode mapping of {lhs}. (Same as: **:vmap**.)
(See pages 421, 425.)

:vm :vm {lhs} {rhs}
Define a keyboard mapping for visual mode. (Same as: **:vmap**.)
(See pages 420, 425.)

:vmap :vmap
List all the visual-mode mappings. (Same as: **:vm**.)
(See pages 421, 423, 425.)

:vmap :vmap {lhs}
List the visual-mode mapping of {lhs}. (Same as: **:vm**.)
(See page 421, 425.)

:vmap :vmap {lhs} {rhs}
Define a keyboard mapping for visual mode. (Same as: **:vm**.)

The Vim Tutorial and Reference

(See page 420, 421, 425.)

:vmapc

:vmapc

:vmapclear

:vmapclear

Clear all the visual-mode mappings.

(See page 425.)

:vme

:[priority] vme {menu-item} {command-string}

:vmenu

:[priority] vmenu {menu-item} {command-string}

Define a menu item that is available for visual mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is **{menu-item}**, and when the command is selected, the command **{command-string}** is executed.

(See page 468.)

:vn

:vn {lhs} {rhs}

Same as **:vmap**, but does not allow remapping of the **{rhs}**. (Same as **:vnoremap**.)

(See page 425.)

:vne

:[n]vne [+opt] [+cmd] [file]

:vnew

:[n]vnew [+opt] [+cmd] [file]

Split the current window vertically and edit a new file. If no file is specified a blank window is created.

(See page 88.)

:vnoremap

:vnoremap {lhs} {rhs}

Same as **:vmap**, but does not allow remapping of the **{rhs}**. (Same as **:vn**.)

(See page 425.)

:vnoreme

:[priority] vnoreme {menu-item} {command-string}

:vnoremenu

:[priority] vnoremenu {menu-item} {command-string}

Like **:vmenu**, but the **{command-string}** is not remapped.

(See page 473.)

:vs

:[n]vs [+opt] [+cmd] [file]

:vsplit

:[n]vsplit [+opt] [+cmd] [file]

Split the window vertically and start editing the given file.

(See page 84.)

The Vim Tutorial and Reference

:vu	:vu {lhs}
:vunmap	:vunmap {lhs}
Remove the visual mode mapping for {lhs} . (See page 425.)	
:vunme	
:vunmenu	:vunme {menu-item} :vunmenu {menu-item}
Remove the visual mode menu item named {menu-item} . The wildcard * will match all menu items. (See page 473.)	
:w	:w[!]
Write out the current file. (Same as: :write .) (See page 77, 78, 165, 224, 243, 288, 338, 442.)	
:w	:[range] w[!]! filename
Write out the specified file. If no filename is specified, write to the current file. The range defaults to the entire file. If the force (!) option is present, overwrite an existing file, or override the read-only flag. (Same as: :write .) (See pages 77, 165, 243.)	
:w	:[range] w[!]! >> file
Append the specified range to the file. This will fail if the file does not exist unless the force (!) option is specified. (Same as: :write .) (See page 77, 443.)	
:wa	:wa
:wall	:wall
Write the file in all windows. (See page 338.)	
:wh	:wh {expression}
:while	:while {expression}
Start a loop. (See page 496.)	
:wi	:wi {width} {height}
Obsolete older command to set the number of rows and columns. Use :set rows and :set columns instead. (Same as: :winsize .) (See page 455.)	

The Vim Tutorial and Reference

:winc	: [count]winc {arg}
:wincmd	: [count]wincmd {arg}
Simulate a CTRL-W command of the form [count] CTRL-W {arg} . (See page 342.)	
:wind	:wind {cmd}
:windo	:windo {cmd}
Execute {cmd} for each window. (See page 342.)	
:winp	:winp {X} {Y}
:winpos	:winpos {X} {Y}
Set the position opt-columns(26-1)of the window on the screen. (See page 455.)	
:winsize	:winsize {width} {height}
Obsolete older command to set the number of rows and columns. Use :set rows and :set columns instead. (Same as: :wi.) (See page 455.)	
:wn	:count] wn[!] {+command} [files]
Shorthand for :write and : [count] next . (Same as: :wnext .) (See page 79, 81, 317.)	
:wN	: [count] wN[!]
Shorthand for :write and : [count] previous . (Same as: :wprevious , :wp , :wNext .) (See page 80, 317.)	
:wnext	: [count] wnnext[!] {+command} [files]
Shorthand for :write and : [count] next . (Same as: :wn .) (See page 79, 81, 317.)	
:wNext	: [count] wNext[!] {+command} [files]
:wp	: [count] wp[!] {+command} [files]
:wprevious	: : [count] wprevious[!] {+command} [files]
Shorthand for :write and : [count] previous . (Same as: :wN , :wNext .) (See page 80, 317.)	

The Vim Tutorial and Reference

:wq	: [range] wq[!] file
	Write the file and exit. If a range is specified, only write the specified lines. If a file is specified, write the data to that file. When the override option (!) is present, attempt to overwrite existing files or read-only files.
	(See pages 450.)
:wqa	:wqa[!]
:wqall	:wqall[!]
	Shorthand for :wall and :qall . (Same as: :xa , :xall .)
	(See page 338.)
:write	:write[!]
	Write out the current file. (Same as: :w .)
	(See page 77, 78, 165, 224, 243, 288, 338, 442.)
:write	: [range] write[!] [filename]
	Write out the specified file. If no [filename] is specified, write to the current file. The range defaults to the entire file. If the force (!) option is present, overwrite an existing file, or override the read-only flag. (Same as: :w .)
	(See pages 77, 165, 243.)
:write	: [range] write[!] >> {file}
	Append the specified range to the file. This will fail if the file does not exist unless the force (!) option is specified. (Same as: :w .)
	(See page 443.)
:ws	:ws {verb}
:wsverb	:wsverb {verb}
	Used for integration with the Sun Visual WorkShop program.
	(See page 251.)
:wv	:wv[!] {file}
:wviminfo	:wviminfo[!] {file}
	Write the .viminfo file specified. If the override option is present (!), any existing file will be overwritten.
	(See page 326.)
:x	: [range] x[!] file
	If the file has been modified, write it. Then exit. If the override (!) option is present, overwrite any existing file. (Same as: :xit .)

The Vim Tutorial and Reference

(See page 289, 450.)

:x

:x

Prompt for an encryption key and assign the resulting value to the '**key**' option.

(See page 213.)

:xa

:xa

:xall

:xall

Write all changed buffers and exit.

(See page 338.)

:xit

:*[range]* xit[!] {file}

If the file has been modified, write it. Then exit. If the override (!) option is present, overwrite any existing file. (Same as: :x, :ex!, :exit.)

(See page 289, 450.)

:xm

:xm

List all the mappings for visual-mode only maps. (Same as: :xmap.)

(See page 425.)

:xm

:xm {lhs}

List the visual mode only mapping of {lhs}. (Same as: :xmap.)

(See page 425.)

:xm

:xm {lhs} {rhs}

Define a keyboard mapping for visual mode only. (Same as: :xmap.)

(See pages 425.)

:xmap

:xmap

List all the visual-mode only mappings. (Same as: :xm.)

(See pages 425.)

:xmap

:xmap {lhs}

List the visual-mode only mapping of {lhs}. (Same as: :xm.)

(See page 425.)

:xmap

:xmap {lhs} {rhs}

Define a keyboard mapping for visual mode only. (Same as: :xm.)

(See page 425.)

The Vim Tutorial and Reference

:xmapc	:xmapc
:xmapclear	:xmapclear
Clear all the visual-mode only mappings. (See page 425.)	
:xme <i>: [priority] xme {menu-item} {command-string}</i>	
:xmenu <i>: [priority] xmenu {menu-item} {command-string}</i>	
Define a menu item that is available for visual mode only. The priority determines its placement in a menu. Higher numbers come first. The name of the menu item is <i>{menu-item}</i> , and when the command is selected, the command <i>{command-string}</i> is executed. (See page 468.)	
:xn	:xn {lhs} {rhs}
:xnoremap	:xnoremap {lhs} {rhs}
Same as :xmap , but does not allow remapping of the <i>{rhs}</i> . (See page 468.)	
:xnoreme <i>: [priority] xnoreme {menu-item} {command-string}</i>	
:xnoremenu <i>: [priority] xnoremenu {menu-item} {command-string}</i>	
Like :xmenu , but the <i>{command-string}</i> is not remapped. (See page 468.)	
:y	: [range] y {register}
:yank	: [range] yank {register}
Yank the range (default = current line) into the register (default = the unnamed register). (See page 445.)	
:z	: [line] z{code} [count]
List the given line (default = current) and a few lines after it. The code controls what section of the text is listed. The <i>[count]</i> defines what "a few" is. (See page 434.)	

:map Mode Table

needs updating -- copy from chapter 24 end

NVO	N	V	O	IC	I	C
:map	:nm	:vm	:om	:map!	:im	:cm
	:nmap	:vmap	:omap	:imap	:cmap	
:no	:nn	:vn	:ono	:no!	:ino	:cno

The Vim Tutorial and Reference

N	V	O	IC	I	C	
:noreamp	:nnremap	:vnoremap	:onoremap	:noremap!	:inoremap	:cnoremap
:unm	:nun	:vu	:ou	:unm!	:iu	:cu
:unmap	:numap	:vumap	:oumap	:unmap!	:iunmap	:cunmap
:mapc	:nmapc	:vmapc	:omapc	:mapc!	:imapc	:cmapc
:mapclear	:nmapclear	:vmapclear	:omapclear	:mapclear!	:imapclear	:cmapclear

Modes

N	Normal
V	Visual
O	Operator pending
I	Insert
C	Command

Appendix E: Visual-Mode Commands

<Esc>	Cancel visual mode. (See page 101, 103.)
	Delete the highlighted text. (Same as visual d and x) (See pages 49, 72, 99, 100, 101, 102, 351.)
CTRL-]	Jump to highlighted tag. (See page 104.)
CTRL-\ CTRL-N	Enter normal mode. (See page 102.)
CTRL-G	Toggle between select and visual mode. (See page 362.)
CTRL-V	Switch to visual block mode or exit block visual mode. (See page 103.)
! {program}	Pipe the selected text through an external program. (See page 76, 359.)
\$	Move to the end of the line and extend the highlighting to the end of all the selected lines. (See page 108, 351.)
<	Shift lines to the left (different in block visual mode.) (See page 104, 116.)
=	Indent the lines. (See page 104, 120.)
>	Shift lines to the right (different in block visual mode.) (See page 104, 116, 130.)
:{command}	Execute a colon-mode command on the selected lines. (See page 359.)
~	Invert the case of the selected text. (See page 357.)
" {register}c	Delete and enter insert mode. (Same as r and s .) (See page 51, 103.)
" {register}C	Delete the selected lines and enter insert mode. (Same as R and s .) (See pages 103 and 107.)
" {register}d	Delete the highlighted text. (Same as visual x and) (See pages 49, 72, 99, 100, 101, 102, 351.)
" {register}D	Delete the highlighted lines. (See page 102, 351.)
g@	Call the function specified by the ' operatorfunc ' to process the text. (See page 510.)
g?	Rot13 the text. (See page 358.)

The Vim Tutorial and Reference

gJ	Join the selected lines with no spaces inserted between the words. (See page 103, 357.)
gq	Format a block. (See page 156, 177, 183, 358, 373.)
gv	Toggle between the current and previous visual-mode selection. (See page 352.)
J	Join the selected lines. (See pages 53, 103, 357.)
K	Look up the selected word using the <i>man</i> command. (See page 104, 130.)
o	Jump to the other end of a visual selection. (See page 355.)
" {register}r	Delete and enter insert mode (different in block visual mode.) (Same as c and s .) (See page 103.)
" {register}R	Delete the selected lines and enter insert mode. (Same as C and S .) (See page 103 and 107.)
" {register}s	Delete and enter insert mode. (Same as c and r .) (See page 103.)
" {register}S	Delete the selected lines and enter insert mode. (Same as C and R .) (See page 103 and 107.)
u	Make the selected case all lowercase. (See page 357.)
U	Make the selected case all uppercase. (See page 357.)
v	Enter line visual mode, or exit to normal mode. (See page 103.)
" {register}x	Delete the highlighted text. (Same as visual d and) (See pages 49, 72, 99, 100, 101, 102, 351.)
" {register}X	Delete the highlighted lines. (Same as visual D) (See page 102.)
" {register}y	Yank the highlighted text into a register. (See pages 74, 74, 103, 236, 236, 351, 365)
" {register}Y	Yank the highlighted lines into a register. (See page 351.)
zf	Create fold. (See page 388.)

Visual Block Commands

>	Move the block to the right. (See page 109.)
<	Move the block to the left. (See page 109.)
A{string}<Esc>	Append string to the right side of each line. (See page 107.)
c{string}<Esc>	Delete the selected text and then insert the string on each line. (See page 106.)
C{string}<Esc>	Delete selected text to end of line, then insert on each line. (See page 106.)
I{string}<Esc>	Insert text on the left side of each line. (See page 105 and 106.)
o	Go to the other corner diagonally. (See page 356.)
r{char}	Replace all the text with a single character. (See page 108.)

Starting Select Mode

gCTRL-H	Start select block mode. (See page 360.)
gh	Start select character mode. (See page 360.)
gH	Start select line mode. (See page 360.)
gv	Do not automatically reselect an area after a command has been executed. (See page 360.)

Select Mode Commands

Arrow, CTRL, Function Keys (cursor motion)	Extend selection. (See page 360.)
{string}<Esc>	Delete the selected text and replace it with string. (See page 360.)
<BS>	Backspace. (See page 360.)
CTRL-G	Go to visual mode. (See page 362.)

The Vim Tutorial and Reference

- | | |
|---------------|--|
| CTRL-H | Delete the selected text. (See page 360.)
(See page 360.) |
| CTRL-O | Switch from select mode to visual mode for one command.
(See page 362.) |

Appendix F: Insert Mode Commands

<BS>	Delete character before the cursor. (See page 321.)
<BS>char1<BS>char2	Enter digraph (only when digraph option set). (See page 282.)
<C-End>	Cursor past end of file. (See page 321.)
<C-Home>	Cursor to start of file. (See page 321.)
<C-Left>	Cursor one word left. (See page 321.)
<C-LeftMouse>	Jump to the tag who's name is under the cursor. (See page 170.)
<C-Right>	Cursor one word right. (See page 321.)
<C-RightMouse>	Jump to the preceding tag in the stack. (See page 170.)
<CR>	Begin new line.
	Delete character under the cursor.
<Down>	Cursor one line down. (See page 321.)
<End>	Cursor past end of line. (See page 321.) 483
<Esc>	End insert mode (unless ' insertmode ' set). (See page 31, 37, 324, 420.)
<F1>	Same as <Help>.
<Help>	Stop insert mode and display help window.
<Home>	Cursor to start of line. (See page 321.)
<Insert>	Toggle insert/replace mode.
<Left>	Cursor one character left. (See page 321.)
<LeftMouse>	Move cursor to mouse click. (See page 170, 483.)
<MiddleMouse>	Paste selected text into buffer at the mouse location. (See page 170.)
<MouseDown>	Scroll three lines downward.
<MouseUp>	Scroll three lines upward.
<NL>	Same as <CR>.
<PageDown>	Scroll one screen forward. (See page 321.)
<PageUp>	Scroll one screen backward. (See page 321.)
<Right>	Cursor one character right. (See page 321.)
<RightMouse>	Extend selection from cursor to current mouse location. (See page 170.)
<S-Down>	Move one screen forward.
<S-Left>	Cursor one word left.
<S-LeftMouse>	Search for the word under the cursor. (If : behave xterm .) Extend selection from cursor to mouse click. (If : behave mswin .) (See page 170.)
<S-MouseDown>	Scroll a full page downward.
<S-MouseUp>	Scroll a full page upward.
<S-Right>	Cursor one word right.
<S-RightMouse>	Search for previous occurrence of the word under the

The Vim Tutorial and Reference

<S-Up>	cursor. (If :behave xterm.) Popup menu. (If :behave mswin.) (See page 170.)
<Tab>	Scroll one screen backward.
<Up>	Insert a <Tab> character.
CTRL-Q	Move one line up. (See page 321.)
CTRL-[Insert previously inserted text and stop insert. (See page 322.)
CTRL-\ CTRL-N	Same as <Esc>. (See page 31, 37, 324, 420.)
CTRL-]	Go to Normal mode. (See page 324.)
CTRL-_	Trigger abbreviation. (See page 420.)
CTRL-A	When 'allowrevins' is set: change language (Hebrew, Farsi) (only works when compiled with +rightleft feature). (See page 256.)
CTRL-C	Insert previously inserted text. (See page 321.)
CTRL-D	Quit insert mode, without checking for abbreviation, unless 'insertmode' set. (See page 420.)
CTRL-E	Delete one shift width of indent in the current line. (See page 119, 364.)
CTRL-G	Insert the character that is below the cursor. (See page 322.)
CTRL-H	Same as <BS>.
CTRL-I	Same as <Tab>.
CTRL-J	Same as <CR>.
CTRL-K {char1} {char2}	Insert digraph. (See page 57, 282.)
CTRL-L	When 'insertmode' is set, this command enables you to leave insert mode. (See page 258.)
CTRL-M	Same as <CR>.
CTRL-N	Find next match for keyword in front of the cursor. (See page 193.)
CTRL-O	Execute one normal mode command, then return to insert mode. (See page 258, 324, 366.)
CTRL-P	Find previous match for word in front of the cursor. (See page 192.)
CTRL-Q	Same as CTRL-V (used for terminal control flow on some terminals). (See page 139, 322, 323, 370.)
CTRL-R register	Insert the contents of a register. (See page 323, 365.)
CTRL-R CTRL-O register	Insert the contents of a register literally and do not auto-indent. (See pages 323, 365, 366.)
CTRL-R CTRL-P register	Insert the contents of a register literally and fix indent. (See pages 365, 366.)
CTRL-R CTRL-R register	Insert the contents of a register literally. (See pages 323.)

The Vim Tutorial and Reference

CTRL-S	Used for terminal control flow. Not a Vim command.
CTRL-T	Insert one shift width of indent in current line. (See page 365.)
CTRL-U	Delete all entered characters in the current line. (See page 321.)
CTRL-V character	Insert next non-digit literally. (See pages 139, 322, 323, 370.)
CTRL-V digit digit	Insert three-digit decimal number as a single byte. (See page 322.)
digit	Delete word before the cursor. (See page 321.)
CTRL-W	Enter CTRL-X sub mode, see the following entries. (See page 195.)
CTRL-X	Search for the tag that completes the word under the cursor. (See pages 195, 196).
CTRL-X CTRL-D	Search for a macro definition for completion. (See pages 195.)
CTRL-X CTRL-E	Scroll up. (See page 201.)
CTRL-X CTRL-F	Search filenames for completion. (See pages 195, 199.)
CTRL-X CTRL-I	Search the current file and <code>#include</code> files for completion. (See page 195.)
CTRL-X CTRL-K	Complete identifiers from dictionary. (See page 195, 200.)
CTRL-X CTRL-L	Search the line completion of the line under of the cursor. (See pages 195, 200.)
CTRL-X CTRL-N	Search for next word that matches the word under the cursor. (See page 195.)
CTRL-X CTRL-O	Perform omni completion. The word list for this is obtained by calling the function named in the ' <code>omnifunc</code> ' option. (See page 201.)
CTRL-X CTRL-P	Search for previous word that matches the word under the cursor. (See page 195.)
CTRL-X CTRL-T	Perform a thesaurus search. (See page 200.)
CTRL-X CTRL-U	Perform user completion. The word list for this is obtained by calling the function named in the ' <code>completesfunc</code> ' option. (See page 201.)
CTRL-X CTRL-V	Guess the type of the word before the cursor and perform the appropriate completion. (See page 200.)
CTRL-X CTRL-Y	Scroll down. (See page 201.)
CTRL-X s	Use the spelling system to search for completions. (See page 196.)
CTRL-Y	Insert the character that is above the cursor. (See pages 322.)
CTRL-Z	When in insert mode, suspend <i>Vim</i> .

The Vim Tutorial and Reference

- | | |
|-----------------|--|
| ^CTRL-D | Delete all indent in the current line, and restore it in the next. (See page 364.) |
| 0 CTRL-D | Delete all indent in the current line. (See page 364.) |

Appendix G: Option List

- A -

aleph	a1	Global Number	Default: 128 for MS-DOS 224 otherwise
ASCII code for the first letter of the Hebrew alphabet. (See page 256.)			
allowrevins	ari	Global Boolean	Default: off
All CTRL-_- to reverse the input direction. (See page 252.)			
altkeymap	akm	Global Boolean	Default: off
Used for Farsi and Hebrew input. (See page 255, 256.)			
ambiwidth	ambw	Global String	Default: single
Used for foreign languages. (See page 253.)			
antialias	anti	Global Boolean	Default: off
If enabled some fonts will be antialiased making them easier to read. (See page 483.)			
autochdir	acd	Global Boolean	Default: off
If set, change the current directory as you change files. (See page 442.)			
arabic	arab	Local Boolean	Default: off
Used for the Arabic language. (See page 254.)			
arabicshape	arshape	Global Boolean	Default: on
Used for the Arabic language. (See page 254.)			
autoindent	ai	Local Boolean	Default: off
Turn on automatic indentation. (See page 116, 118, 118, 157, 364, 552)			
autoread	ar	Global Boolean	Default off
If a file is changed outside <i>Vim</i> , just read it in instead of asking you about it. (See page 146.)			
autowrite	aw	Global Boolean	Default: off
Instead of warning you when your command is going to discard change, write out the file automatically. (See 79, 157.)			
autowriteall	awa	Global Boolean	Default: off
Turns on automatic writing for a number of commands. (See page 79.)			

- **B** -

background **bg** Global String Default: **dark** or **light** depending on terminal type.

Tells *Vim* if the default backup ground color is light or dark. (See page 112, 482.)

backspace **bs** Global String Default: empty

Defines how backup space types commands work in insert mode. (See page 151.)

backup **bk** Global Boolean Default: off

If set, create backup files. (See page 217.)

backupcopy **bkc** Global String *Vi* Default: Unix: **yes**
Otherwise: **auto**

Defines the method to be used to make a backup copy of a file. (See page 218.)

backupdir **bd** Global String Default: System dependent

A list of directories in which to place the backup. The first writable directory on the list gets the file. (See page 217.)

backupext **bex** Global String Default: ~
VMS: _

String that's added to a file name to make it a backup file name. (See page 217.)

backupskip **bsk** Global String Default: /tmp/*, \$TMPDIR/*, \$TMP/*,
\$TEMP/*

A set of file patterns that when matched will cause *Vim* to skip backup generation. (See page 218.)

balloondelay **bdelay** Global Number Default: 600

Delay in milliseconds before a balloon may pop up. (See page 251.)

ballooneval **beval** Global Boolean Default: off

Enables the balloon text. (See page 251.)

balloonexpr **bexpr** Global String Default: empty

Set the function to be called when deciding what balloon text to display. (See page 251.)

binary **bin** Local Boolean Default: off

Indicates that the file is binary. (See page 189, 257.)

bioskey **biosk** Global Boolean Default: on

When set, keyboard input is obtained from the BIOS. (See page 539.)

bomb Local Boolean Default: off

The Vim Tutorial and Reference

If set a byte order mark (BOM) is placed at the beginning of the file. (See page 253.)

breakat **brk** Global String Default: ^I!@*-+;:,./?

Defines characters which can be used for breaking long lines. (See page 332.)

browsedir **bsdir** Global String Default: **last**

Controls the initial directory for browse mode. (See page 477.)

bufhidden **bh** Local String Default: empty

If set this buffer becomes hidden. (Don't use unless you know what you are doing.) (See page 94.)

buflisted **bl** Local Boolean Default: on

If set the buffer is show in the list of buffers. If off, it is not shown. (See page 94.)

buftype **bt** Local String Default: empty

Type of buffer. (See page 94.)

- C -

casemap **cmp** Global String Default: **internal,keepascii**

Defines the character set for the case changing operators. (See page 253.)

cdpath **cd** Global String Default: equivalent to **\$CDPATH** or , ,

A set of directories that will be searched when changing directory. (See page 442.)

cedit Global String *Vi* Default: empty
 Vim Default: **CTRL-F**

The key that opens the command line window. (See page 427.)

charconvert **ccv** Global String Default: empty

Used to convert characters when different encodings are used. (See page 253.)

cindent **cin** Local Boolean Default: off

If set use C style indenting. (See page 116, 156, 365, 552.)

cinkeys **cink** Local String Default: **0{,0},0**

A set of keys that trigger **cindent** mode. (See page 376, 377.)

cinoptions **cin** Local String Default: empty

Defines how **cindent** mode works. (See page 376, 377, 379.)

cinwords **cinw** Local String Default: **if,else,while,do,for,switch**

Words that cause the next line to be indented in '**cindent**' mode. (See page 118, 376, 383.)

The Vim Tutorial and Reference

clipboard **cb** Global String X windows Default: **autoselect**,
exclude:cons\|linux
Otherwise: empty

Defines how the clipboard is used. (See page 481.)

cmdheight **ch** Global Number Default: **1**

Number of lines for command prompts. (See page 541, 542.)

cmdwinheight **cwh** Global Number Default: **7**

Number of lines that can be used for a command window. (See page 427.)

columns **co** Global Number Default: 80 or terminal width

Number of columns in the window. (See page 455.)

comments **com** Local String Default:
s1:/*,mb:*,ex:*/,://,b:#,:%,:,XCOMM,n:>,fb:-

List of strings that can start a comment. (See page 156, 374.)

commentstring **cms** Local String Default **/*%s*/**

Comment template used for creating folding markers. (See page 389.)

compatible **cp** Global Boolean Default: Off unless a *Vim* specific initialization file is found (i.e. *.vimrc*)

Controls *Vi* compatibility. (See page 28, 559.)

complete **cpt** Local String Default: **,,w,b,u,t,i**

Controls how completion works. (See page 194.)

completesfunc **cfu** Local String Default: empty

Function that is called as a result of the insert mode **CTRL-X CTRL-U** command. (See page 201.)

completeopt **cot** Global String Default: **menu,preview**

A list of options that define how insert mode completion is to be done. (See page 194.)

confirm **cf** Global Boolean Default off

Confirm operations such as **:q** with a modified buffer. (See page 541.)

conskey **consk** Global Boolean Default: off

When on, try to go directly to the BIOS for console keys. (Leave off unless you know what you are doing.) (See page 539.)

copyindent **ci** Local Boolean Default: off

If set, try to copy the existing indentation structure when reindenting a file. (See page 371.)

The Vim Tutorial and Reference

diffopt **dip** Global String Default: **filler**

Controls how diff mode works. (See page 385.)

digraph **dg** Global Boolean Default: off

Enable the entry of digraphs using **{char1}<BS>{char2}**. (See page 282.)

directory **dir** Global String Default: System dependent

Defines where a swap file can be created. (See page 223.)

display **dy** Global String Default: empty

Defines how text is displayed.

- E -

eadirection **ead** Global String Default: **both**

When '**equalalways**' is set defines if vertical windows, horizontal windows, or both are to be affected. (See page 345.)

edcompatible **ed** Global Boolean Default: off

Turning this option on make the **e** and **g** flags of **:substitute** toggle. (For *Vi* compatibility.) Please do not turn this on. (See page 559.)

encoding **enc** Global String Default: **\$LANG** or **latin1**

Sets the character encoding. (See page 562.)

endofline **eol** Local Boolean Default: on

If not set, *Vim* will not automatically add an EOL to a file that does not have one. (See page 188.)

equalalways **ea** Global Boolean Default: on

When set try and keep windows the same size. (See page 344.)

equalprg **ep** Global String Default: empty

Program to use for the = command. If not set, *Vim*'s internal indentation program will be used. (See page 373.)

errorbells **eb** Global Boolean Default: off

If set, error which would ring the bell instead flash the screen. (See page 544.)

errorfile **ef** Global String Amiga Default: **AztecC.Err**

Others: **errors.err**

Name of the error file for quickfix mode. (See page 145.)

errorformat **efm** Global String Default: Complex

Defines the format of the error messages for quickfix mode. (See page 407.)

esckeys **ek** Global Boolean *Vim* Default: on

Vi Default: off

If set, *Vim* will recognize function keys even if they send an escape sequence. (See page 540.)

The Vim Tutorial and Reference

eventignore ei Global String Default: empty

A list of event names that will be ignored. (Turns off events for some :autocomd commands.) (See page 209.)

expandtab et Local Boolean Default: off

In insert mode turn tabs into spaces if set. (See page 216, 370.)

exrc ex Global Boolean Default: off

If set, initialization files in the current directory will be read. (See page 538.)

- F -

fileencoding fenc Local String Default: empty

The current file encoding. (See page 255, 256, 257.)

fileencodings fencs Global String Default: System dependent

A list of file encodings to try when reading a file. (See page 206.)

fileformat ff Global String Default: File dependent

Current file format. (See page 188.)

fileformats ffs Global String Default: System dependent

A list of file formats which Vim will try and detect when reading a new file. (See page 188.)

filetype ft Local String Default: – File dependent.

The type of file. (See page 113, 205)

fillchars fcs Global String Default: **vert:|,fold:-**

Define the fill characters for the status line and other informational lines. (See page 547.)

fkmap fk Global Boolean Default: off

Used for Faris editing. (See page 255.)

foldclose fc1 Global String Default: empty

When set to all, folds are closed automatically when the cursor moves off of them. (See page 391.)

foldcolumn fdc Local Number Default: 0

The width of the column showing the number of lines folded. (See page 391.)

foldenable fen Local Boolean Default: on

When off, show all folds. When on, do folding normally. (See page 390.)

foldexpr fde Local String Default: **0**

The expression to be used when '**foldmethod**' is **expr**. (See page 392, 561.)

foldignore fdi Local String Default: **#**

The Vim Tutorial and Reference

Lines starting with this character will be ignored when doing indent type folding. (See page 125.)

foldlevel **fdl** Local Number Default: 0.

Defines the level of folding for this buffer. (See page 124.)

foldlevelstart **fdls** Global Number Default: -1

Initial '**foldlevel**' to be used when editing a new file. (See page 125.)

foldmarker fmr Local String Default: {{{,}}}

If we are using the marker method of folding, this option defines the markers. (See page 387.)

foldmethod **fdm** Local String Default: **manual**.

Defines how folding is to be done. (See page 123, 387, 391, 392.)

foldminlines **fml** Local Number Default: 1

The minimum number of lines for which to display a fold. (See page 391.)

foldnestmax **fdn** Local Number Default: 20

The maximum nesting level of folds. (See page 391.)

foldopen **fdo** Global String Default: **block,hor,mark,percent,quickfix,search,tag,undo**

The commands for which folds will be opened. (See page 391.)

foldtext **fdt** Local String Default: **foldtext()**

Configures how text folds are displayed. (See page 390.)

formatoptions fo Local String Vim Default: **tcq**
Vi Default: **vt**

A series of options that define how automatic formatting is to be done. (See page 156, 180, 180, 182, 183, 375, 552.)

formatlistpat **fip** Local String Default: ^\s*\d\+[\t]:.)}\t]\s*

A regular expression that is used to detect lists so that

passed to the format programming. (See page 183.)

Format	IP	Global String	Default: empty
None	file	1	1 to 8 characters

Name of the command used to format lines for `gq` if `formatexpr` is empty.
(See page 183.)

formatexpr **tex** Local String Default: empty

Expression used to format lines for **gq**. If this is not set, the lines are run through '**formatprg**'. (See page 183, 561.)

rsync **rs** Global Boolean Default: on

The Vim Tutorial and Reference

When enabled, *Vim* tries to make sure that data is written to disk using the `fsync()` system call. (See page 183.)

- G -

gdefault **gd** Global Boolean Default: off

When set all `:substitute` commands will be global. (See page 437.)

grepformat **gfm** Global String Default: `%f:%l%m, %f %l%m`

Format used to recognize what comes out of the `:grep` command. (The `:grep` command is obsolete. Use `:vimgrep` instead.) (See page 411.)

grepprg **gp** Global String Default: System dependent

Program to use for the `:grep` command. (The `:grep` command is obsolete. Use `:vimgrep` instead.) (See page 411.)

guicursor **gcr** Global String Default: System dependent

Define how the cursor looks in GUI mode. (See page 484.)

guifont **gfn** Global String Default: empty

Font to use for the GUI. (See page 482.)

guifontset **gfs** Global String Default: empty

Fonts to be used for the GUI. The first is for English, the second for special characters. (See page 252.)

guifontwide **gfw** Global String Default: empty

Font to use for double wide characters. (See page 252.)

guiheadroom **ghr** Global Number Default: **50**

The number of pixels that you expect the windowing system to use for decorations. (So when we go full screen we don't push any decorations off the display.) (See page 486.)

guioptions **go** Global String Default: System dependent

A set of options which controls the behavior and appearance of the GUI. (See page 327, 456, 482.)

guipty Global Boolean Default: on

In GUI mode, try and open a pty (Psuedo Teletype) when executing an external process. (See page 486.)

guitablabel **gtl** Global String Default: empty

Text to appear in the tab label for the GUI. (See page 347.)

guitabtooltip **gtt** Global String Default: empty

Tooltip for the tab line of a GUI. (See page 347.)

- H -

helpfile **hf** Global String Default: System dependent

Name of the top level help file. (See page 560.)

helpheight **hh** Global Number Default: 20

Minimum height of the help window. (See page 549.)

helplang **hlg** Global String Default: locale dependent

Comma separated list of languages to search for help text. (See page 40.)

hidden **hid** Global Boolean Default: off

Make buffers that become abandon hidden instead of making them just go away. (See page 94.)

highlight **hl** Global String Default: Complex string

Defines the highlighting for various system items. (See page 550.)

hlsearch **hls** Global Boolean Default: off

If set, search results are highlighted. (See page 61, 157, 290, 298, 326)

history **hi** Global Number *Vim* Default: **20**

Vi Default: **0**

The number of entries remembered in the command mode (:) and search (/) histories. (See page 325, 449.)

hkmap **hk** Global Boolean Default: off

Used for Hebrew input. (See page 256.)

hkmap **hkp** Global Boolean Default: off

Option for configuring the Hebrew keyboard. (See page 256.)

- I -

icon Global Boolean Default: off, on when title can be restored

When enabled, the name in the iconified *Vim* window will be set to the value of '**iconstring**'. If that option is not set, the name of the current file will be used. (See page 463.)

iconstring Global String Default: empty

The string to be used when the editing window is iconified. You have to have the 'icon' option enabled for this to work. (See page 463.)

ignorecase **ic** Global Boolean Default: off

Determines if searches and completion commands are case sensitive or insensitive. (See page 193, 291, 491, 525.)

imactivatekey **imak** Global String Default: empty

Specifies which keys are used by the Input Method when using the X Windows system. (See page 254.)

imcmdline **imc** Global Boolean Default: off

The Vim Tutorial and Reference

If set automatically turns the input method (for foreign languages) when enter commands. (Except for search patterns which is controlled by the '**'imsearch'** option.) (See page 254.)

imdisable **imd** Global Boolean Default: System dependent

When set, the input method is disabled. (See page 254.)

iminsert **imi** Local Number Default: System dependent

Specifies how foreign language text is to be input. (See page 254.)

imsearch **ims** Local Number Default: System Dependent

Specifies how foreign language text is to be input. (See page 254.)

include **inc** Global String Default: `^\s*#\s*include`

Pattern that matches a #include line. (See page 402.)

includeexpr **inex** Local String Default: empty

Expression that changes a included file name from text to file name. For C this is a noop, but for Java you have to change each “.” to “/”. (See page 403, 399.)

incsearch **is** Global Boolean Default: off

Turn on incremental searching. (See page 62, 157.)

indentexpr **inde** Local String Default: Language specific.

Expression which controls how indentation is done. (See page 561.)

indentkeys **indk** Local String Default: `0{,0},:,0#,!^F,o,o,e`

A set of keys that when typed will trigger an indentation event.

infercase **inf** Local Boolean Default: off

When doing replacement and word completion, see if you can determine the correct case of the new word. (See page 193.)

insertmode **im** Global Boolean Default: off

When enabled, *Vim* works in insert mode as Default:. This option is useful for people who don't know *Vim* and don't want to learn. However, if you're reading this book, you are not one of those people. (See page 258.)

isfname **isf** Global String Default: System dependent

Determines which characters are considered part of a file name. (See page 263.)

isident **isi** Local String Default: system dependent

Determines which characters are part of an identifier. (See page 263.)

iskeyword **isk** Global String Default: Operating system and compatibility mode dependent.

Defines what characters are considered part of a keyword. (See page 131, 261.)

The Vim Tutorial and Reference

isprint **isp** Global String Default: System dependent

Determines which characters are printable. (See page 263.)

- J -

joinspaces **js** Global Boolean Default: on

When enabled, *Vim* will insert spaces when two lines are joined. (See page 179.)

- K -

key Local String Default: empty

The current encryption key. (See page 213.)

keymap **kmp** Local String Default: empty

Set the keyboard mapping. (See page 254.)

keymodel **km** Global String Default: empty

A quick and dirty way for setting the behavior of certain special keys. (See page 169, 467.)

keywordprg **kp** Global String System dependent.

Program that's used when the **k** command is executed. (See page 131.)

- L -

langmap **lmap** Global String Default: empty

Define a keyboard map for a foreign language. (See page 252.)

langmenu **lm** Global String Default: empty

Language for menus. (See page 474.)

laststatus **ls** Global Number Default: 1

Determines when a status line is displayed at the bottom of a window. (See page 344.)

lazyredraw **lz** Global Boolean Default: off

If set, redrawing will be done as little as possible in order to save terminal bandwidth. (See page 558.)

linebreak **lbr** Local Boolean Default: off

If enabled *Vim* will try to break long lines in nice places. (See page 331.)

lines Global Number Default: Terminal height.

Number of lines to use for the editing window. (See page 455.)

linespace **lsp** Global Number Default: 0

Windows: 1

Amount of whitespace in pixels to add between lines in the GUI. (See page

The Vim Tutorial and Reference

485.)

lisp Local Boolean Default: off

Turns on a number of options to make editing Lisp easier. A holdover from the *Vi* days. (See page 216, 552, 559.)

lispwords lw Global String Default: Long and complex.

Words to use for Lisp indenting. (See page 559.)

list Local Boolean Default: off

Make all the invisible characters visible. Looks ugly, but you can see everything. Good for finding out where you have tabs and where you have just a series of spaces. (See page 138, 433, 549.)

listchars lcs Global String Default: **eol:\$**

When the '**list**' option is set, this controls how the invisible characters are displayed on the screen. (See page 138, 549.)

loadplugins lpl Global Boolean Default: on

If set, load plugins from the plugin directory. (See page 155.)

- M -

macatsui Global Boolean Default: on

When set, enabled a hack to work around some drawing problems which are only seen on Macintosh computers. Set it to off if you are on a Mac and experience drawing problems.

Ideally this option should go away as soon as the underlying bug is fixed. (See page 562.)

magic Global Boolean Default: on

Changes the way certain pattern matching characters work. (See page 304, 435, 525.)

makeef mef Global String Default: empty

Name of the error file for :**make**. If no string is specified, a temporary file is used. If you put "##" in the file name it will be replaced by a unique number. (See page 406.)

makeprg mp Global String Default: **make**

VMS: **MMS**

The program to use when :**make** is executed. (See page 406.)

matchpairs mps Local String Default: **(:),{:},[:]**

Things that can be considered matching pairs. Examples include (), [], and <>. (See page 394.)

matchtime mat Global Number Default: **5**

The time, in tenths of a second, to show the matching parenthesis (assuming '**showmatch**' is set.) In order to be compatible with *Nvi*, this option uses tenths of a second, while every other timing option uses milliseconds. (See page 394.)

The Vim Tutorial and Reference

maxcombine **mco** Global Number Default: **2**

Maximum number of combining characters that are displayed when entering text. (You can enter more, you just can't see them.) (See page 254.)

maxfuncdepth **mfd** Global Number Default: **100**

Maximum length of the function call stack. (See page 557.)

maxmapdepth **mmd** Global Number Default: **1000**

Number of times one mapping can trigger another. This prevents bad mappings from going on endlessly. (See page 557.)

maxmem **mm** Global Number Default: System dependent

Maximum amount of memory to use for one buffer. (See page 557.)

maxmempattern **mmp** Global Number Default: **1000**

Maximum amount of memory to use for pattern matching. (See page 557.)

maxmemtot **mmt** Global Number Default: System dependent

Total amount of memory to use for buffers. (See page 557.)

menuitems **mis** Global Number Default: **25**

Maximum number of items for a menu like the buffer menu who's contents are generated automatically. (See page 469.)

mkspellmem **msm** Global String Default: **460000,2000,500**

Sets tuning parameters for the **:mkspell** command. (See page 333.)

modeline **ml** Local Boolean *Vim* Default: on,
Vi Default: off

If enabled Vim will check for mode lines within the files. These lines contain Vim directives and are used to supply file specific to Vim. (See page 216, 537.)

modelines **mls** Global Number Default: **5**

If non-zero the first and last '**modelines**' number of lines is searched for a mode line. (See page 372.)

modifiable **ma** Local Boolean Default: on

When enabled, the buffer can be modified. (See page 212.)

modified **mod** Local Boolean Default: off

Set automatically when the buffer is modified. (See page 225.)

more Global Boolean *Vim* Default: on
Vi Default: off

When enabled, if the screen fills with messages, you'll get a "More" prompt. (See page 552.)

mouse Global String Default: empty
GUI, MS-DOS and Win32: **a**

Enable the use of the mouse for certain types of text terminals. (See page 465.)

mousefocus **mousef** Global Boolean Default: off

If enabled focus is automatically switched to whatever window is under the mouse. (See page 464.)

The Vim Tutorial and Reference

mousehide **mh** Global Boolean Default: on

If enabled the mouse is hidden when typing text. (Actually mis-named, the cursor is hidden. The mouse remains visible right next to your keyboard.) (See page 466.)

mousemodel **mousem** Global strings Default: **extend**

MS-DOS and Win32: **popup**

Defines how the mouse works when selecting text. (See page 169, 464, 466.)

mouseshape **mouses** Global String Default: **i:beam,r:beam,s:updown,**
sd:cross, m:no,ml:up-arrow,
v:rightup-arrow

Defines how the mouse should look in various modes. (See page 484.)

mosetime **moset** Global Number Default: **500**

Maximum time between the two mouse clicks of a double click. (See page 466.)

mzquantum **mzq** Global Number Default: **100**

Controls polling times for MzScheme threads. (See page 249.)

- N -

nrformats **nf** Local String Default: **octal,hex**

Defines the number formats recognized by *Vim*. Can be any combination of **octal** and **hex**. (Decimal is always recognized.) (See page 278, 552.)

number **nu** Local Boolean Default: off

Show line numbers in front of each line. (See page 46, 159, 365.)

numberwidth **nuw** Local Number *Vim* Default: **4**

Vi Default: **8**

When line numbering is turned on, the minimum number of columns for the line number. (See page 550.)

- O -

omnifunc **ofu** Local String Default: empty

Function to be used for the insert omni completion commands (**CTRL-X CTRL-O**). (See page 201.)

operatorfunc **opfunc** Global String Default: empty

The function called by the **g@** operator. (See page 510.)

osfiletype **oft** Local String Default: System dependent.

For systems which support a file type, the file type to write. (See page 258.)

- P -

paragraphs para Global String Default: **IPLPPPQPP LIPplpipbp**

When editing troff files, this option specifies the macros that start a paragraph. (See page 190.)

paste Global Boolean Default: off

When enabled a lot of other options are set so you can cut text from another window and easily paste it in *Vim*. (See page 552.)

pastetoggle pt Global String Default: empty

Define a key that will toggle the 'paste' option. (See page 552.)

patchexpr pex Global String Default: empty

Expression to use for patching files. (See page 384.)

patchmode pm Global String Default: empty

When set, if a file of this extension does not exist, *Vim* will use it to create a backup file. Once this file is created, it is not overwritten. (See page 217.)

path pa Global String Default: System dependent

Path on which to look for file when using **gf**, **:find**, and similar commands. (See page 194, 398Error: Reference source not found.)

preserveindent pi Local Boolean Default: off

When redoing the indentation, attempt to preserve the existing indentation as much as possible. (See page 371.)

previewheight pvh Global Number Default: **12**

Default height of the preview window. (See page 549.)

previewwindow pww Local Boolean Default: off

If set, then this window is the preview window. (Only one preview window is allowed.) (See page 394.)

printdevice pdev Global String Default: empty

Name of the device to be used for hardcopy output. (See page 451.)

printencoding penc Global String System dependent

Character encoding for **:hardcopy** output. (See page 451.)

printexpr pexpr Global String System dependent.

The command that prints the Postscript file produced for **:hardcopy** output. (See page 451.)

printfont pfn Global String Default: **courier**

Name of the font for **:hardcopy** output. (See page 451.)

printheader pheader Global String Default: **%<%f%h%m%=>Page %N**

The Vim Tutorial and Reference

Specifies the header for :**hardcopy** output. (See page 451.)

printmbcharset **pmbcs** Global String Default: empty

Controls the character set for CJK (Korean) printing. (See page 451.)

printmbfont **pmbfn** Global String Default: empty

Controls the fonts to be used for CJK (Korean) printing. (See page 451.)

printoptions **popt** Global String Default: empty

Controls how :**hardcopy** printing is done. (See 166.)

prompt Global Boolean Default: on

When on a ":" prompt is used in Ex mode. (See page 159.)

pumheight **ph** Global Number Default: 0

Determines the maximum number of items to show in the popup menu for Insert mode completion. When zero as much space as available is used.

- Q -

quoteescape **qe** Local String Default: \

The string to use for quoting characters when a shell command is issued. (See page 447.)

- R -

readonly **ro** Local Boolean Default: off

If enabled, no changes are allowed to the current file. (See page 225.)

remap Global Boolean Default: on

Allows for mappings to work recursively. (See page 424.)

report Global Number Default: 2

When more than this many lines are changed, *Vim* will output a short message telling you how many where changed. (See page 548.)

restorescreen **rs** Global Boolean Default: on

When set *Vim* will attempt to restore the original terminal screen when existing. (See page 552.)

revins **ri** Global Boolean Default: off

Inserting character will work right to left. (See page 252, 552.)

rightleft **rl** Local Boolean Default: off

Enables right to left mode. (See page 252.)

rightleftcmd **rlc** Local String Default: **search**

Edit right to left instead of left to right. (See page 252.)

The Vim Tutorial and Reference

ruler **ru** Global Boolean Default: off

Show ruler. (See page 269, 548, 552.)

rulerformat **ruf** Global String Default: empty

Defines the look of a ruler line if set. (See page 548.)

runtimepath **rtp** Global String Default: System dependent

Path to use for locating runtime files. (See page 157, 532.)

- S -

scroll **scr** Local Number Default: half the window height

Number of lines to scroll with **CTRL-U** and **CTRL-D** commands. (See page 269.)

scrollbind **scb** Local Boolean Default: off

All windows with '**scrollbind**' set will scroll together. (See page 385.)

scrolljump **sj** Global Number Default: 1

Minimum number of lines to scroll when scrolling is needed. (See page 272.)

scrolloff **so** Global Number Default: 0

Minimum of lines to keep above or below the cursor before the screen scrolls. (See page 272.)

scrollopt **sbo** Global String Default: **ver, jump**

Determines how scrolling is done. (See page 386.)

sections **sect** Global String Default: **SHNHH HUnhsh**

For users of troff, controls the macros that denote a section start. (See page 191.)

secure Global Boolean Default: off

When set, locks *Vim* down so that a user can't do bad things like start a shell or perform other insecure commands. (See page 538.)

selection **sel** Global String Default: **inclusive**

Controls how selection mode works. (See page 169, 483.)

selectmode **s1m** Global String Default: empty

Defines what starts select mode. (See page 169, 360, 466.)

sessionoptions ssop Global String Default: **blank, buffers, curdir, folds, help, options, tabpages, winsize"**

Controls what goes into a session created by the :**mksession** command. (See page 348, 348.)

shell **sh** Global String Default: System dependents.

The Vim Tutorial and Reference

Name of the shell program to use for shell commands. (See page 447, 530.)

shellcmdflag **shcf** Global String Default: System dependent.

The flag you have to send to the shell to tell it that a command files. (See page 447.)

shellpipe **sp** Global String Default: System dependent

Determines how shell output is piped to the user and a file. (See page 447.)

shellquote **shq** Global String Default: System dependent

Determines show shell arguments are quoted. (See page 447.)

shellredir **srr** Global String Default: System dependent

Controls how the output of shells are redirected to a file. (See page 447, 530.)

shellslash **ssl** Global Boolean Default: off

When set a forward slash is used when expanding file names. (See page 447.)

shelltemp **stmp** Global Boolean *Vi* Default: off,

Vim Default: on

When enabled use temporary files for shell output. When disabled, pipes are used. (See page 447.)

shelltype **st** Global Number Default: **0**

Determines how shell commands are spawned on an Amiga. (See page 258.)

shellxquote **sxq** Global String Default: System dependent

Determines how arguments are quoted for shell commands. Don't fiddle with this unless you know what you are doing. (See page 447.)

shiftround **sr** Global Boolean Default: off

When set, indent commands move things to the a rounded indent stop. When clear, shift commands move things a full '**shiftwidth**' regardless of the current position. (See page 372.)

shiftwidth **sw** Local Number Default: **8**

Number of spaces for indenting. (See page 115, 157, 365, 365, 368, 377.)

shortmess **shm** Global String *Vim* Default: **filnxtToo**

Vi Default: empty

POSIX Default: **A**

A set of letters that control when short messages are used instead of long ones. This help prevent excessive Hit Enter prompts due to long messages. (See page 543.)

shortname **sn** Local Boolean Default: off

When turned on, file names are restricted to the MS-DOS 8.3 style. You have to have a pretty old and restricted system to need this option. (See page 225.)

The Vim Tutorial and Reference

showbreak **sbr** Global String Default: empty

When wrapping lines, this string is put at the beginning of the continuation line. The character ">" is a common value for this option. (See page 332.)

showcmd **sc** Global Boolean *Vim* Default: on
Unix: off
Vi Default: off

When typing a multi-character command, show the portion already entered on the status line. (See page 542, 542.)

showfulltag **sft** Global Boolean Default: off

When completing a tag in insert mode, show not only the tag (procedure name) but the full tag (procedure name and arguments). (See page 198.)

showmatch **sm** Global Boolean Default: off

When enabled, inserting a parenthesis, bracket, or similar character causes the cursor to temporarily jump to the matching character. (See page 216, 394, 552.)

showmode **smd** Global Boolean *Vim* Default: on
Vi Default: off

When set, show the current mode in the status line. (See page 542, 542.)

showtabline **stal** Global Number Default: 1

Tells *Vim* when to display the tab line at the top of the screen. Values include:

- 0 – Never
- 1 – Only if two or more tabs are present.
- 2 – Always

(See page 348.)

sidescroll **ss** Global Number Default: 0

Minimum number of character to scroll when moving horizontally. The default (0) causes the cursor to jump to the middle of the screen when scrolling. (See page 272.)

sidescrolloff **siso** Global Number Default: 0

If you have '**nowrap**' set, the minimum number of columns to keep on either side of the cursor (if possible). (See page 272.)

smartcase **scs** Global Boolean Default: off

If enabled and you have '**ignorecase**' then typing in a mixed character string will cause '**ignorecase**' to be ignored. (See page 292.)

smartindent **si** Local Boolean Default: off

Enables an indenting algorithm that mostly works for C like languages, but has been obsoleted by '**cindent**'. (See page 116, 118, 552)

smarttab **sta** Global Boolean Default: off

The Vim Tutorial and Reference

When enabled a <Tab> at the front of the line inserts spaces or blanks according to your 'shiftwidth'. In the middle of a line, it inserts tabs. (See 368.)

softtabstop **sts** Local Number Default: **0**

Number of columns that are used for tabstops when the <Tab> key is pressed. Depending on your settings, this may result in a tab or spaces being inserted. (See 368, 552.)

spell Local Boolean Default: off

When on spell checking will be done. (See page 184.)

spellcapcheck **spc** Local String Default: [.?!]_[\`]

Regular expression used to find an end of sentence. (See page 333.)

spellfile **spf** Local String Default: empty

Name of the file where *Vim* is to store words added to the dictionary. (See page 186, 187.)

spelllang **spl** Local String Default: **en**

A list of languages to use for spelling. (See page 185.)

spellsuggest **sps** Global String Default: **best**

A list of methods to be used for deciding how to suggest spelling corrections. Values include **best**, **double**, **fast**, {number}, **file:{filename}**, **expr:{expression}**. (See page 333.)

splitbelow **sb** Global Boolean Default: off

When enabled, the split command creates a new window below the current one. (See page 345.)

splitright **spr** Global Boolean Default: off

When enabled, the split command creates the new window to the right of the current one. (See page 345.)

startofline **sol** Global Boolean Default: on

When enabled, changes the behavior of some movement commands so that they also move the cursor to the first non-blank column of the line. If disabled, Vim attempts to keep the cursor in the same column. (See page 557.)

statusline **stl** Global String Default: empty

Define how the status line is displayed. (See page 545.)

suffixes **su** Global String Default: **.bak,~, .o, .h, .info, .swp, .obj**

When doing a wildcard match, put files with these suffixes last on the list. (See '**wildignore**' for how to completely ignore files.) (See page 517, 554.)

suffixesadd **sua** Local String Default: empty

The Vim Tutorial and Reference

List of suffixes used when searching for a file using **gf**. (See page 399.)

swapfile **swf** Local Boolean Default: on

If set, use a swap file for this buffer. (See page 223.)

swapsync **sfs** Global String Default: **fsync**

When set, causes a sync operation to be executed each time the swap file is written. This causes *Vim* to tell the operating system to flush the disk buffers so that the data is really written to the disk. If set to **sync** all buffers for all files will be flushed. Setting it to the null string turns off this operation. (See page 223.)

switchbuf **swb** Global String Default: empty

Controls how buffers are displayed when switching buffers. (See 95, 96, 410.)

synmaxcol **smc** Local Number Default: **3000**

Limit on the number of columns to be used for searching for syntax highlighting elements. (0 is no limit.) (See page 596.)

syntax **syn** Local String Default: empty

The name of the syntax to use for syntax highlighting. This can actually contain more than one language, for example **c.doxygen** tells *Vim* to use C syntax highlighting and then the doxygen highlighting. The special string ON and OFF turn syntax highlighting on and off. (See page 417.)

- T -

tabline **tal** Global String Default: empty

Defines how the tab line (the line at the top of the screen containing a list of tabs) looks in terminal (non-GUI) mode. (See page 348.)

tabpagemax **tpm** Global Number Default: **10**

Maximum number of tabbed panes that can be opened from the command line. (See page 98.)

tabstop **ts** Local Number Default: **8**

Number of spaces for each tab stop. (This value was chosen because the old Teletype terminals had tab stops hardwired in at 8. Most programs honor this value, so don't change this value unless you are ready to deal with a lot of problems external programs.) (See page 365, 368, 370.)

tagsearch **tbs** Global Boolean Default: on

If enabled, use a binary search to locate tags. This is faster for large tagfiles, but won't work if the tagfile is not sorted. (See page 411.)

taglength **tl** Global Number Default: **0**

If non-zero, tags are significant up to this number of characters. (See page 412.)

The Vim Tutorial and Reference

tagrelative **tr** Global Boolean Default: on

If enabled, then tags are resolved relative to the directory where the tagfile resides. (See page 412.)

tags **tag** Global String Default: **./tags, tags**

If compiled with with the
+emacs_tags:

./tags, ./TAGS, tags, TAGS

A list of files which contains the tags for use with tag related commands. (See page 412.)

tagstack **tgst** Global Boolean Default: on

When enabled, the tagstack acts normally. When disabled, the **:tag** and **:tselect** commands with an argument will not push a tag onto the stack. (See page 412.)

term on String Default: System dependent

The name of the terminal being used. (See page 558.)

term bidi **tbidi** Global Boolean Default: off
mlterm: on

If set, the terminal handles bidirection text instead of *Vim*. (See page 253.)

termencoding **tenc** Global String Default: System dependent

Character encoding system used by the GUI. (See page 253.)

terse Global Boolean Default: off

When set, output terse messages. See the '**shortmess**' option for details. (See page 544.)

textauto **ta** Global Boolean *Vim* Default: on
Vi Default: off

This option is obsolete. Use '**fileformats**'. (See page 562.)

textmode **tx** Local Boolean Operating system dependent

This option is obsolete. Use '**fileformat**'. (See page 562.)

textwidth **tw** Local Number Default: 0

The width of the text being inserted. Lines longer than this will be automatically wrapped. A value of 0 turns off automatic line wrapping. (See page 157, 175, 180, 181, 183, 183, 216, 552.)

thesaurus **tsr** Global String Default: empty

Name of the file used for the thesaurus completion commands (**CTRL-X CTRL-T**). (See page 194,223.)

tildeop **top** Global Boolean Default: off

The Vim Tutorial and Reference

When set the tilde (~) command acts like an operator. If not set, the tilde (~) command acts like a motion command. (See page 282, 560.)

timeout **to** Global Boolean Default: on

ttimeout Global Boolean Default: off

These two options control whether *Vim* will timeout in the middle of a function key sequence or sequence defined by the :map command. The values are:

'timeout'	'ttimeout'	Action
off	off	do not time out
on	n.a.	Time out on both :map and function key sequences
off	on	time out key codes

(See page 540.)

timeoutlen **tm** Global Number Default: 1000

ttimeoutlen **ttm** Global Number Default: -1

These two options control the time (in milliseconds) allowed between characters when *Vim* is looking for a sequence of keystrokes. Terminal function keys send out such sequences, but sequences can also be defined using the :map command.

Normally only '**timeoutlen**' is used. If '**ttimeoutline**' is set to then it is used for function key sequences, and '**timeoutlen**' is then only used for :map sequences. (See page 540.)

title Global Boolean Default: Terminal dependent

If set, *Vim* will attempt to change the title of the terminal window in which it is running. (See page 461.)

titlelen Global Number Default: 85

What percentage of the window to use for the title. (See page 462.)

titleold Global String Default: **Thanks for flying Vim**

When *Vim* exists, it tries to restore the original title. If it can't it will use the value of this option as the title to be displayed after *Vim* has stopped. (See page 463.)

titlestring Global String Default: empty

The title of the window. (See page 463.)

toolbar **tb** Global String Default: **icons, tooltips**

Controls what goes into the toolbars. Possible values are: **icons, text, horiz, tooltips**. (See page 461.)

toolbariconsize **tbis** Global String Default: **small**

The Vim Tutorial and Reference

The size of the icons in the toolbar. Values are **`tiny`**, **`small`**, **`medium`**, and **`large`**. (See page 462.)

ttybuiltin **tbi** Global Boolean Default: on

When enabled Vim will search its internal terminal database before searching any external ones. (See page 558.)

ttyfast **tf** Global Boolean Default: on for most terminals

If set indicates you have a fast terminal and *Vim* can redraw the screen using a system which sends more characters, but looks better. (See page 558.)

ttymouse **ttym** Global String Default: depends on '**term**'

Controls how mouse events are sent to *Vim* through a terminal. (See page 558.)

ttyscroll **tsl** Global Number Default: **999**

Maximum number of lines to scroll the screen. If there are more lines to scroll the window is redrawn. For terminals where scrolling is very slow and redrawing is not slow this can be set to a small number, e.g., 3, to speed up displaying. (See page 559.)

ttytype **tty** Global String Default: from \$TERM

Alias for 'term', see above. (See page 558.)

- 11 -

undolevels ul Global Number 100

Unix, VMS, Win32, OS/2: **1000**

Maximum number of changes that can be undone. (See page 284.)

updatecount **uc** Global Number Default: **200**

The number of characters you can type before the swap file is written. (See page 222.)

updatetime **ut** Global Number Default: **4000**

If set, the number of milliseconds of idle time before the swap file is written.
(See page 206, 222.)

- V -

verbose **vbs** Global Number Default: 0

Controls how much debugging information is displayed. (See page 560.)

verbosefile **vfile** Global String Default: empty

If set, the name of a file where verbose messages are logged. (See page 561.)

The Vim Tutorial and Reference

Macintosh: \$VIM:vimfiles:view
VMS: sys\$login:vimfiles/view
RiscOS: Choices:vimfiles/view

Directory where view files (**:mkview**) are stored. (See page 350.)

viewoptions vop Global String Default: **folds,options,cursor**

Controls what's saved using the **:mkview** command. (See page 350.)

viminfo vi Global String *Vi* mode: empty
Vim mode for MS-DOS, Windows and OS/2: '**20,<50,s10,h,rA:,rB:**'
Vim mode for other systems: '**20,<50,s10,h**'

A string defining what data is stored in a *.viminfo* file. (See page 324-326, 488.)

virtualedit ve Global String Default: empty

Tells *Vim* when you can do virtual editing. (See page 281.)

visualbell vb Global Boolean Default: off

Use a visual bell instead of beeping. (See page 545.)

- W -

warn Global Boolean Default: on

Give a warning message when a shell command is used while the buffer has been changed. (See page 544.)

weirdinvert wiv Global Boolean Default: off

Provided for backward compatibility with version 4.x. Mostly it's here to get around some problems with strange terminals. (See page 559.)

whichwrap ww Global String *Vim* Default: **b,s**
Vi Default: empty

Specifies which characters are allowed to move the cursor past the end of line or beginning on line. (See page 268.)

wildchar wc Global Number *Vim* mode: <**Tab**>
Vi mode: **CTRL-E**

Character that starts wildcard expansion in command line mode. (See page 553.)

wildcharm wcm Global Number Default: **0**

The character number of a character will act like '**wildchar**' when used inside macros. (See page 553.)

wildignore wig Global String Default: empty

A list of file patterns of files that you want to ignore when doing command line completion. For example, you probably want to ignore all object files because

The Vim Tutorial and Reference

you never edit them in *Vim*. (See page 517, 554.)

wildmenu **wmnu** Global Boolean Default: off

When '**wildmenu**' is enabled then attempting to complete a command line command with <Tab> may result in a menu being shown giving you a list of possible completions to accept. (See page 554.)

wildmode **wim** Global String Default: **full**

Along with '**wildchar**', controls how completion mode operates. Values include:

<empty>	Complete only the first match
full	Cycle through each complete match.
longest	Complete the longest possible string. If this does not result in a completion, then use the next completion mode.
longest:fill	Like longest, but start ' windmenu ' mode if enabled.
list	If more than one match, show a list of possible matches.
list:full	When more than one match, list all matches, then select the first full match.
list:longest	When there is more than one match, list all the matches and then select the longest string.

(See page 555.)

wildoptions **wop** Global String Default: empty

In command mode the **CTRL-D** key causes a matching list of files to be displayed. When this option is set to **tagfile**, a matching list of tags and files will be displayed.

winaltkeys **wak** Global String Default: **menu**

If set to "no", the ALT key will work like it normally does in Microsoft Windows. It will select menu items. If set to "menu", then ALT is just another keyboard modifier that can be used for *Vim* commands. (See page 483, 539.)

window **wi** Global Number Default: screen height - 1

When **CTRL-F**, <PageUp>, **CTRL-B**, and <PageDown> are used to move the screen up or down a page, this option defines the number of lines in a page. (See page 270, 272.)

winheight **wh** Global Number Default: 1

Minimum height of the current window. (See page 344.)

winfixheight **wfh** Local Boolean Default: off

Do not change the height of this window even if '**equalalways**' is set. (See page 345.)

winfixwidth **wfw** Local Boolean Default: off

Do not change the width of this window even if '**equalalways**' is set. (See page

The Vim Tutorial and Reference

345.)

winminheight **wmh** Global Number Default: **1**

Minimum height of a window that's not the current window. (See page 344.)

winminwidth **wmw** Global Number Default: **1**

Minimum width of the windows that are not current. (See page 344.)

winwidth **wiw** Global Number Default: **20**

Minimum width of the current window. (See page 344.)

wrap Local Boolean Default: on

If set, long lines will wrap around and be displayed on the screen. If not set the ends of long lines go off the screen and can only be seen when you use horizontal scrolling. (See page 327, 331.)

wrapmargin **wm** Local Number Default: **0**

When typing characters with auto-wrap turned on, the number of characters to use for a right margin. (See page 176, 552.)

wrapscan **ws** Global Boolean Default: on

If set searches wrap past the beginning or end of the file. (See page 293.)

write Global Boolean Default: on

If set, the user is allowed to write files. If not set, you can look but not touch (write) files. (See page 557.)

writeany **wa** Global Boolean Default: off

If set, when writing files do not force the user to use the overwrite (!) option if a file is going to overwrite an existing file. (See page 557.)

writebackup **wb** Global Boolean Default: on with **writebackup** feature off otherwise

If set, then files are written by making a backup, writing the file, and deleting the backup. (See page 218.)

writedelay **wd** Global Number Default: **0**

A debugging option which inserts the specified amount of time (in milliseconds) between each character output. Basically a simple way of slowing down the editor so you can see how it redraws the screen. (See page 560.)

Appendix H: Vim License Agreement

VIM LICENSE

- I) There are no restrictions on distributing unmodified copies of Vim except that they must include this license text. You can also distribute unmodified parts of Vim, likewise unrestricted except that they must include this license text. You are also allowed to include executables that you made from the unmodified Vim sources, plus your own usage examples and Vim scripts.
- II) It is allowed to distribute a modified (or extended) version of Vim, including executables and/or source code, when the following four conditions are met:
 - 1) This license text must be included unmodified.
 - 2) The modified Vim must be distributed in one of the following five ways:
 - a) If you make changes to Vim yourself, you must clearly describe in the distribution how to contact you. When the maintainer asks you (in any way) for a copy of the modified Vim you distributed, you must make your changes, including source code, available to the maintainer without fee. The maintainer reserves the right to include your changes in the official version of Vim. What the maintainer will do with your changes and under what license they will be distributed is negotiable. If there has been no negotiation then this license, or a later version, also applies to your changes. The current maintainer is Bram Moolenaar <Bram@vim.org>. If this changes it will be announced in appropriate places (most likely vim.sf.net, www.vim.org and/or comp.editors). When it is completely impossible to contact the maintainer, the obligation to send him your changes ceases. Once the maintainer has confirmed that he has received your changes they will not have to be sent again.
 - b) If you have received a modified Vim that was distributed as mentioned under a) you are allowed to further distribute it unmodified, as mentioned at I). If you make additional changes the text under a) applies to those changes.
 - c) Provide all the changes, including source code, with every copy of the modified Vim you distribute. This may be done in the form of a context diff. You can choose what license to use for new code you add. The changes and their license must not restrict others from making their own changes to the official version of Vim.
 - d) When you have a modified Vim which includes changes as mentioned under c), you can distribute it without the source code for the changes if the following three conditions are met:
 - The license that applies to the changes permits you to distribute the changes to the Vim maintainer without fee or restriction, and permits the Vim maintainer to include the changes in the official version of Vim without fee or restriction.
 - You keep the changes for at least three years after last distributing the corresponding modified Vim. When the maintainer

The Vim Tutorial and Reference

or someone who you distributed the modified Vim to asks you (in any way) for the changes within this period, you must make them available to him.

- You clearly describe in the distribution how to contact you. This contact information must remain valid for at least three years after last distributing the corresponding modified Vim, or as long as possible.
 - e) When the GNU General Public License (GPL) applies to the changes, you can distribute the modified Vim under the GNU GPL version 2 or any later version.
 - 3) A message must be added, at least in the output of the ":version" command and in the intro screen, such that the user of the modified Vim is able to see that it was modified. When distributing as mentioned under 2)e) adding the message is only required for as far as this does not conflict with the license used for the changes.
 - 4) The contact information as required under 2)a) and 2)d) must not be removed or changed, except that the person himself can make corrections.
- III) If you distribute a modified version of Vim, you are encouraged to use the Vim license for your changes and make them available to the maintainer, including the source code. The preferred way to do this is by e-mail or by uploading the files to a server and e-mailing the URL. If the number of changes is small (e.g., a modified Makefile) e-mailing a context diff will do. The e-mail address to be used is <maintainer@vim.org>
- IV) It is not allowed to remove this license from the distribution of the Vim sources, parts of it or from a modified version. You may use this license for previous Vim releases instead of the license that they came with, at your option.

Author's Note

By Steve Oualline

The people behind *Vim* have spent a lot of time and effort to make one of the best editors in the world. Yet they do not ask anything for themselves; instead, they ask that you help some of the poorest and most needy people in Africa. Please send them a donation.

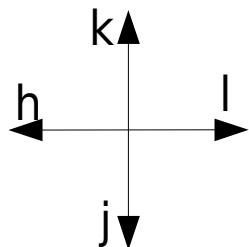
If you work for a medium-size or large company, please take the time to tell your boss how much using *Vim* has helped you and encourage your company to make a substantial donation.

The people behind *Vim* are good people. Please help them out.

Appendix I: Basic Vim Quick Reference

Appendix J: Vim Quick Reference

Minimum Command Set. Learn This First.



Basic Movement

Basic Movement

[number]j – Down

[number]k – Up

[number]h – Left

[number]r – Right

Editing Commands

u – Undo

CTRL-R – Redo

i{text}<Esc> – Insert text in front of cursor

a{text}<Esc> – Insert text after cursor

[number]x – Delete characters

[number]dd – Delete Lines

Getting Out

zz – Write file and exit

:q! – Abort edits and discard all work (since editing started or the last :write command).

Note: On Linux and Unix you must enable the Vim command set

Use the command:

```
$ touch .vimrc
```

to create a .vimrc file which tells Vim it's OK to act like Vim.

Vertical Movement / Scrolling

[number]CTRL-B – Screen down (also <PageUp>)

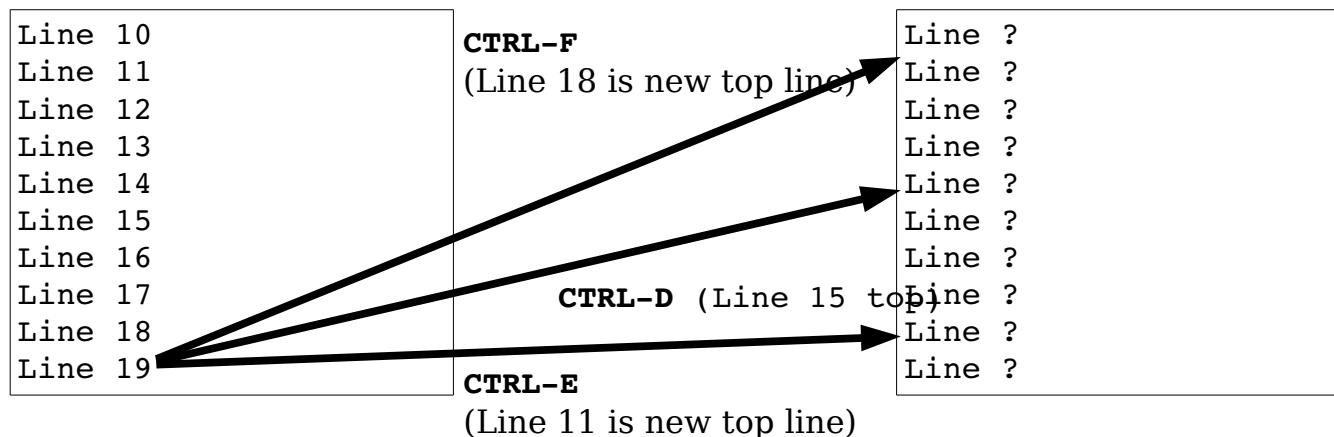
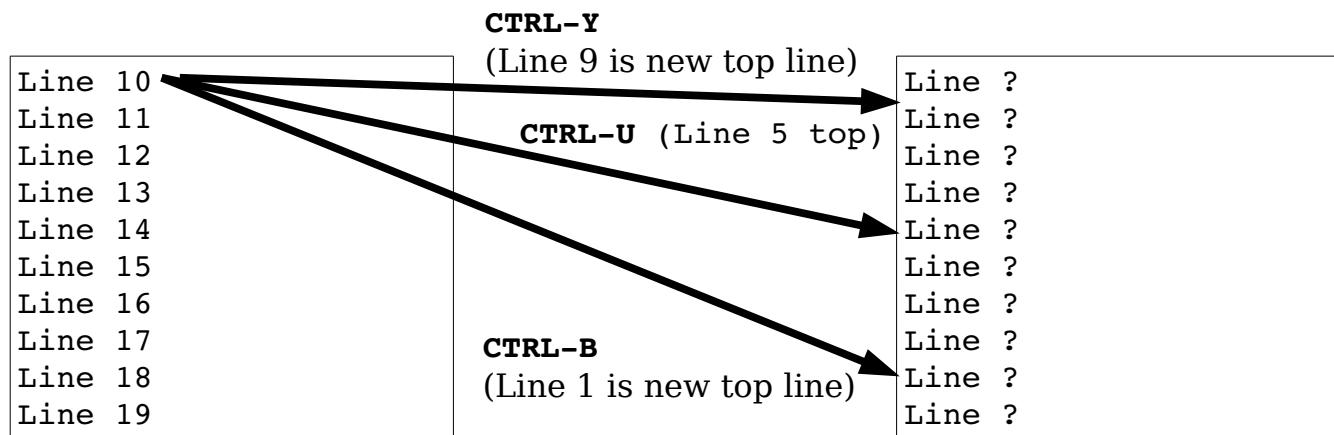
[number]CTRL-U – ½ Window down (Actually the number of lines defined by 'scroll'. The **[number]** set the size of the movement ('scroll'), down not specify the number of moves.)

[number]CTRL-Y – Lines down

[number]CTRL-F – Screens up

[number]CTRL-D – ½ Window up. Actually the number of lines defined by 'scroll'. The **[number]** set the size of the movement ('scroll'), down not specify the number of moves.)

[number]CTRL-E – Lines up



The Vim Tutorial and Reference

- [number]-** – Go up **[number]** lines to start
- [number]<CR>** – Forward to start of the next **[number]** line
- [number]_** – Cursor to start of **[number]** lines lower

User interfaces are interesting. First we had text, but that was not good enough. Then we had icons, but then the icons got to confusing, so we added tooltips. So now when you place a cursor over an icon you get text. The icon you can't understand with the text you can, replaced the original text menus.

- (minus) <CR> 2-

- m{mark}** – Place mark.
- ` {mark}** -- Go to mark.
- ' {mark}** – Go to first non-blank character of the line containing **{mark}**
- {count}%** – Go to **{count}** percent of the file. The **{count}** must be specified because % with no **{count}** goes to the matching brace, parenthesis, or curly bracket.
- [number]G** -- Go to line **[number]** (default = last line).

Screen Location

- [number]H** line from top of screen
- M** Middle of the screen
- [number]L** line from bottom of the screen

H → "What's the fastest way to move 500GB of data daily from Santa Cruz to Los Angeles?"
M → Answer: FedEx.
L →

Screen Redrawing

[count]z<cr> Redraw with this line at the top of the window. If **[count]** is specified, the line will be **[count]** lines from the top.

```
A dozen, a gross, and a score,  
Plus three times the square root of  
four,  
Divided by seven,  
Plus five time eleven,  
Equals nine squared plus zero, no  
more.
```

z<cr>

```
Plus five time eleven,  
Equals nine squared plus zero, no  
more.  
***  
A UNIX sales lady, Lenore,
```

[count]z{height}<cr>

Like **z<cr>** only set the window height as well.

[count]z+ Redraw the screen with **[count]** line at the top. If no **[count]** specified, this command acts like **z<cr>**.

```
Line 10  
Line 11  
Line 12  
Line 13  
Line 14
```

2z+

```
Line 2  
Line 3  
Line 4  
Line 5  
Line 6
```

[count]z- Put line **[count]** at the bottom of the screen. If no **[count]** is specified, put the current line at the bottom.

```
Line 10  
Line 11  
Line 12  
Line 13  
Line 14
```

12z-

```
Line 8  
Line 9  
Line 10  
Line 11  
Line 12
```

[count]z. Put line **[count]** at the center of the screen. If no **[count]** is specified, put the current line at the center.

```
Line 10  
Line 11  
Line 12  
Line 13  
Line 14
```

33z.

```
Line 31  
Line 32  
Line 33  
Line 34  
Line 35
```

Virtual movement

j – Move up	gj – Virtual Move up
k – Move down	gk – Virtual Move down
l – Move right	gl – Virtual Move right
h – Move left	gh – Virtual Move left

'**virtualedit**' option values:

- block** Allow virtual editing in Visual block mode.
- insert** Allow virtual editing in Insert mode.
- all** Allow virtual editing in all modes.
- onemore** Allow the cursor to move just past the end of the line

Commands for English Text

:set tw={number}

Set the width for automatic text wrapping. (Does not wrap existing text.)

:set spell Turn on spell checking

z= Suggest corrections for the misspelled word under the cursor.

[s Previous spelling error

]s Next spelling error

Text Movement

' (Sentence start

') Sentence end

[number] (Sentences backward

[number]) Sentences forward

' { Paragraph start

' } Paragraph end

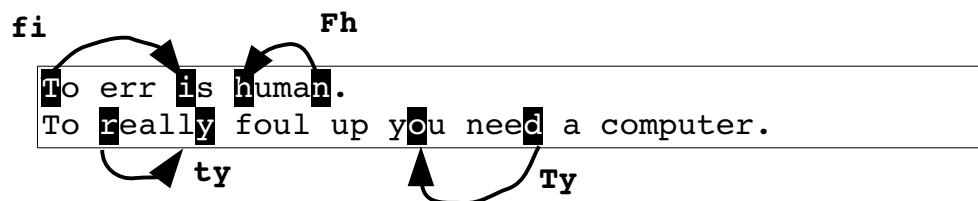
[number] { Paragraph backwards

[number] } Paragraph forward.

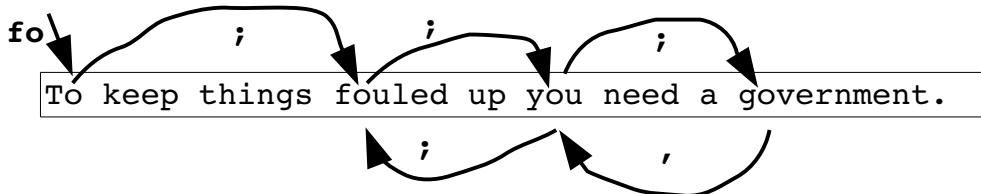
gq{motion} Format text

Horizontal Movement

- [number]fx** Single character forward search, stop on character.
- [number]Fx** Single character reverse search, stop on character.
- [number]tx** Single character forward search, stop before character.
- [number]Tx** Single character forward search, stop before character.

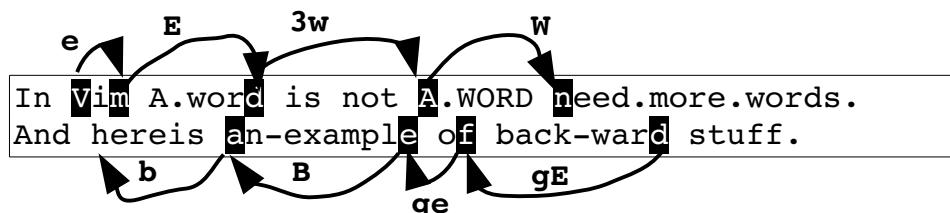


- [number];** Repeat last single character search
- [number],** Repeat last single character search in the other direction



The Vim Tutorial and Reference

- [number]e** Forward to end of word.
- [number]E** Forward to end of WORD.
- [number]w** Forward to start of word.
- [number]W** Forward to start of WORD.
- [number]b** Backward to start of word.
- [number]B** Backward to start of WORD.
- [number]ge** Backward to end of word.
- [number]gE** Backward to end of WORD.



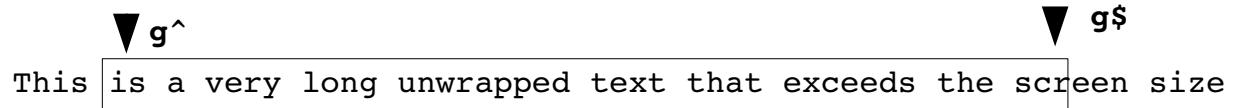
^ First character of the line

\$ Last character on the line



g0 First character of the screen line

g\$ Last character of the screen line



m{mark} Place mark

' {mark} Go to line containing mark

` {mark} Go to character containing mark

Changing text

[number]i{text}<ESC>

Insert text at the current cursor position..

[number]I{text}<ESC>

Insert text at the beginning (first no blank character) of the line.

[number]a{text}<ESC>

Insert text after the cursor.

[number]A{text}<ESC>

Append text to the end of the line.

[number]O Open [number] line above

[number]o Open [number] lines below

xp Reverse characters

[number]{“{register}}d{move}

Delete text from the current position to where the move takes you.

[number]{“{register}}dd

Delete lines.

[number]{“{register}}D

Delete to the end of line (and [number]-1 more lines).

[number]{“{register}}P

Put text in front of the cursor.

[number]{“{register}}P

Put text in back of the cursor.

The Vim Tutorial and Reference

[“{register}][number]c{motion}{text}<ESC>

Change text.

[“{register}][number]cc{text}<ESC>

Change lines of text.

[“{register}][number]C{text}<ESC>

Change text from the cursor to the end of the line.

[number]r{char}

Replace character under the cursor.

[number]R{text}<ESC>

Replace a series of characters.

[number]gr{char}

Replace a single virtual character.

[number]gR{text}<ESC>

Replace a series of virtual characters.

[number]J Join lines

[number]gJ Join lines without adding a space

[number][“{register}]s{text}<ESC>

Delete **[number]** characters, then enter insert mode.

[number][“{register}]S{text}<ESC>

Delete **[number]** lines, then enter insert mode.

u Undo all the changes on a line (twice redoes them).

[number][“{register}]x

Delete characters before the cursor.

The Vim Tutorial and Reference

[number] [{"register}]y{motion}

Yank text into a register. (Copy.)

[number] [{"register}]yy

Yank lines into a register. (Copy.)

[number] [{"register}]Y

Yank to the end of the line.

[number]gU{motion}

Make the text upper case.

[number]gUgU

[number]gUU Make lines upper case.

[number]gu{motion}

Make the text lower case.

[number]gugu

[number]guu Make lines lower case.

[number]g?{motion}

Encode / decode text with rot13.

[number]g?g?

[number]g?? Encode / decode lines with rot13.

[number]!! {motion} {command}

Filter lines through **{command}**.

[number]!!

Filter lines through **{command}**.

Windows

[number]CTRL-W +

Increase the current window's height.

[number]CTRL-W -

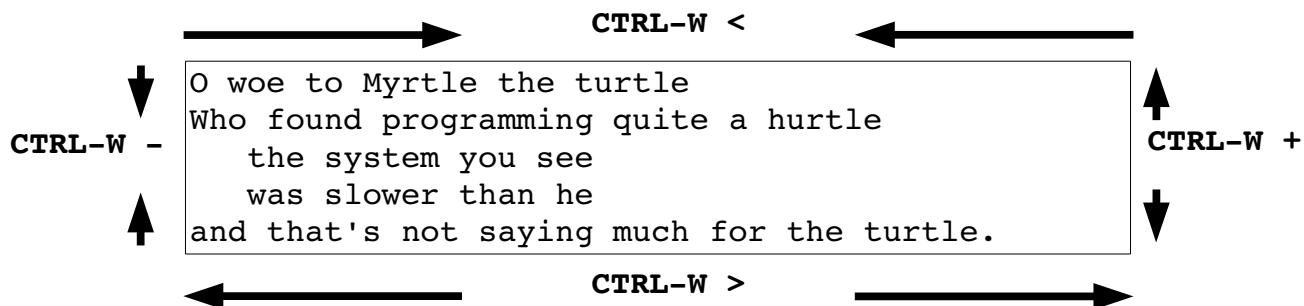
Decrease the current window's height.

[number]CTRL-W <

Increase the current window's width.

[number]CTRL-W >

Decrease the current window's width.



CTRL-W = Make all windows the same height.

[number]CTRL-W _

Set window height.

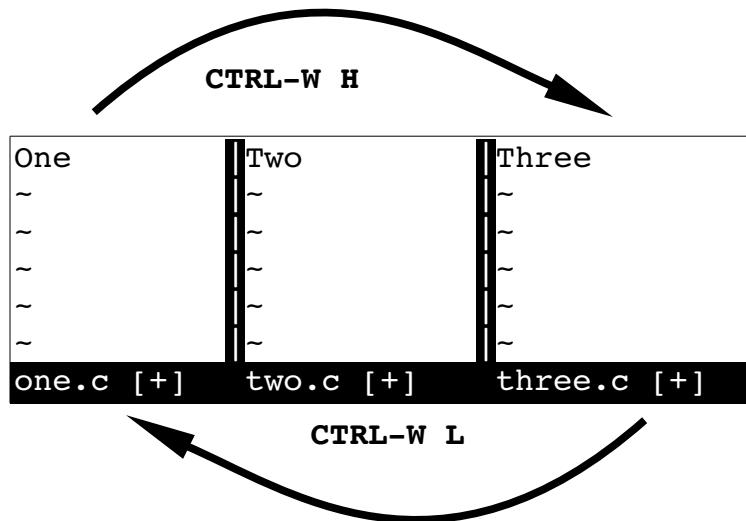
[number]CTRL-W |

Set window width.

CTRL-W H Move the window to far left.

CTRL-W L Move the window to far right

The Vim Tutorial and Reference

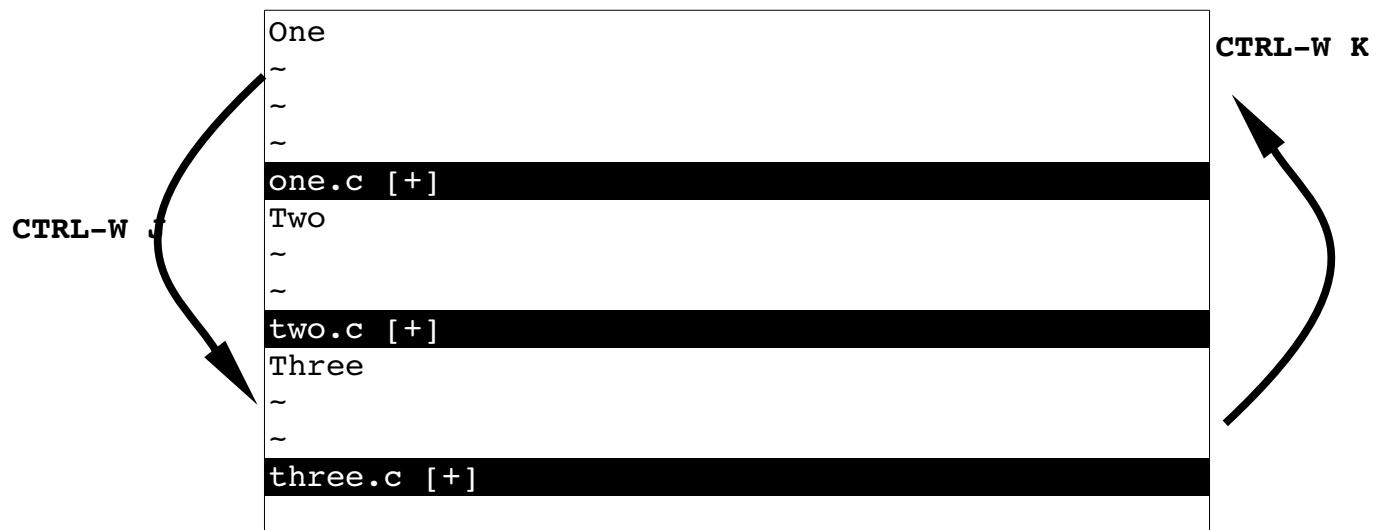


CTRL-W K

Window to top

CTRL-W J

Window to bottom



CTRL-W R

Rotate windows up.

CTRL_W r

Rotate windows down.

The Vim Tutorial and Reference



CTRL_W x Exchange window



CTRL-WW Previous window (wrap)

CTRL-Wn Next window (wrap)

CTRL-Wb Bottom window

CTRL-Wj Window up

CTRL-Wk Window down

CTRL-Wl Window left

CTRL-Wh Window left

CTRL-Wt Window top

CTRL-WP Preview window

The Vim Tutorial and Reference

Others

CTRL-Ws	Split current window
CTRL-Wv	Vertical split
CTRL-W^	Split and edit alternate file
CTRL-Wf	Split and edit file under cursor
CTRL-WF	Split and edit file with line number
CTRL-Wgf	Edit file with new tab
CTRL-WgF	Same thing with line number
CTRL-Wn	New window
CTRL-WT	Make window a new tab.
CTRL-Wc	Close window
CTRL-Wo	Close everyone else (only window)
CTRL-Wq	Quit window
CTRL-Wz	Close preview

Multiple Files

:args {file-list}

Set the file list to the given files. Wildcards like “*”, “**”, and “?” may be used.

CTRL-^ Edit alternate file

:n Next file in the argument list

:p Previous file in the argument list

:first First file on the list

:last Last file on the list.

gf Edit the file who's name is under the cursor

gF Like **gf** but if a line number follows, position the cursor on that line.

CTRL-W gf Edit the file who's name is under the cursor in a new tab.

Searching

*	Search word forward for the word under the cursor
#	Search word backward for the word under the cursor
g#	Search string forward for the word under the cursor
g*	Search string backwards for the word under the cursor
/ {text}	Search forward
? {text}	Search reverse
n	Repeat search
N	Repeat search backward

Search Pattern Reference

Simple atoms

Pattern	Description	Example	Result
<code>\abc</code>	Literal	<code>/abc</code>	<code>abcdef cba</code>
<code>\\"</code>	Literal \	<code>/\\</code>	<code>a\b</code>
<code>\/</code>	Literal /	<code>/\/</code>	<code>a/b</code>
<code>\.</code>	Literal .	<code>/\.</code>	<code>a.b a+b</code>
<code>\{</code>	Literal {	<code>/\{</code>	<code>a{b a+b</code>
<code>\[</code>	Literal [<code>/\[</code>	<code>a[b a+b</code>
<code>\\$</code>	Literal \$	<code>/\\$</code>	<code>a\$b a+b</code>
<code>\^</code>	Literal ^	<code>/\^</code>	<code>a^b a+b</code>
<code>\%dnnn</code>	The character who's decimal number is <i>nnn</i>		
<code>\%xnn</code>	The character who's hexadecimal number is <i>nn</i>		
<code>\%onnn</code>	The character who's octal number is <i>nnn</i>		
<code>\%unnnn</code>	The multibyte character who's number is <i>nnnn</i>		
<code>\%Unnnnnnnn</code>	The large multibyte character who's number is <i>nnnnnnnn</i>		
<code>\e</code>	<Esc> Character		
<code>\t</code>	<Tab> Character		
<code>\r</code>	<CR> Character		
<code>\b</code>	<BS> Character		
<code>\n</code>	End of line		

Character Classes

Pattern	Description	Example	Result
<code>\a</code>	Alphabetic character	<code>\a</code>	<code>ab123(*&ab</code>
<code>\i</code>	Identifier (defined by ' <code>isident</code> ')	<code>\i</code>	<code>abc def a10b</code>
<code>\I</code>	Identifier excluding digits	<code>\I</code>	<code>abc def a10b</code>
<code>\k</code>	Keyword (defined by ' <code>iskeyword</code> ')	<code>\k</code>	<code>abc def a10b</code>
<code>\K</code>	Keyword excluding digits	<code>\K</code>	<code>abc def a10b</code>
<code>\f</code>	File name (defined by ' <code>isfname</code> ')	<code>\f</code>	<code>sam.txt /root/sam.txt f10.dat</code>
<code>\F</code>	File name excluding digits	<code>\F</code>	<code>sam.txt /root/sam.txt f10.dat</code>
<code>\p</code>	Printable (defined by ' <code>isprint</code> ')	<code>\p</code>	<code>sam.txt ^A^B^C f10.dat</code>
<code>\P</code>	Printable name excluding digits	<code>\P</code>	<code>sam.txt ^A^B^C f10.dat</code>
<code>\s</code>	Whitespace	<code>\s</code>	<code>Space tab done</code>

The Vim Tutorial and Reference

Pattern	Description	Example	Result
\s	Non-whitespace	\s	Space tab done
\d	Digit	\d	1234 abc xyz
\D	Non-digit	\D	1234 abc xyz
\x	Hex-digit	\x	1234 abc xyz
\X	Non-hex digit	\X	1234 abc xyz
\o	Octal-digit	\o	1234567890 abc xyz
\O	Non-octal digit	\O	1234567890 abc xyz
\w	Word character (a-z,A-Z,0-9, underscore)	\w	_test10 -- a -- test
\W	Non-word character	\W	_test10 -- a -- test

Modifiers

Pattern	Description	Example	Result
\c	Ignore case	\cabc	ABC abc
\C	Match case	\ca\cbc	aBC abc ABC
\m	Turn ' magic ' off		
\M	Turn ' magic ' on		
\v	Turn very magic on		
\V	Turn off very magic		
\z	Ignore differences in Unicode combination characters		

Grouping and Repeats

Pattern	Description	Example	Result
\~	Last substitute string		
\(xx\)	Grouping	/\(\(ab\)\)\1	abab abx
\1	First match inside \(...\)		
\9	Ninth match inside \(...\)		

Sets of Characters

Pattern	Description	Example	Result
[abc]	Any of the given [abc] characters	abcdefgaaxxyycc	abcdefgaaxxyycc
[a-f]	From a to f	[a-f]	axbxefxx
[^abc]	Any character except a,b, or c	[^a-f]	axbxefxx
\%[string]	Match as much of string as possible	a\%[bcd]	abxx abcxx

The Vim Tutorial and Reference

Anchors

Pattern	Description	Example	Action
<code>^</code>	Start of line	<code>^abc</code>	<code>abc abc abc</code>
<code>\$</code>	End of line	<code>abc\$</code>	<code>abc abc abc</code>
<code>_^</code>	Start of line (can be inside pattern)	<code>abc_^def</code>	<code>line with abc</code> <code>def on next</code>
<code>_\$</code>	End of line (can be inside pattern)	<code>abc_\$def</code>	<code>line with abc</code> <code>def on next</code>
<code>\<</code>	Beginning of word	<code>\<abc</code>	<code>abc xxabc</code>
<code>\></code>	End of word	<code>abc\></code>	<code>abcxx xxabc abcyy</code>
<code>\%^</code>	Beginning of file	<code>\%^abc</code>	<code>abc</code>
<code>\%\$</code>	End of file	<code>abc\%\$</code>	<code>abc</code>
<code>\%v</code>	Inside visual area		<code>abc</code>
<code>\%#</code>	Cursor position		
<code>\%51</code>	Inside line 55		
<code>\%51</code>	Inside column 5		
<code>\%71</code>	Inside virtual column 7		
<code>\%'m</code>	Mark <i>m</i> position		
<code>\zs</code>	Set start of match		
<code>\ze</code>	Set end of match		

Repeats and Wildcards

Pattern	Description	Example	Result
<code>.</code>	Any single character	<code>a.c</code>	<code>abc axc ac</code>
<code>_.</code>	Any single character (eol included)	<code>a_.c</code>	<code>abc a c</code>
<code>*</code>	Repeat 0 or more times	<code>ab*</code>	<code>ax abx abbx</code>
<code>\+</code>	Repeat 1 or more times	<code>ab\+</code>	<code>ax abx abbx</code>
<code>\=</code>	Zero or one times	<code>ab\=</code>	<code>ax abx abbx</code>
<code>\?</code>	Zero or one times	<code>ab\?</code>	<code>ax abx abbx</code>
<code>\{n,m}</code>	Between n to m times	<code>ab\{2,3}</code>	<code>ab abb abbb abbbb</code>
<code>\{n}</code>	Exactly n times	<code>ab\{2}</code>	<code>ab abb abbb abbbb</code>

The Vim Tutorial and Reference

Pattern	Description	Example	Result
\{n,}	At least n times	ab\{2,}	ab abb abbb abbbb
\{,m}	m or less times	ab\{,2}	ab abb abbb abbbb
\{}	0 or more times (like *)	ab\{\}	ax ab abb abbb abbbb
\{-n,m}	Between n to m times, as few as possible	ab\{-2,3}	ab abb abbb abbbb
\{-n}	Exactly n times, as few as possible	ab\{-2}	ab abb abbb abbbb
\{-n,}	At least n times, as few as possible	ab\{-2,}	ab abb abbb abbbb
\{-,m}	m or less times, as few as possible	ab\{-,2}	ab abb abbb abbbb
\{-}	0 or more times (like *), as few as possible	ab\{-}	ax ab abb abbb abbbb

Choices

Pattern	Description	Example	Result
a b	a or b	a b	abcdef cba
a\&b	a followed by b	a\&b	ab ac

Zero Width Conditionals

Pattern	Description	Example	Result
\@>	Makes the preceding atom a whole pattern	\(a*\)ab	aaab
\@=	Zero with pattern match for the preceding atom which must follow.	\(a*\)ab foo\(\bar\)\@\=	aaab foobar foofoo
\@!	Zero with pattern not matching for the preceding atom which must follow.	foo\(\bar\)\@\!	foobar foofoo
\@<=	Zero with pattern match for the preceding atom which must come before.	f[ox]*\(\oo\)\@\=<	foxo fxoo
\@<!	Zero with pattern non-match for the preceding atom which must come before.	f[ox]*\(\oo\)\@\<!	foxo fxoo

For programmers

Command	Go to	If one tag	If many	Result in
:tag {tag}	Tag	Jump	Jump	Same window
:tselect {tag}	Tag	Select	Select	Same window
:tjump {tag}	Tag	Jump	Select	Same Window
:stag {tag}	Tag	Jump	Jump	New window
:stselect {tag}	Tag	Select	Select	New window
:stjump {tag}	Tag	Jump	Select	New Window
:ptag {tag}	Tag	Jump	Jump	Preview Window
:ptselect	Tag	Select	Select	Preview Window
:ptjump	Tag	Jump	Select	Preview Window
CTRL-]	Tag	Jump	Jump	Same Window
g]	Tag	Select	Select	Same Window
g CTRL-]	Tag	Jump	Select	Same Window
gf	File	Jump	N.A.	Same Window
CRL-Wf	File	Jump	N.A.	Same Window
CTRL-Wgf	File	Jump	N.A.	New tab
CTRL-W]	Tag	Jump	Jump	New Window
CTRL-Wg]	Tag	Select	Select	New Window
CTRL-wgCTRL-]	Tag	Jump	Select	New Window
CTRL-W}	Tag	Jump	Jump	Preview Window
CTRL-Wg}	Tag	Jump	Select	Preview Window

CTRL-Wd Split and jump to definition

CTRL-Wi Split and jump to declaration of id

[number]>> Right shift lines

>{motion} Right shift lines to **{motion}**.

<< Left shift

<{motion} Left shift lines to **{motion}**.

={motion} Reindent

=% Reindent to matching curly bracket

The Vim Tutorial and Reference

==	Reindent line
[p	Paste but with current indent
%	Match curly bracket
K	Keyword lookup
CTRL-X	Increment the number under the cursor
CTRL-A	Decrement the number under the cursor
K	Do a man on the keyword under the cursor

Program searches

Command	Search For	Start	Result
[d	Macro definition	Start of file	Show line
]d	Macro definition	Current position	Show line
[D	Macro definition	Start of file	Show all
]D	Macro definition	Current position	Show all
[CTRL-D	Macro definition	Start of file	Jump
] CTRL-D	Macro definition	Current position	Jump
CTRL-W d	Macro definition	Start of file	New window
CTRL-W i	Line containing word	Start of file	Open new window
[CTRL-I	Line containing word	Start of file	Jump
] CTRL-I	Line containing word	Current position	Jump
[i	Line containing word	Start of file	Show line
] i	Line containing word	Current position	Show line
[I	Line containing word	Start of file	List all
] I	Line containing word	Current position	List all

[#	Go to the next unmatched #if/#else/#endif
] #	Go to the previous unmatched #if/#else/#endif
[(Go to the previous unmatched ().
] (Go to the next unmatched ().

The Vim Tutorial and Reference

gD Go to global declaration

gd Go to local declaration

Text selection

Command	Delimiter	Include Delimiter	Example
a"	"	Yes	print "Hello world"
a'	'	Yes	ch = 'x';
a(a) ab	[] ()	Yes	x = data[j]; x = f(y);
a< a>	<>	Yes	<bold>Bold text
a{ a} aB	{}	Yes	if (a) { b(c); }
aw	word	Yes	This is an example
aW	WORD	Yes	This word-style example
a[a]	[]	Yes	a = x[3];
a`	`	Yes	x = `ls -1`
ap	Paragraph	Yes	This is a sentence. And this is another. Different paragraph.
as	Sentence	Yes	No way. Yes way.
at	Tag block	Yes	<i>Yell</i>
i"	"	No	print "Hello world"
i'	'	No	ch = 'x';
i(i) ib	[] ()	No	x = data[j]; x = f(y);
i< i>	<>	No	<bold>Bold text
i{ i} iB	{}	No	if (a) { b(c); }

The Vim Tutorial and Reference

Command	Delimiter	Include Delimiter	Example
iw	word	No	This is an example
iw	WORD	No	This word-style example
i[i]	[]	No	a = x[3];
i`	`	No	x = `ls -1`
ip	Paragraph	No	This is a sentence. And this is another. Different paragraph.
is	Sentence	No	No way. Yes way.
it	Tag block	No	<i>Yell</i>

Display options

- :set number** Show line numbers
- :set list** Show unprintable characters
- :set linebreak** Break lines at nice boundaries
- :set breakat** The nice boundaries to use if '**linebreak**' set.
- :set nowrap** Do not wrap the screen

Diff mode

- do** Get (obtain) difference from the other window
- dp** Put difference to the other window
- :diffsplit {file}** Split the window and start diff mode.
- :diffthis** Add this file to the ones being used for this diff.
- :diffoff** Turn off diff mode
- :diffupdate** Update the differences between files
- [number] [c]** Backward change.
- [number]]c** Forward change

Folding

zf{motion}	Fold text
z0	Open all folds under the cursor
zo	Open a single fold under the cursor
zC	Close all folds under the cursor
zc	Close fold under the cursor
zR	Set ' foldlevel ' to it's lowest level
zr	Reduce ' foldlevel ' by one
zi	Invert ' foldenable '
[z	Move to start of fold
] z	Move to end of fold

Misc commands

qa{commands}q	Start macro
@a	Execute macro
q:	Edit : commands
q/	Edit searches
q?	Edit reverse searches

Index

basic.....	28
editing.....	28
Basic.....	28
Movement Commands.....	28
Basic Editing.....	28
Basic Movement Commands.....	28
Command.....	
Delete.....	
Normal Mode.....	28
Esc.....	
Insert Mode.....	31
i.....	
Normal Mode.....	30
Insert.....	
Normal Mode.....	28
x.....	
Normal Mode.....	33
<Enter>.....	
Insert Mode.....	33
<Esc>.....	
Insert Mode.....	31
Command mode.....	
:set.....	
Command.....	28
compatible.....	
Option.....	28
compatible option.....	28
cp.....	
option.....	28
cp option.....	28
CTRL-H.....	
Normal Mode Command.....	32
CTRL-H, Normal Mode Command. .32	
CTRL-J.....	
Normal Mode Command.....	32
CTRL-K.....	
Normal Mode Command.....	32
CTRL-K, Normal Mode Command...32	
CTRL-N.....	
Normal Mode Command.....	32
CTRL-N, Normal Mode Command. .32	
CTRL-P.....	
Normal Mode Command.....	32
CTRL-P, Normal Mode Command..32	
Delete Character.....	
Normal Mode.....	33
Delete Command (Normal Mode)...28	
deleting characters, Normal Mode.	33
Down (j).....	
Normal Mode.....	31
Down (k).....	
Movement Command (Normal mode).....	31
Esc (key) Insert Mode command...31	
Exiting the editor.....	28
Getting Out.....	
Insert Mode.....	31
Getting out of Insert Mode.....	31
gvim command.....	29
Help.....	
Command.....	28
Help Command.....	28
Inseert.....	
Mode.....	30
Insert.....	
Mode.....	30
Insert (i).....	
Normal Mode.....	30
Insert command (i).....	30
Insert Command (Normal Mode)...28	
Insert Mode.....	30
Leavingf Insert Mode.....	31
Left (h).....	
Movement Command (Normal mode).....	31
Normal Mode.....	31
Linux.....	28
Microsoft Windows,.....	29
Mode Visual.....	30
Modes.....	30
Movement.....	
Basic.....	

The Vim Tutorial and Reference

Command.....	28
Normal Mode.....	
Delete.....	
Command.....	28
Insert.....	
Command.....	28
Normal Mode command (h).....	
Left.....	31
Normal Mode command (j).....	
Down.....	31
Normal Mode command (k).....	
Up.....	31
Normal Mode command (l).....	
Right.....	31
Right (l).....	
Movement Command (Normal mode).....	31
Normal Mode.....	31
running vim.....	29
starting vim.....	29
Trouble (Getting out of).....	31
UNIX.....	28
Up (j).....	
Movement Command.....	31
Movement Command (Normal mode).....	31
Up (k).....	
Normal Mode.....	31
Vi mode.....	28
vim.....	
command line command.....	29
Vim.....	
Mode.....	28
vim command.....	29
Vim mode.....	28
Visual.....	
Mode.....	30
Visual Mode.....	30
Windows, Microsoft.....	29
x command, Normal Mode.....	33
x, Normal Mode Command.....	33
:set.....	
Command command.....	28
:set command.....	28
.vimrc.....	
File.....	28
.vimrc file.....	28
<BS>.....	
Normal Mode Command.....	32
<BS>, Normal Mode Command.....	32
<Down>.....	
Normal Mode Command.....	32
<Down>, Normal Mode Command.....	32
<Enter>, Insert Mode.....	33
<Enter>, Insert Moder.....	33
<Esc> Insert Mode command.....	31
<Left>.....	
Normal Mode Command.....	32
<Left>, Normal Mode Command.....	32
<NL>.....	
Normal Mode Command.....	32
<NL>, Normal Mode Command.....	32
<Right>.....	
Normal Mode Command.....	32
<Right>, Normal Mode Command.....	32
<Space>.....	
Normal Mode Command.....	32
<Space>, Normal Mode Command.....	32
<Up>.....	
Normal Mode Command.....	32
<Up>, Normal Mode Command.....	32