VIM-Handbuch

Jens Buttenschön

Version vom 24. November 2025

Inhaltsverzeichnis

Vorwort xvii

Hinweise zur Benutzung xix

1. Grundlagen 1

1.1. VIM-Modi . . . 1
1.1.1. Normal Mode . . . 1
1.1.2. Insert Mode . . . 2
1.1.3. Visual Mode . . . 3
1.1.4. Replace Mode . . . 4
1.1.5. Command-Line Mode . . . 5
1.1.6. Operator-Pending Mode . . . 6
1.1.7. Ex-Mode (selten benötigt) . . . 7
1.2. Operator–Motion–Konzept . . . 7
1.2.1. Aufbau einer Operation . . . 8
1.2.2. Operatoren (Tätigkeiten) . . . 8
1.2.3. Motions (Bewegungen und Ziele) . . . 9
1.2.4. Counts (Wiederholungen) . . . 10
1.2.5. Operator-Pending Mode . . . 11
1.2.6. Aussagekraft des Modells . . . 12
1.3. Basisbewegungen . . . 12
1.3.1. Zeichenweise Bewegung . . . 12
1.3.2. Zeilenanfang und Zeilenende . . . 14
1.3.3. Wortbasierte Bewegungen . . . 15
1.3.4. Bewegungen über mehrere Zeilen . . . 16
1.3.5. Seitenweises Scrollen . . . 17
1.3.6. Direktes Springen in der Datei . . . 18
1.3.7. Klammerbezogene Bewegung . . . 19
1.3.8. Warum Basisbewegungen so wichtig sind . . . 20

2. Eingabemodi **23**
2.1. Insert . . . 23
2.1.1. Einstieg in den Insert-Mode . . . 23
2.1.2. Verhalten im Insert-Mode . . . 24
2.1.3. Einfügen von Steuerzeichen . . . 25
2.1.4. Einfügen aus Registern . . . 26
2.1.5. Insert-Mode in Makros . . . 27
2.2. Append . . . 28
2.2.1. Einstieg in den Append-Modus . . . 28
2.2.2. Typisches Verhalten . . . 28
2.2.3. Wann a sinnvoll ist . . . 29
2.2.4. Wann A die bessere Wahl ist . . . 29
2.2.5. Append in Makros . . . 30
2.3. Neue Zeilen (o, O) . . . 30
2.3.1. Bedeutung von o und O . . . 30
2.3.2. Verhalten und Eigenschaften . . . 31
2.3.3. Praktische Beispiele . . . 31
2.3.4. Einsatz in Makros . . . 32
2.3.5. Unterschiede zu Insert/Append . . . 32
2.4. Replace Mode (r, R) . . . 33
2.4.1. Einzelnes Ersetzen mit r . . . 33
2.4.2. Replace Mode mit R . . . 33
2.4.3. Verhalten am Zeilenende . . . 34
2.4.4. Undo-Verhalten . . . 34
2.4.5. Makrotauglichkeit . . . 34
2.5. Spezialbefehle im Insert Mode . . . 35
2.5.1. Löschbefehle im Insert-Modus . . . 35
2.5.2. Einfügen von Registerinhalten . . . 36
2.5.3. Einfügen von wörtlichen Zeichen . . . 36
2.5.4. Navigieren im Insert-Modus . . . 37
2.5.5. Auto-Completion im Insert-Modus . . . 37
2.5.6. Spezialfunktionen . . . 38
2.5.7. Makrotauglichkeit der Insert-Spezialbefehle . . . 39

3. Navigation **41**
3.1. Wort- und Satznavigation . . . 41
3.1.1. Wortbasierte Navigation (w, e, b) . . . 41

3.1.2. WORD-basierte Navigation (W, E, B) 42
3.1.3. Vergleich von `word` und `WORD` 43
3.1.4. Satznavigation ((und)) 44
3.1.5. Praktische Beispiele 44
3.1.6. Warum diese Navigation so mächtig ist 44
3.2. Absatz- und Dateinavigation 45
3.2.1. Absatznavigation ({ und }) 45
3.2.2. Navigation innerhalb des sichtbaren Bereichs 46
3.2.3. Scrollbasierte Navigation 47
3.2.4. Dateiweite Navigation 48
3.2.5. Warum diese Navigation besonders makrotauglich ist . . . 49
3.3. Scrollbefehle 50
3.3.1. Scrollen mit Cursorbewegung 50
3.3.2. Scrollen ohne Cursorbewegung 51
3.3.3. Ausrichten der aktuellen Zeile im Fenster 52
3.3.4. Scrollen in Verbindung mit relativer Navigation 53
3.3.5. Makrohinweise zum Scrollen 54
3.3.6. Zusammenfassung 54
3.4. Klammernavigation 55
3.4.1. Springen zur passenden Klammer (%) 55
3.4.2. Navigation im Klammerkontext 57
3.4.3. Klammerobjekte (für Operatoren) 58
3.4.4. Klammern in Makros 59
3.4.5. Zusammenfassung 60

4. Editieren (Operatoren) 61
4.1. Löschen (d) 61
4.1.1. Grundprinzip 61
4.1.2. Löschen kompletter Zeilen 62
4.1.3. Löschen mit Textobjekten 63
4.1.4. Löschen mit Counts 64
4.1.5. Löschen bis zu Suchergebnissen 65
4.1.6. Einfügen nach Löschoperationen 66
4.1.7. Makrohinweise für Löschoperationen 67
4.1.8. Zusammenfassung 68
4.2. Ändern (c) 68
4.2.1. Grundprinzip 68

4.2.2. Zeilenbasierte Änderungen 69
4.2.3. Ändern mit Textobjekten 70
4.2.4. Counts mit c 71
4.2.5. Unterschied zwischen cw und ce 72
4.2.6. Warum c makrosensibel ist 73
4.2.7. Beispiele aus der Praxis 73
4.2.8. Zusammenfassung 73
4.3. Kopieren und Einfügen (y, p) 74
4.3.1. Grundprinzip 74
4.3.2. Zeilenweises Kopieren 75
4.3.3. Kopieren mit Textobjekten 75
4.3.4. Einfügen (p und P) 76
4.3.5. Register 77
4.3.6. Suchbasiertes Kopieren 78
4.3.7. Makrohinweise 79
4.3.8. Zusammenfassung 80
4.4. Ersetzen (r, R) 80
4.4.1. Einzelzeichen ersetzen (r) 80
4.4.2. Mehrfaches Ersetzen (r mit Count) 81
4.4.3. Überschreibmodus (Replace Mode, R) 82
4.4.4. Unterschied zwischen c und R 83
4.4.5. Visueller Replace Mode (gR) 84
4.4.6. Makrohinweise 85
4.4.7. Zusammenfassung 86
4.5. Wiederholen (.) 86
4.5.1. Grundprinzip 86
4.5.2. Was . wiederholt — und was nicht 87
4.5.3. Wiederholen von Texteinfügungen 88
4.5.4. Sonderfall: x 89
4.5.5. Wiederholen und Textobjekte 90
4.5.6. Counts und Wiederholen 90
4.5.7. Warum . makrotauglich ist 91
4.5.8. Zusammenfassung 92

5. Textobjekte **93**
5.1. Innere und äußere Objekte (i, a) 93
5.1.1. Grundprinzip 93

5.1.2. Innere Objekte (i) 94
5.1.3. Äußere Objekte (a) 95
5.1.4. Visuelle Auswahl mit Textobjekten 96
5.1.5. Makrohinweise 97
5.2. Wort-, Satz- und Absatzobjekte 97
5.2.1. Wortobjekte 98
5.2.2. Satzobjekte 99
5.2.3. Absatzobjekte 100
5.2.4. Praxisempfehlungen und Makrohinweise 101
5.3. Klammerobjekte 101
5.3.1. Unterstützte Klammerarten 101
5.3.2. Innere Klammerobjekte (i(, i{, i[, i<) 102
5.3.3. Äußere Klammerobjekte (a(, a{, a[, a<) 103
5.3.4. Verschachtelte Strukturen 104
5.3.5. Klammerobjekte im Visual Mode 106
5.3.6. Makrohinweise 106
5.3.7. Zusammenfassung 107

6. Suche und Ersetzen 109
6.1. Einfache Suche 109
6.1.1. Besonderheiten der einfachen Suche 109
6.1.2. Such-Historie 109
6.1.3. Schnelle Wiederholung 110
6.2. Rückwärts- und Vorwärtssuche 110
6.2.1. Treffer-Navigation 110
6.2.2. Operatoren mit richtungsgebundener Suche 111
6.2.3. Verhalten bei Fehltreffern 112
6.2.4. Historie und Wiederholung 112
6.3. Globale Ersetzungen 112
6.3.1. Mehrere Ersetzungen pro Zeile 113
6.3.2. Ersetzungen im gesamten Dokument 113
6.3.3. Interaktive Ersetzungen 113
6.3.4. Bereichsbezogene Ersetzungen 114
6.3.5. Ersetzungen mit Backreferences 114
6.3.6. Ersetzungen in Makros 115
6.4. Ersetzungs-Syntax 116
6.4.1. Adresse 116

6.4.2. Suchmuster 116
6.4.3. Ersatztext 117
6.4.4. Flags 117
6.4.5. Alternative Trennzeichen 118
6.4.6. Ersetzungs-Besonderheiten 118

7. Visual Mode 119
7.1. Zeichenweise Auswahl 119
7.1.1. Aktivieren des Visual Mode 119
7.1.2. Bewegen der Auswahl 120
7.1.3. Anwenden von Operatoren 121
7.1.4. Spezialfall: Wiederholbarkeit 122
7.1.5. Praktische Anwendungsfälle 123
7.1.6. Zusammenfassung 124
7.2. Zeilenweise Auswahl 124
7.2.1. Aktivieren des Visual Line Mode 124
7.2.2. Erweitern der Auswahl 125
7.2.3. Operatoren im Visual Line Mode 126
7.2.4. Beispielszenarien 126
7.2.5. Wiederholbarkeit 127
7.2.6. Makrohinweise 127
7.2.7. Zusammenfassung 127
7.3. Blockweise Auswahl 127
7.3.1. Aktivieren des Blockmodus 127
7.3.2. Erweitern der Auswahl 128
7.3.3. Operatoren im Blockmodus 129
7.3.4. Einfügen in mehreren Zeilen 130
7.3.5. Typische Anwendungsfälle 131
7.3.6. Wiederholbarkeit und Makrohinweise 131
7.3.7. Zusammenfassung 131
7.4. Operatoren im Visual Mode 132
7.4.1. Wiederholen und Makros 133
7.4.2. Zusammenfassung 133

8. Bereiche und Ex-Befehle 135
8.1. Range-Syntax 135
8.1.1. Grundaufbau 135

8.1.2. Wichtige Adressen 136
8.1.3. Relative Adressen 137
8.1.4. Adress-Kombinationen 138
8.1.5. Komma vs. Semikolon 138
8.1.6. Sonderadresse % 140
8.1.7. Range ohne Befehl 141
8.1.8. Fehlertoleranz 142
8.2. Globale Ex-Kommandos (:g) 143
8.2.1. Grundform 143
8.2.2. Einfache Beispiele 143
8.2.3. Negativauswahl 144
8.2.4. Kombination mit Bereichen 144
8.2.5. Kettenbefehle 145
8.2.6. Sondermuster 146
8.2.7. Fehlerquellen 147
8.2.8. Makrotauglichkeit 148
8.3. Filterbefehle (!) 149
8.3.1. Grundprinzip 149
8.3.2. Filter im Visual Mode 150
8.3.3. Filter auf Bereichen 151
8.3.4. Typische Anwendungsfälle 152
8.3.5. Komplexe Workflows 153
8.3.6. Wichtige Hinweise 154
8.4. Ex-Befehlsüberblick 155
8.4.1. Dateioperationen 156
8.4.2. Navigation im Ex-Modus 157
8.4.3. Bearbeitung durch Bereiche 158
8.4.4. Filter und externe Programme 159
8.4.5. Fenster und Buffer 159
8.4.6. Optionen und Status 161
8.4.7. Meta-Kommandos 162
8.4.8. Quickfix-Workflow 163

9. Makros und Register **167**
9.1. Makros aufzeichnen 167
9.1.1. Grundprinzip 167
9.1.2. Makro ausführen 168

9.1.3. Letztes Makro wiederholen 168
9.1.4. Makros und Wiederholung mit 168
9.1.5. Makrotaugliche Regeln 169
9.1.6. Makros auf mehrere Zeilen anwenden 169
9.1.7. Makro prüfen 170
9.2. Makros ausführen 171
9.2.1. Makro in bestimmtem Register ausführen 171
9.2.2. Letztes ausgeführtes Makro wiederholen 172
9.2.3. Makro mehrfach hintereinander ausführen 173
9.2.4. Auf andere Register zugreifen 174
9.2.5. Makro im Register einsehen 174
9.2.6. Makro abbrechen 175
9.3. Registerübersicht 175
9.3.1. Typen von Registern 176
9.3.2. Benannte Register (”a–”z) 177
9.3.3. Numerische Register (”0–”9) 178
9.3.4. Zwischenablage-Register (”+, ”*) 178
9.3.5. Black-Hole-Register (”_) 179
9.3.6. Register anzeigen 180
9.4. Register gezielt nutzen 180
9.4.1. Gezielt in ein Register schreiben 180
9.4.2. Register für Makros verwenden 181
9.4.3. Anhängen an Register 181
9.4.4. Delete-History sauber halten 181
9.4.5. Clipboard präzise steuern 181
9.4.6. Register kombinieren 181
9.4.7. Register inspizieren 182
9.5. Makro-Optimierung 182
9.5.1. Grundprinzipien stabiler Makros 182
9.5.2. Vermeiden instabiler Bewegungsarten 183
9.5.3. Fehlervermeidung durch deterministische Schritte 183
9.5.4. Makros modularisieren 184
9.5.5. Register für Makrotests nutzen 184
9.5.6. Makros mit Textobjekten kombinieren 185
9.5.7. Makro-Debugging 185
9.5.8. Best Practices 186

10. Faltung (Folding) 187
10.1. Faltungen erzeugen . . . 187
10.1.1. Manuelle Faltungen . . . 187
10.1.2. Faltungen über Bereiche . . . 189
10.1.3. Markerbasierte Faltungen . . . 190
10.1.4. Syntaxbasierte Faltungen . . . 193
10.1.5. Einrückungsbasierte Faltungen . . . 194
10.1.6. Manuelle Kontrolle der Faltmethoden . . . 196
10.2. Öffnen und Schließen . . . 198
10.2.1. Grundbefehle . . . 198
10.2.2. Alle Ebenen öffnen und schließen . . . 199
10.2.3. Globale Fold-Operationen . . . 200
10.2.4. Hilfreiche Sichtkontrolle . . . 202
10.2.5. Makrotauglichkeit . . . 203
10.3. Faltungstiefe . . . 204
10.3.1. Foldlevel: zentrale Steuerung . . . 204
10.3.2. Dynamische Steuerung . . . 206
10.3.3. Befehle zum Öffnen/Schließen aller Folds . . . 207
10.3.4. Initiale Faltstruktur beim Laden einer Datei . . . 209
10.3.5. Makros und Faltungstiefe . . . 210
10.4. Faltungen aktualisieren . . . 211
10.4.1. Folds im aktuellen Buffer neu berechnen . . . 211
10.4.2. Fold-Ansicht speichern und laden (`:mkview`/`:loadview`) 212
10.4.3. Alle Folds neu erzeugen . . . 213
10.4.4. Faltungen hart zurücksetzen . . . 214
10.4.5. Faltung nach Änderungen im Visual Mode . . . 216
10.4.6. Automatische Aktualisierung beim Speichern . . . 217
10.5. Faltmethoden . . . 218
10.5.1. Manual (`manual`) . . . 219
10.5.2. Indent (`indent`) . . . 220
10.5.3. Syntax (`syntax`) . . . 222
10.5.4. Expr (`expr`) . . . 223
10.5.5. Marker . . . 226
10.5.6. Diff . . . 227
10.5.7. Foldmethoden wechseln . . . 228

11. Fenster und Splits **231**
11.1. Fenster teilen . . . 231
11.1.1. Horizontale Splits . . . 231
11.1.2. Vertikale Splits . . . 232
11.1.3. Splits beim Öffnen von Dateien . . . 233
11.1.4. Split mit Zeilensprung . . . 235
11.1.5. Splits mit unterschiedlichen Puffern . . . 236
11.1.6. Mehrere Splits kombinieren . . . 237
11.2. Fenster wechseln . . . 238
11.2.1. Wechseln per Tastenkombination . . . 239
11.2.2. Wechseln per Ex-Befehl . . . 240
11.2.3. Wechsel zum zuletzt oder nächsten Fenster . . . 241
11.2.4. Direkter Sprung per Puffer-ID . . . 241
11.2.5. Fenster wechseln mit Maus . . . 243
11.2.6. Fensterwechsel nach Öffnen neuer Splits . . . 244
11.2.7. Fensterliste anzeigen . . . 245
11.3. Fenstergrößen ändern . . . 246
11.3.1. Relative Größenänderungen . . . 246
11.3.2. Absolute Größen setzen . . . 247
11.3.3. Fensterverteilung automatisch ausbalancieren . . . 249
11.3.4. Fenstergrößen sperren . . . 250
11.3.5. Fenstergrößen mit Maus ändern . . . 251
11.4. Buffer in Splits . . . 252
11.4.1. Buffer in neuem Fenster öffnen . . . 252
11.4.2. Zwischen Buffern im gleichen Fenster wechseln . . . 253
11.4.3. Buffer in mehreren Fenstern gleichzeitig . . . 254
11.4.4. Unabhängige Cursorpositionen . . . 255
11.4.5. Bufferliste anzeigen . . . 256
11.4.6. Buffer sauber geschlossen halten . . . 258
11.5. Terminal-Splits . . . 259
11.5.1. Terminal starten . . . 259
11.5.2. Modi im Terminalfenster . . . 259
11.5.3. Terminal schließen . . . 260
11.5.4. Typische Workflows . . . 260
11.5.5. Modi: Terminal-Insert- und Normalmodus . . . 261
11.5.6. Fenstersteuerung . . . 262
11.5.7. Scrollback und Suche . . . 262

11.5.8. Terminal schließen 263

12. Datei und Projekt 265
12.1. Speichern 265
12.1.1. Grundbefehle 265
12.1.2. Speichern trotz Schreibschutz 266
12.1.3. Speichern in nicht existierende Verzeichnisse 267
12.1.4. Automatisches Speichern 268
12.1.5. Wechselnde Dateiformate / Encoding 270
12.1.6. Backup- und Swap-Dateien 271
12.1.7. Speichern und Kommandos kombinieren 273
12.2. Datei öffnen 275
12.2.1. Datei direkt öffnen 275
12.2.2. Relative und absolute Pfade 276
12.2.3. Datei in neuem Fenster öffnen 277
12.2.4. Von Datei zu Datei springen 278
12.2.5. Dateien aus dem Dateibrowser öffnen 279
12.2.6. Nicht vorhandene Dateien öffnen 280
12.2.7. Dateien sicher neu laden 282
12.2.8. Ganze Verzeichnisse öffnen 283
12.3. Beenden 284
12.3.1. Standardbefehle 284
12.3.2. Beenden erzwingen 285
12.3.3. Beenden über ZZ und ZQ 286
12.3.4. Beenden trotz Swap-Dateien 287
12.3.5. Buffer schließen ohne VIM zu verlassen 288
12.3.6. Session speichern und später beenden 289
12.3.7. Terminal-Fenster beenden 291
12.4. Buffer-Management 292
12.4.1. Buffer anzeigen 293
12.4.2. Zwischen Buffern wechseln 294
12.4.3. Buffer schließen 295
12.4.4. Neuen Buffer erzeugen 296
12.4.5. Buffer benennen 297
12.4.6. Buffer speichern 298
12.4.7. Versteckte Buffer (hidden buffers) 299
12.4.8. Buffer gleichzeitig anzeigen 300

12.4.9. Buffer löschen ohne Projektverlust 302
12.5. Tabs (optional) 303
12.5.1. Tabs erstellen 303
12.5.2. Tabs wechseln 304
12.5.3. Tabs schließen 305
12.5.4. Tabs reorganisieren 306
12.5.5. Tabs und Fensterlayouts 307
12.5.6. Tabs automatisch öffnen 308

13. Konfiguration und vimrc 311
13.1. Grundlagen der VIM-Konfiguration 311
13.1.1. Was VIM beim Start lädt 312
13.1.2. vimrc: Speicherort und Varianten 314
13.1.3. Konfigurationsprinzipien 316
13.2. Aufbau einer strukturierten vimrc 319
13.2.1. Header und Grundsetup 319
13.2.2. Konfigurationsblöcke 323
13.2.3. Externe Dateien einbinden 326
13.3. Optionen gezielt konfigurieren 329
13.3.1. Anzeige und Benutzeroberfläche 329
13.3.2. Einrückung und Tabs 332
13.3.3. Suche und Ersetzen 336
13.3.4. Backup, Swap und Undo 340
13.4. Mappings und Abkürzungen 343
13.4.1. Grundlagen von Mappings 343
13.4.2. Leader und LocalLeader 346
13.4.3. Praxisbeispiele für Mappings 349
13.4.4. Abkürzungen (abbreviations) 352
13.5. Autocommands und augroups 355
13.5.1. Autocommand-Grundlagen 355
13.5.2. augroup sauber nutzen 358
13.5.3. Typische Anwendungsfälle 361
13.6. Dateityp-spezifische Konfiguration 364
13.6.1. filetype plugin indent on 364
13.6.2. `ftplugin`-Dateien 366
13.6.3. Indent- und Syntax-Anpassungen 369

13.7. Plugin-Management (optional) 371
13.7.1. Plugins ohne Plugin-Manager 371
13.7.2. Plugin-Manager grob einordnen 374
13.7.3. Konfigurationsstruktur für Plugins 377
13.8. Performance, Diagnose und Debugging 380
13.8.1. Startzeit analysieren 380
13.8.2. Konfigurationsfehler finden 383
13.8.3. Saubere Fallbacks und Bedingungen 387
13.9. Beispiel-vimrc 391
13.9.1. Minimalbeispiel 391
13.9.2. Strukturierte Arbeits-vimrc 393
13.9.3. Übungen zur eigenen `vimrc` 397

Schlussbemerkung **401**

Vorwort

Dieses Handbuch verfolgt ein klares Ziel: die Arbeitslogik von VIM so darzustellen, dass du sie präzise nachvollziehen, im Alltag reproduzieren und belastbar anwenden kannst. Es geht nicht um Schnellanleitungen oder »Tricks«, sondern um ein strukturiertes Verständnis der internen Mechanik: Operatoren, Bewegungen, Textobjekte, Makros, Register, Faltungen, Splits sowie projektbezogene Arbeitsweisen.

Mein eigener Zugang zu `vi` begann im Wintersemester 1988/89 an der Fachhochschule Emden. Prof. Dr. Schmidtmann hat uns den Editor nicht als notwendiges Übel, sondern als präzises Werkzeug vermittelt – mit Geduld, Humor und einem klaren Blick für saubere Arbeitsweisen. Ohne diesen frühen, didaktisch sauberen Einstieg hätte ich VIM später vermutlich nicht in dieser Tiefe in meinen Arbeitsalltag integriert.

Einer seiner Sätze ist bis heute hängengeblieben: Eines Tages kam er ins Labor, schmunzelte und meinte nur: »Da hat gerade einer `/tmp` gemountet – auf eine Diskette.« Die meisten haben nicht verstanden, warum das so komisch ist; wer sich mit UNIX-Strukturen auskannte, schon. Diese Art von stillem Augenzwinkern für saubere, durchdachte Arbeitsweisen prägt auch den Anspruch dieses Handbuchs.

Alle Inhalte in diesem Handbuch sind praktisch erprobt und iterativ überarbeitet worden. Konstruktionen, die sich als unklar, fehleranfällig oder unpraktisch erwiesen haben, wurden entfernt oder angepasst. Jede Technik, die du hier findest, ist so beschrieben, dass sie in realen Arbeitsumgebungen tragfähig ist und sich reproduzierbar einsetzen lässt.

Wenn du VIM nicht nur bedienen, sondern beherrschen möchtest, findest du hier einen methodischen Zugang, der die Komplexität des Editors greifbar macht: Du kannst die Kapitel nacheinander durcharbeiten oder gezielt in Themen wie Makros, Register oder Visual Mode einsteigen und die Beispiele unmittelbar ausprobieren.

Dieses Handbuch ist Prof. Dr. Schmidtmann gewidmet, dessen Art, `vi` zu lehren, die Grundlage für alles gelegt hat, was hier systematisiert und weitergeführt wird.

Hinweise zur Benutzung

Dieses Handbuch ist so aufgebaut, dass du jedes Kapitel und jede Section grundsätzlich auch isoliert lesen kannst. Viele Beispiele kannst du direkt in VIM ausprobieren; sie sind bewusst praxisnah formuliert und verzichten auf unnötige Abstraktion.

Tasten und Kommandos werden einheitlich dargestellt:

- einzelne Tasten wie **d**, **w**, **Esc** oder **Ctrl**-**v** erscheinen als Keycaps; aufeinanderfolgende Tastenfolgen werden immer mit getrennt geschrieben, z. B. **d** **w** oder **3** **d** **d**.
- Ex-Befehle werden mit Doppelpunkt als `:write`, `:ls` oder `:quit` gesetzt.
- längere Kommandos, Mappings oder Konfigurationsblöcke erscheinen in `verbatim`- oder Code-Umgebungen, damit du sie ohne Umdeutung übernehmen kannst.

Zur Lektüre kannst du zwei Wege wählen:

- Wenn du neu in VIM bist, lies die Kapitel in der vorgesehenen Reihenfolge. So bekommst du das Zusammenspiel aus Operatoren, Bewegungen und Textobjekten als zusammenhängendes Modell.
- Wenn du bestimmte Themen gezielt nachschlagen willst (z. B. Makros, Register, Visual Mode, Faltungen oder Buffer-Management), kannst du die entsprechenden Kapitel direkt ansteuern. Die Beispiele sind so geschrieben, dass keine zwingenden Querverweise nötig sind.

Verweise auf andere Abschnitte werden im Text nach dem Muster »Abschnitt 1.2.4 (Seite 12)« gesetzt. In der PDF-Version sind diese Angaben anklickbar, sodass du bei Bedarf schnell zu den ausführlicheren Erklärungen springen und mit den Navigationspfeilen deines PDF-Viewers wieder zurückspringen kannst.

Das gesamte Dokument ist so angelegt, dass Änderungen und Erweiterungen leicht nachgezogen werden können. Struktur und Darstellung der Kommandos bleiben dabei konsistent, damit du dich auch in späteren Versionen schnell orientieren kannst.

1. Grundlagen

1.1. VIM-Modi

VIM arbeitet nicht wie klassische Texteditoren. Stattdessen besitzt er unterschiedliche Modi, die jeweils eine eigene Bedeutung haben. Das Verständnis dieser Modi ist grundlegende Voraussetzung für effizientes Arbeiten.

1.1.1. Normal Mode

Der Normal Mode ist der Ausgangszustand von VIM. Nach dem Start von VIM und nach dem Verlassen von Einfüge-Modi befindet man sich immer in diesem Modus. In ihm werden Navigation, Operatoren und Befehlsfolgen ausgeführt; Text wird *nicht* direkt eingegeben.

- **Aufruf:** **Esc** oder **Ctrl** - **[**.
- **Zweck:** Navigation, Operatoren, Kombinieren von Befehlen.
- **Typisches Verhalten:** Nahezu jede Taste ist ein Befehl (Bewegung, Operator, Ex-Kommando, Makro).

Der Normal Mode ist damit der „Steuerraum“ von VIM: Hier werden Bewegungen, Operatoren und Ex-Befehle komponiert. Das bewusste Trennen von *Steuern* (Normal Mode) und *Schreiben* (Insert Mode) ist der Kern des VIM-Bedienkonzepts.

Übungen

1. Öffne eine beliebige Textdatei in VIM. Wechsle mehrfach zwischen Normal- und Insert-Mode:
   - **Esc** für den Wechsel in den Normal Mode
   - **i**, **a**, **o** für den Wechsel in den Insert Mode

   Beobachte in der Statuszeile, wann VIM im Insert Mode ist.
2. Navigiere im Normal Mode für einige Minuten nur mit **h**, **j**, **k**, **l** durch den Text, ohne Pfeiltasten zu benutzen. Ziel: Der Cursor soll sich *intuitiv* anfühlen, ohne dass du über die Tasten nachdenken musst.

3. Drücke im Normal Mode bewusst ein paar Buchstaben (z. B. a s d f) und beobachte, dass der Text nicht verändert wird. Beende VIM anschließend mit `:q!`, um die Datei ohne Speichern zu schließen.

1.1.2. Insert Mode

Im Insert Mode wird Text direkt eingegeben. Das Verhalten ähnelt klassischen Editoren: Die Tastatur erzeugt Zeichen im Dokument. VIM bleibt aber weiterhin *modal* – der Insert Mode ist nur *eine* von mehreren Arbeitsflächen.

- **Aufruf (häufigste Varianten):** i, I, a, A, o, O, sowie Operatoren wie c…
- **Rückkehr in den Normal Mode:** Esc
- **Zweck:** Freies Schreiben, Einfügen und Überschreiben von Text.

Die wichtigsten Einstiegskommandos in den Insert Mode:

- i — Insert *vor* dem aktuellen Zeichen.
- a — Append *nach* dem aktuellen Zeichen.
- I — Insert am *Zeilenanfang*.
- A — Append am *Zeilenende*.
- o — Neue Zeile *unterhalb* einfügen und dort in den Insert Mode wechseln.
- O — Neue Zeile *oberhalb* einfügen und dort in den Insert Mode wechseln.
- c… — „Change“: Inhalt gemäß Motion oder Textobjekt löschen und direkt in den Insert Mode wechseln (z. B. c w, c i ().

Typisch für VIM ist, dass man nur gezielt in den Insert Mode geht, um einen Block Text zu schreiben, und anschließend mit Esc in den Normal Mode zurückkehrt, um weiter zu navigieren oder zu bearbeiten.

Übungen

1. Öffne eine Textdatei und probiere nacheinander:
   - i in der Mitte einer Zeile,
   - a direkt danach,
   - I am Anfang einer Zeile,
   - A am Ende einer Zeile.

Beobachte genau, *wo* der Cursor steht, bevor du tippst. Verlasse den Insert Mode jeweils mit **Esc**.

2. Erzeuge mit **o** und **O** mehrere neue Zeilen über und unter der aktuellen Zeile und schreibe kurzen Beispieltext hinein. Navigiere zwischendurch bewusst im Normal Mode (**h** **j** **k** **l**).
3. Wähle ein Wort in einer Zeile und ersetze es mit **c** **w** durch ein anderes Wort. Wiederhole das Gleiche mit **c** **i** **w**. Achte auf den Unterschied im Verhalten.

1.1.3. Visual Mode

Der Visual Mode dient zur Markierung zusammenhängender Textbereiche. Die Auswahl kann zeichenweise, zeilenweise oder blockweise (rechteckig) erfolgen. Befehle wie **d**, **y** oder **c** wirken im Visual Mode nicht mehr nur auf die aktuelle Cursorposition, sondern auf die gesamte markierte Region.

- **Aktivieren:**
  v (zeichenweise)
  V (zeilenweise)
  Ctrl-**v** (blockweise / Visual Block Mode).
- **Beenden:** **Esc** kehrt in den Normal Mode zurück und hebt die Markierung auf.
- **Zweck:** Selektieren von Bereichen, um anschließende Operationen wie Löschen (**d**), Kopieren (**y**) oder Ersetzen (**c**) gezielt auf die markierte Auswahl anzuwenden.

Übungen

1. **Wort löschen mit Visual Mode**
   Bewege den Cursor auf den Anfang eines Wortes und markiere es vollständig mit **v** und Bewegungstasten (z. B. **w**). Lösche anschließend die Markierung mit **d**.
2. **Mehrere Zeilen verschieben**
   Markiere zehn komplette Zeilen im Visual Line Mode und verschiebe sie.
   Variante 1 (schrittweise – unpraktisch):
   V **j** **j** **j** **j** **j** **j** **j** **j** **j** **j** *(Visual Line Mode aktivieren und*

[j] *zehn Mal drücken)*

Variante 2 (effizient – mit Zählwert):
[V][1][0][j] *(Visual Line Mode aktivieren und mit* [1][0][j] *sofort zehn Zeilen markieren)*

Hinweis: Statt [j] *können Sie ebenso* [k] *verwenden, um die Markierung in die Gegenrichtung (nach oben) zu erweitern.*

Schneide die markierten Zeilen mit [d] aus und füge sie an anderer Stelle mit [p] wieder ein.

Weitere Beispiele und eine ausführlichere Erklärung der Zählwerte finden Sie in Abschnitt **??** (Seite **??**).

3. **Blockweise Spalte editieren**
   Markiere in mehreren Zeilen eine Spalte mit [Ctrl]-[v] und den Bewegungstasten. Füge am Anfang der markierten Spalte Text ein: [I], gewünschten Text tippen, dann [Esc] drücken und beobachten, dass der Text in allen markierten Zeilen eingefügt wird.

1.1.4. Replace Mode

Der Replace Mode ist eine Variante des Insert Mode: Neueingaben überschreiben vorhandene Zeichen, statt sie zu verschieben. Er eignet sich, wenn bereits vorhandener Text in gleicher Breite angepasst werden soll.

- **Aktivieren:** [R] schaltet in den Replace Mode, bis du mit [Esc] in den Normal Mode zurückkehrst.
- **Einzelzeichen ersetzen:** [r] gefolgt von einem Zeichen ersetzt nur das aktuelle Zeichen unter dem Cursor durch dieses neue Zeichen.
- **Zweck:** Schnelles Überschreiben von Text, ohne den restlichen Zeilenaufbau zu verändern.

Übungen

1. **Zahl im Text korrigieren**
   Navigiere auf eine Zahl (z. B. 2024) und ersetze die letzten beiden Ziffern mit [R] durch 25. Beende den Replace Mode mit [Esc] und kontrolliere, dass die Zeile nicht länger geworden ist.
2. **Tippfehler im Wort korrigieren**

Setze den Cursor auf einen falschen Buchstaben in einem Wort und ersetze ihn mit **r** und dem korrekten Zeichen, ohne in den Insert Mode zu wechseln.

3. **Kurze Sequenz überschreiben**
   Stelle den Cursor an den Anfang eines kurzen Wortes und überschreibe das gesamte Wort mit **R** und neuer Eingabe. Brich den Replace Mode anschließend mit **Esc** ab.

1.1.5. Command-Line Mode

Der Command-Line Mode erlaubt Befehle, die am unteren Bildschirmrand in einer Befehlszeile eingegeben werden. Er wird vor allem für Suchen, Ersetzungen, Dateikommandos und globale Operationen genutzt.

- **Aktivieren:** **:**, **/** oder **?** öffnen die Befehlszeile:
  - **:** für Ex-Befehle (z. B. `:w`, `:q`, `:substitute`)
  - **/** für vorwärts gerichtete Suche
  - **?** für rückwärts gerichtete Suche
- **Bestätigen/Abbrechen:** Ein Befehl wird mit **Enter** ausgeführt. Mit **Esc** oder **Ctrl**-**c** brichst du die Eingabe ab, ohne etwas auszuführen.
- **Zweck:**
  - Dateien speichern und schließen (`:w`, `:q`, `:wq`)
  - Suchen und Ersetzen im gesamten Puffer
  - Globale Operationen und komplexere Ex-Befehle

Übungen

1. **Speichern und Beenden**
   Öffne eine Testdatei, nimm eine kleine Änderung vor und speichere mit `:w`**Enter**. Beende Vim anschließend mit `:q`**Enter**. Wiederhole den Ablauf und nutze diesmal direkt `:wq`**Enter**.
2. **Einfache Suche**
   Füge in einer Datei mehrmals das Wort `Test` ein. Suche mit **/** `Test` **Enter** nach dem nächsten Vorkommen und springe mit **n** weiter durch die Treffer. Nutze **N**, um in die Gegenrichtung zu springen.
3. **Globale Ersetzung**

Ersetze in einer Datei alle Vorkommen von `foo` durch `bar` mit: `:\%s/foo/bar/g` **Enter**. Kontrolliere, dass alle `foo`-Vorkommen im Puffer ersetzt wurden.

1.1.6. Operator-Pending Mode

Der Operator-Pending Mode entsteht, wenn ein Operator eingegeben wurde und Vim auf eine Bewegung oder ein Textobjekt wartet. Er ist die Grundlage des „Operator–Motion"-Konzepts: Zuerst kommt der Operator, dann die Bewegung bzw. das Textobjekt.

- **Auslösen:** Sobald du einen Operator wie **d**, **c** oder **y** eingibst, wechselt Vim in den Operator-Pending Mode und wartet auf eine Bewegung oder ein Textobjekt.
- **Beispiele:**
  - **d** **w** – löscht vom Cursor bis zum Wortende
  - **d** **$** – löscht vom Cursor bis zum Zeilenende
  - **d** **i** **(** – löscht den Inhalt innerhalb der nächsten runden Klammern
  - **c** **i** **"** – ändert den Inhalt innerhalb der nächsten Anführungszeichen
- **Zweck:** Operator und Bewegung werden zu einer präzisen Operation verknüpft. Statt vieler Spezialkommandos gibt es ein einheitliches Schema: „Was tun?" (Operator) + „Worauf?" (Bewegung oder Textobjekt).

Übungen

1. **Wortbasiertes Löschen**
   Schreibe eine Zeile mit mehreren Wörtern. Stelle den Cursor auf den Beginn eines Wortes in der Mitte der Zeile und führe nacheinander **d** **w**, **d** **e** und **d** **$** aus. Beobachte, wie sich der Löschbereich je nach Bewegung ändert.
2. **Arbeiten mit Textobjekten**
   Erzeuge eine Zeile wie: `if (x + y) { do_something(); }` Stelle den Cursor irgendwo zwischen die runden Klammern und probiere **d** **i** **(** und **c** **i** **(** aus. Wiederhole das Spiel mit Anführungszeichen (**d** **i** **"** usw.), sofern vorhanden.
3. **Eigene Kombinationen testen**

Nimm dir einen Beispieltext und kombiniere verschiedene Operatoren (d, c, y) mit Bewegungen (w, b, $, 0) und Textobjekten (i (, i p, a w). Notiere dir 2–3 Kombinationen, die du besonders nützlich findest.

1.1.7. Ex-Mode (selten benötigt)

Der Ex-Mode ist ein permanenter, zeilenorientierter Kommandomodus. Er stammt historisch aus `ex` und wird in modernem VIM nur noch selten verwendet. Statt einzelner `:...`-Befehle in der Statuszeile arbeitet man im Ex-Mode Zeile für Zeile nur mit Befehlen.

- Aufruf: Q
- Verlassen: `:visual` (oder Wechsel zurück in einen Normal/Visual Mode)
- Zweck: Historischer Modus, heute fast nur für Spezialfälle oder Skripting-Umgebungen relevant.

Übungen

- Wechsle testweise in den Ex-Mode mit Q, gib ein bis zwei einfache Befehle (z. B. `:set number`, `:write`) ein und kehre mit `:visual` wieder in den normalen Arbeitsmodus zurück.
- Überlege dir, in welchen Situationen du `:-Befehle` im normalen Command-Line Mode (:) bevorzugst und warum der Ex-Mode dafür selten wirklich nötig ist.

1.2. Operator–Motion–Konzept

Das Operator–Motion–Konzept ist das Fundament des effizienten Arbeitens in VIM. Jede komplexere Aktion basiert auf einer Kombination aus einem *Operator* (also einer Tätigkeit wie Löschen oder Ändern) und einer *Motion* (also einer Bewegung oder einem Textobjekt, das den Zielbereich definiert).

In klassischen Editoren selektiert man zuerst mit der Maus und entscheidet dann, was geschehen soll. VIM kehrt dieses Modell um: Zuerst kommt der Operator, dann die Motion. Merksatz: „Was tun?“ (Operator) + „Worauf?“ (Motion). So entstehen präzise, reproduzierbare Operationen ohne Umwege.

1.2.1. Aufbau einer Operation

Eine vollständige Operator–Motion-Sequenz hat immer im Kern die Form
„Operator“ + „Motion“.

Typische Beispiele:

- d w – Löscht vom Cursor bis zum Anfang des nächsten Wortes.
- d $ – Löscht vom Cursor bis zum Zeilenende.
- c i (– Ändert den *Inhalt* der nächsten runden Klammer (...), die Klammern selbst bleiben erhalten.
- y a p – Kopiert den aktuellen Absatz („a paragraph“) in den Default-Registerinhalt.

Optional können Ziffern als Zähler dazukommen, etwa 3 d w oder d 3 w, um die Operation mehrfach zu wiederholen.

Übungen

- Lege eine Testdatei mit mehreren Wörtern pro Zeile an und probiere systematisch d w, d 2 w, d 3 w und d $ aus. Beobachte genau, welcher Textbereich jeweils gelöscht wird.
- Erzeuge ein paar Zeilen mit Klammerausdrücken wie (foo), (bar baz) und teste den Unterschied zwischen c i ((Inhalt ändern) und d a ((alles inkl. Klammern löschen).
- Schreibe zwei bis drei Absätze normalen Textes und verwende y a p, um ganze Absätze zu kopieren und mit p an anderen Stellen wieder einzufügen. Achte darauf, wie VIM den Absatzbereich definiert.

1.2.2. Operatoren (Tätigkeiten)

Operatoren definieren, *was* getan werden soll. Sie wirken immer auf den Bereich, den eine Motion oder ein Textobjekt festlegt.

Typische Operatoren sind:

- d – löschen
- c – ändern (löschen + Wechsel in den Insert Mode)
- y – kopieren („yank“)
- g U – in Großbuchstaben umwandeln

- g u – in Kleinbuchstaben umwandeln
- > – einrücken
- < – ausrücken

Viele Operatoren haben eine „Doppelvariante“, die ohne explizite Motion auf die gesamte Zeile wirkt:

- d d – löscht die gesamte Zeile
- c c – ändert die gesamte Zeile
- y y – kopiert die gesamte Zeile

Übungen

- Erzeuge einige Testzeilen und probiere d d, c c und y y aus. Achte darauf, wie immer die komplette Zeile betroffen ist, unabhängig von der Cursorposition.
- Schreibe eine Zeile mit gemischter Groß-/Kleinschreibung und teste g U w und g u w auf unterschiedlichen Wörtern. Beobachte, wie sich nur das jeweils betroffene Wort ändert.
- Lege einen kleinen Codeblock mit Einrückungen an und verwende > und < zusammen mit Wort- oder Zeilenbewegungen, z. B. > G oder > a p, um Einrückungen gezielt zu verändern.

1.2.3. Motions (Bewegungen und Ziele)

Motions definieren, *wohin* eine Operation wirkt. Sie bestimmen also den Zielbereich für den zuvor gewählten Operator.

Typische Motions sind:

- Wortbewegungen: w, e, b
- Zeilenende: $
- Satz- / Absatzbewegungen:) (nächster Satz), ((vorheriger Satz), } (nächster Absatz), { (vorheriger Absatz)
- Klammernavigation: % (zur passenden Klammer springen)
- Textobjekte: i w, a w, i (, a {

Die Motion legt den Zielbereich exakt fest. Aus **d** (Operator) und **w** (Motion) wird **d** **w**: „lösche bis zum Beginn des nächsten Wortes“. Genauso ergibt **d** **$** „lösche bis zum Zeilenende“.

Übungen

- Schreibe eine Beispielzeile mit mehreren Wörtern und bewege den Cursor nur mit **w**, **e** und **b**. Notiere dir, wie sich diese drei Bewegungen unterscheiden.
- Nutze in einem längeren Text **)** und **}** und beobachte, wann VIM einen „Satz“ und wann einen „Absatz“ erkennt.
- Kombiniere Operatoren mit Textobjekten, z. B. **d** **i** **w**, **d** **a** **w**, **d** **i** **(**, **d** **a** **{** und analysiere, welche Teile des Textes jeweils betroffen sind.

1.2.4. Counts (Wiederholungen)

Nahezu jede Operation in VIM kann mit einem *Count* (Zahl) kombiniert werden. Der Count gibt an, wie oft eine Motion oder Operation ausgeführt werden soll.

- **3** **d** **w** löscht die nächsten drei Wörter
- **2** **d** **a** **{** löscht zwei aufeinanderfolgende Blockobjekte
- **5** **j** bewegt den Cursor fünf Zeilen nach unten

Counts können sowohl vor dem Operator als auch vor der Motion stehen:

- **3** **d** **w** löscht drei Wörter
- **d** **3** **w** löscht ebenfalls drei Wörter

Wichtig ist: VIM kombiniert Count, Operator und Motion zu einer einzigen, präzisen Aktion. Ein zusätzlicher Count vor einem reinen Bewegungsbefehl (**1** **0** **j**, **7** **w**) ändert nur die Bewegung, ohne etwas zu löschen oder zu ändern.

Übungen

- Öffne eine Testdatei mit mehreren längeren Zeilen und bewege dich ausschließlich mit **j** / **k** und Counts wie **5** **j**, **1** **0** **k**. Beobachte, wie schnell du damit große Bereiche überfliegst.

- Nutze 3 d w, 2 d e und d 3 w in einer Beispielzeile und vergleiche das Ergebnis. Prüfe bewusst, ob sich 3 d w und d 3 w in deinem Fall identisch verhalten.
- Teste 2 d a { und 3 d d in einem Code-Beispiel mit mehreren Blöcken oder Zeilen und notiere dir, welche Struktur VIM jeweils als Einheit behandelt.

1.2.5. Operator-Pending Mode

Der Operator-Pending Mode entsteht, sobald ein Operator eingegeben wurde (z. B. d), VIM aber noch auf eine Motion oder ein Textobjekt wartet. In diesem Zwischenzustand ist klar, *was* passieren soll, aber noch nicht *wo*.

- Aktivierung: durch Operatoren wie d, c, y, g U, g u, >, < usw.
- Erwartet: eine Motion oder ein Textobjekt, z. B. w, $, i w, a p.
- Abbruch: Esc bricht den Operator-Pending Mode ab, ohne etwas zu ändern.
- Vorteil: präzise Operationen ohne Maus oder visuelle Selektion.

Typische Kette:

- d w löscht bis zum Anfang des nächsten Wortes
- d $ löscht bis zum Zeilenende
- c i w ändert das aktuelle Wort („inner word“)

Counts lassen sich mit dem Operator-Pending Mode kombinieren:

- 3 d w löscht die nächsten drei Wörter
- d 3 w identisches Ergebnis, Count steht bei der Motion

Übungen

- Tippe nacheinander d w, d $ und c i w in einer Testzeile. Achte bewusst darauf, dass VIM nach dem Operator kurz auf deine Entscheidung „wartet“.
- Brich den Operator-Pending Mode gezielt ab: gib d ein und drücke danach Esc. Überzeuge dich, dass nichts gelöscht wurde.
- Teste Counts im Operator-Pending Mode: 3 d w und d 3 w in einem Satz mit vielen Wörtern. Prüfe, ob beide Varianten dasselbe Ergebnis liefern.

1.2.6. Aussagekraft des Modells

Das Operator–Motion–Konzept ist der Kern der Effizienz von VIM. Statt für jede Einzelfunktion einen eigenen Befehl zu definieren, beantwortet VIM systematisch zwei Fragen:

- „Was tun?“ – der Operator
- „Worauf?“ – die Motion oder das Textobjekt

Aus wenigen Operatoren und Bewegungen entsteht so eine große Anzahl präziser Befehle. Diese Modularität ist kein Nebeneffekt, sondern das zentrale Designprinzip von VIM: Wer Operatoren und Motions sicher kombinieren kann, skaliert seine Fähigkeiten, ohne immer neue Spezialbefehle lernen zu müssen.

Übungen

- Notiere dir fünf Operator–Motion-Kombinationen, die du im Alltag bereits verwendest (z. B. **d** **w**, **c** **i** **(**, **y** **a** **p**) und benenne jeweils klar „Was?“ und „Worauf?“.
- Überlege dir eine typische Maus-Aktion (z. B. „Wort markieren und löschen“) und übersetze sie bewusst in eine Operator–Motion-Sequenz in VIM.
- Variiere gezielt nur den Operator oder nur die Motion: Teste etwa **d**+**w**, **y**+**w**, **g** **U**+**w** auf derselben Textstelle und beobachte, wie sich das Ergebnis ändert.

1.3. Basisbewegungen

Basisbewegungen sind die fundamentalen Navigationsbefehle im Normal Mode. Sie bilden die Grundlage für alle Operator–Motion-Kombinationen und sind unverzichtbar für effizientes Arbeiten. Viele komplexere Bewegungen in VIM bauen direkt auf diesen einfachen Kommandos auf und erweitern sie nur um größere Sprünge (Wörter, Sätze, Absätze, Dateien).

Wer die Basisbewegungen sicher beherrscht, kann den Cursor präzise steuern und Operatoren wie **d**, **c** oder **y** gezielt damit kombinieren.

1.3.1. Zeichenweise Bewegung

Diese Bewegungen verschieben den Cursor um jeweils ein Zeichen bzw. eine Zeile:

- **h** ein Zeichen nach links

- l ein Zeichen nach rechts
- j eine Zeile nach unten
- k eine Zeile nach oben

Diese vier Tasten bilden die grundlegende Navigationsmatrix. Sie sind entscheidend, weil Operatoren wie d, c oder y direkt mit ihnen kombiniert werden können (z. B. d h, d j).

Übungen

- Bewege den Cursor in einer beliebigen Datei ausschließlich mit h, j, k, l und verzichte bewusst auf die Pfeiltasten. Beobachte, wie sich deine Fingermotorik anpasst.
- Wende einfache Operator–Motion-Kombinationen mit Basisbewegungen an: d h, d l, d j, d k. Notiere dir in einem Kommentar, was jede Kombination genau tut.
- Teste Counts mit Basisbewegungen: 5 j, 3 k. Übertrage das anschließend auf Operatoren, etwa 3 d j oder 2 d k, und prüfe gezielt, welche Bereiche jeweils betroffen sind.

Alternative: Pfeiltasten

VIM unterstützt dieselben Bewegungen ebenfalls über die vier Pfeiltasten der Tastatur. Funktional sind die Pfeiltasten mit h, j, k, l identisch – jede Bewegung lässt sich dadurch auf zwei Arten ausführen.

Es gibt jedoch Unterschiede, die in der Praxis relevant werden:

- Die Pfeiltasten erzeugen andere Keycodes als h, j, k, l.
- Bei der Aufzeichnung von Makros werden Pfeiltasten als Terminal-Escape-Sequenzen gespeichert, was Makros weniger robust und schlechter lesbar macht.
- In minimalen Umgebungen (z. B. SSH, Rescue-Shells, serielle Konsolen) funktionieren die Pfeiltasten nicht immer zuverlässig.
- h, j, k, l sind auf allen Tastaturlayouts verfügbar und verhalten sich international identisch.

Aus diesen Gründen bevorzugen erfahrene VIM-Nutzer die HJKL-Navigation, insbesondere für Makros und automatisierbare Arbeitsabläufe. Funktional sind beide

Varianten gleichwertig – in der Praxis unterscheiden sie sich vor allem hinsichtlich Stabilität und Kompatibilität.

Übungen

- Navigiere fünf Minuten lang nur mit den Pfeiltasten, anschließend fünf Minuten nur mit h, j, k, l. Notiere, welche Variante sich schneller und präziser anfühlt.
- Zeichne ein kurzes Makro einmal mit Pfeiltasten und einmal mit h, j, k, l auf. Schau dir mit `:reg` an, wie die Sequenzen im Register gespeichert wurden.
- Arbeite in einem SSH- oder TTY-Setup bewusst ohne Pfeiltasten und gewöhne dir an, ausschließlich mit HJKL zu navigieren.

1.3.2. Zeilenanfang und Zeilenende

Für viele Operationen ist es entscheidend, ob der Cursor wirklich am absoluten Zeilenanfang steht oder nur am ersten sichtbaren Zeichen (also hinter der Einrückung). VIM unterscheidet hier sehr genau:

- 0 springt an den absoluten Zeilenanfang (Spalte 1), unabhängig von Leer- oder Tab-Zeichen.
- ^ springt auf das erste Nicht-Leerzeichen der Zeile, typische Position für Code-Einrückungen.
- $ springt an das Zeilenende (letztes sichtbares Zeichen).

Diese Bewegungen werden fast immer mit Operatoren kombiniert, um Zeilenbereiche schnell zu bearbeiten:

- d $ löscht vom aktuellen Cursor bis zum Zeilenende.
- c 0 ändert vom aktuellen Cursor bis zum absoluten Zeilenanfang (inkl. führender Leerzeichen).
- y ^ kopiert vom aktuellen Cursor bis zum ersten Nicht-Leerzeichen der Zeile.

Merke dir die Semantik: 0 kümmert sich um die komplette Zeile inklusive Einrückung, ^ respektiert die logische Einrückung (wichtig bei Code), $ kapselt das „rechte Ende“ eines Bereichs.

Übungen

- Erzeuge mehrere unterschiedlich eingerückte Codezeilen und beobachte den Unterschied zwischen **0** und **^**. Notiere dir, wann du welche Variante bevorzugst.
- Teste folgende Kombinationen und prüfe das Ergebnis jeweils mit **u**: **d** **0**, **d** **^** und **d** **$**. Achte darauf, welche Teile der Zeile genau verschwinden.
- Nutze **c** **0** und **c** **^** in einer bestehenden Datei, um Zeilenanfänge umzustrukturieren (z. B. Einrückung ändern oder Präfixe wie **/** **/** entfernen).
- Kombiniere Counts mit **$**, etwa **3** **d** **$**, um das Zeilenende über mehrere Zeilen hinweg zu bearbeiten. Prüfe, wie sich das von **d** **3** **j** + **d** **$** unterscheidet.

1.3.3. Wortbasierte Bewegungen

VIM unterscheidet zwei Ebenen von „Wörtern“, die auf unterschiedlicher Zerlegung der Zeile beruhen:

- *word*: eine zusammenhängende Folge von Zeichen, die zur Menge der „Wortzeichen“ gehören (standardmäßig Buchstaben, Ziffern und Unterstrich). Welche Zeichen dazu zählen, wird über die Option `iskeyword` gesteuert. Alles andere (Leerzeichen, Interpunktion, Operatoren, Klammern ...) trennt *words* voneinander.
- *WORD*: eine zusammenhängende Folge von beliebigen Nicht-Leerzeichen. Erst ein Leerzeichen oder Zeilenumbruch beendet das *WORD*; Interpunktion und Sonderzeichen gehören also mit zum Block.

Damit gilt: Kleinbuchstaben (*word*-Ebene) arbeiten feiner auf Bezeichnern und ähnlichen Einheiten, Großbuchstaben (*WORD*-Ebene) arbeiten gröber auf ganzen Blöcken zwischen Leerzeichen.

Bewegungen auf word-Ebene

- **w** springt zum Anfang des nächsten *word*
- **e** springt zum Ende des aktuellen bzw. nächsten *word*
- **b** springt zum Anfang des vorherigen *word*

Bewegungen auf WORD-Ebene

- **W** springt zum Anfang des nächsten *WORD*

- E springt zum Ende des aktuellen bzw. nächsten *WORD*
- B springt zum Anfang des vorherigen *WORD*

Pragmatische Daumenregel:

- *word*-Ebene (w, e, b) für Text und Code, bei denen Interpunktion und Operatoren eigene Sprungziele sein sollen.
- *WORD*-Ebene (W, E, B) für Kommandozeilen, Pfade und Argumentlisten, bei denen alles bis zum nächsten Leerzeichen als ein Block behandelt werden soll.

Übungen

- Lege eine Zeile wie `printf("Fehler: Code = 42; Pfad=/var/log/app.log");` an, setze den Cursor an den Zeilenanfang und bewege dich nacheinander mit w, e, b darüber. Wiederhole dasselbe mit W, E, B und vergleiche die Sprungpunkte: Wo hält VIM an Satzzeichen, wo springt es darüber hinweg?
- Wiederhole dieselbe Übung mit einer Kommandozeile wie `rm -rf /var/log/app.log,backup.tar.gz` und beobachte den Unterschied zwischen w/e/b und W/E/B.
- Nutze Counts: teste 3 w, 2 b und 4 W auf einer längeren, gemischten Kommandozeile (Optionen, Pfade, Flags) und notiere dir typische Sprungabstände, die du praktisch findest.
- Erzeuge eine Liste mit Dateinamen und Pfaden (z. B. aus einem `:ls` oder `:args`) und benutze gezielt E und B, um schnell an Dateiendungen und Verzeichnisgrenzen zu springen. Versuche, ohne Pfeiltasten auszukommen.

1.3.4. Bewegungen über mehrere Zeilen

Neben der zeichenweisen Navigation mit j und k bietet VIM einige Bewegungen, die speziell für den Zeilenwechsel gedacht sind und automatisch auf das erste Nichtleerzeichen springen.

- + springt in die nächste Zeile auf das erste Nichtleerzeichen
- - springt in die vorherige Zeile auf das erste Nichtleerzeichen
- j mit Count (n j) bewegt den Cursor *n* Zeilen nach unten (gleiche Spalte)

- k mit Count (n k) bewegt den Cursor *n* Zeilen nach oben (gleiche Spalte)

Der Unterschied zwischen + / - und j / k ist subtil, aber wichtig: + und - richten den Cursor am ersten sichtbaren Zeichen der Zeile aus, während j und k die aktuelle Spalten- position beibehalten. Das ist vor allem bei eingerücktem Code relevant.

Übungen

- Erzeuge einen eingerückten Codeblock (z. B. mit führenden Leerzeichen oder Tabs) und bewege dich zeilenweise mit j und k. Wiederhole die gleichen Bewegungen mit + und - und beobachte die unterschiedlichen Cursorpositionen.
- Setze den Cursor an den Anfang eines längeren Absatzes und springe mit 5 j bzw. 5 k durch den Text. Wiederhole die Übung mit + und - und vergleiche, welche Variante für dich lesefreundlicher ist.
- Kombiniere Operatoren mit Zeilenbewegungen:
  - d + löscht den Rest der aktuellen Zeile und die erste nichtleere Position der nächsten
  - 2 d k löscht die aktuelle und die zwei vorherigen Zeilen (bei passender Position)

  Probiere verschiedene Counts und beobachte genau, welche Bereiche jeweils entfernt werden.

1.3.5. Seitenweises Scrollen

Wenn der Text länger als eine Bildschirmseite ist, lohnt sich das seitenweise Scrollen. Diese Befehle verschieben den sichtbaren Bildausschnitt, ohne die logische Cursorposition zu verlieren.

- Ctrl - f scrollt eine Bildschirmseite nach vorne
- Ctrl - b scrollt eine Bildschirmseite zurück
- Ctrl - d scrollt eine halbe Bildschirmseite nach vorne
- Ctrl - u scrollt eine halbe Bildschirmseite zurück

Wichtig: Der Cursor bleibt dabei im sichtbaren Bereich der Datei. Wenn der Bildschirm scrollt, wird der Cursor nur dann mit verschoben, wenn er sonst aus

dem Sichtfenster fallen würde. So kannst du den Kontext ändern, ohne ständig die Cursorposition neu setzen zu müssen.

Übungen

- Öffne eine Datei mit deutlich mehr als einer Bildschirmseite Text. Bewege dich zunächst mit **j** und **k** und beobachte, wie der Cursor den Bildschirm zum Scrollen zwingt. Wiederhole das gleiche Szenario mit **Ctrl**-**d** und **Ctrl**-**u** und vergleiche das Verhalten.
- Scrolle mit **Ctrl**-**f** und **Ctrl**-**b** durch eine längere Datei und achte darauf, wie VIM den Cursor möglichst im sichtbaren Bereich hält. Versuche, nach jedem Scrollen sofort mit einem Operator (**d**, **c**, **y**) eine Operation auszuführen.
- Kombiniere seitenweises Scrollen mit Suchen:
  - Scrolle mit **Ctrl**-**f** vorwärts, führe dann eine Suche mit `:/Muster` aus und beobachte, wie sich Bildausschnitt und Cursor zueinander verhalten.
  - Wiederhole die Übung mit **Ctrl**-**b** und rückwärts gerichteten Suchen (`:?Muster`).

  Ziel: Du sollst ein Gefühl dafür entwickeln, wann Scrollen und wann Suchen sinnvoller ist.

1.3.6. Direktes Springen in der Datei

Für grobe Navigation durch die Datei stellt VIM spezielle Sprungbefehle zur Verfügung. Statt sich zeilenweise vorzuarbeiten, kannst du damit direkt an den Anfang, das Ende oder eine konkrete Zeile springen.

- **g** **g** springt zur *ersten Zeile* und dort an die *erste Spalte*.
- **G** springt zur *letzten Zeile* und dort an die *erste Spalte* (also an den Zeilenanfang, nicht ans Dateiende).
- <n> **G** springt zur Zeile <n> und dort an die erste Spalte.

Hinweis: Letzte Zeile vs. tatsächliches Dateiende

G bewegt den Cursor zur letzten *textuellen Zeile*, nicht zwingend an das echte Dateiende (EOF). Das Dateiende liegt ggf. hinter dem letzten sichtbaren Zeichen, z. B. nach einem finalen Zeilenumbruch.

Um an das tatsächliche Dateiende zu springen, kombinierst du:

G $

Dies bewegt den Cursor zur letzten Zeile *und* an deren Ende.
Typische Kombination:

- `:set number` Zeilennummern anzeigen
- 1 2 3 G direkt in Zeile 123 springen

Übungen

- Öffne eine längere Datei, aktiviere mit `:set number` die Zeilennummern und springe mit g g, G und <n> G systematisch durch verschiedene Bereiche der Datei. Beobachte, wo der Cursor jeweils landet.
- Erzeuge am Dateiende eine zusätzliche Leerzeile. Springe mit G und anschließend mit G $. Achte darauf, wie sich die Cursorposition im letzten Absatz verändert.
- Kombiniere Sprünge mit Operatoren:
  - d G löscht vom aktuellen Cursor bis zur letzten Zeile.
  - d g g löscht vom aktuellen Cursor bis zum Dateianfang.

  Führe diese Befehle zunächst in einer Testdatei aus, um ein Gefühl für die Reichweite solcher Operator–Motion-Kombinationen zu bekommen.

1.3.7. Klammerbezogene Bewegung

Für das Navigieren in verschachtelten Strukturen (Code, Konfigurationsdateien, LaTeX) ist die klammerbezogene Bewegung besonders wichtig:

- % springt von einer Klammer zur jeweils passenden Gegenklammer (( ), { }, [ ]), sofern der Cursor auf einer dieser Klammern steht.

Diese Bewegung wird in der Praxis meist mit Textobjekten kombiniert, z. B.:

- d i (löscht den Inhalt der runden Klammern, die den Cursor umgeben.
- c i (ändert den Inhalt der aktuellen Klammer.
- d a { löscht einen kompletten Block inklusive geschweifter Klammern.

`%` hilft, die Struktur zu verstehen und schnell zwischen öffnender und schließender Klammer zu springen – gerade in tief verschachteltem Code ist das deutlich robuster als reines Scrollen.

Übungen

- Öffne eine Datei mit verschachteltem Code (z. B. C, Python mit Klammern, oder eine LaTeX-Datei mit vielen `{`...`}`). Positioniere den Cursor abwechselnd auf öffnenden und schließenden Klammern und springe mit `%` hin und her.
- Kombiniere `%` mit Operatoren:
  - `d` `%` löscht vom Cursor bis zur passenden Klammer.
  - `y` `%` kopiert den Bereich zwischen zwei Klammern.

  Probiere dies in einer Testdatei aus und beobachte, wie sich die Reichweite der Operation verändert.
- Übe gezielt Textobjekte rund um Klammern:
  - `c` `i` `(`, `d` `i` `(`, `y` `i` `(`
  - `c` `i` `{`, `d` `a` `{`, `y` `a` `{`

  Stelle dir dabei immer die Frage: „Welcher Bereich wird genau erfasst?“

1.3.8. Warum Basisbewegungen so wichtig sind

Jede komplexe VIM-Interaktion nutzt die Basisbewegungen als Bausteine. Wer sie sicher beherrscht, muss nicht mehr darüber nachdenken, *wie* der Cursor bewegt wird, sondern kann sich auf die eigentliche Bearbeitungsaufgabe konzentrieren.

Das führt zu:

- präziseren Operator–Motion-Kombinationen,
- weniger unnötigem Scrollen oder Suchen mit der Maus,
- stabileren Makros, weil die Bewegungen definiert und reproduzierbar sind,
- klaren, wiederholbaren Editierprozessen, die auch in anderen Dateien funktionieren.

Langfristig sind diese Bewegungen das Muskelgedächtnis von VIM: Sie laufen automatisiert ab und bilden die Grundlage für alle weiteren Techniken.

Übungen

- Nimm dir eine beliebige Textdatei und erledige eine kleine Aufgabe (z. B. eine Liste umsortieren oder ein paar Zeilen umschreiben), *ohne* Pfeiltasten oder Maus – nur mit h, j, k, l, 0, ^, $, w, b, e.
- Notiere dir für jede Basisbewegung mindestens eine typische Operator–Motion-Kombination, z. B. d w, d $, y 0, c 3 w. Spiele diese Kombinationen in einer Testdatei bewusst mehrmals durch.
- Zeichne ein kurzes Makro auf, das eine einfache wiederkehrende Aufgabe nur mit Basisbewegungen und einem Operator löst (z. B. q a … q, dann @ a und @ @). Prüfe, ob das Makro auch in anderen, ähnlich aufgebauten Bereichen der Datei stabil funktioniert.

2. Eingabemodi

2.1. Insert

Der Insert-Mode (i) ist der klassische Eingabemodus in VIM. Im Insert-Mode verhält sich VIM weitgehend wie ein normaler Editor: getippte Zeichen erscheinen direkt im Text, der Cursor wandert nach rechts und die meisten Tasten haben ihre wörtliche Bedeutung.

Entscheidend ist jedoch, *wie* man in den Insert-Mode gelangt – denn jede Einstiegsmethode bestimmt die Cursorposition und damit die spätere Makrotauglichkeit. Ein schlecht gewählter Einstieg macht Makros fragil, ein gut gewählter Einstieg (z. B. zeilenweise) hält sie stabil.

2.1.1. Einstieg in den Insert-Mode

Die wichtigsten Einstiege in den Insert-Mode sind:

- i vor dem Cursor einfügen
- I am Zeilenanfang einfügen (erstes Nichtleerzeichen)
- a nach dem Cursor einfügen
- A am Zeilenende einfügen
- o neue Zeile *unterhalb* einfügen und in den Insert-Mode wechseln
- O neue Zeile *oberhalb* einfügen und in den Insert-Mode wechseln

Makro-Praxis:

- A ist für zeilenweise Makros häufig robuster als i, weil es unabhängig von der aktuellen Spaltenposition ans Zeilenende springt.
- I ist sinnvoll, wenn Makros an der *logischen* Zeilenanfang-Spalte (erstes Nichtleerzeichen) ansetzen sollen – etwa bei Code-Einrückungen.
- o und O eignen sich, um Makros mit neu eingefügten Zeilen zu kombinieren (z. B. Log- oder Kommentarzeilen).

Übungen

- Lege drei Zeilen an:

  ```
  foo
  bar
  baz
  ```

  Setze den Cursor auf das „b“ von `bar` und teste nacheinander **i**, **a**, **I**, **A**. Beende jede Eingabe mit **Esc** und beobachte, wo der Cursor steht und wie sich der Text verändert.
- Setze den Cursor in die Mitte eines Absatzes und benutze **o** und **O**, um neue Zeilen oberhalb bzw. unterhalb einzufügen. Fülle die neuen Zeilen mit kurzem Text, beende mit **Esc** und navigiere mit **k** / **j** durch den Block, um die neue Struktur zu prüfen.
- Nimm eine Datei mit mehreren Codezeilen und zeichne ein einfaches Makro auf:
  1. Cursor auf eine Codezeile stellen
  2. **A** drücken, Kommentar wie `//  TODO` anhängen, **Esc**
  3. Makro im Register **q** speichern (**q** **q** … **q**)

  Wende das Makro mit **@** **q** auf die nächsten Zeilen an. Wiederhole das Experiment anschließend mit **i** statt **A** und beobachte, wie viel empfindlicher das Makro auf unterschiedliche Cursorpositionen reagiert.

2.1.2. Verhalten im Insert-Mode

Im Insert-Mode gelten folgende Grundregeln:

- **Esc** oder **Ctrl**-**[** verlässt den Insert-Mode
- **Backspace** löscht das Zeichen links vom Cursor
- **Ctrl**-**h** wirkt wie **Backspace**
- **Ctrl**-**w** löscht das letzte Wort (bis zur Wortgrenze)
- **Ctrl**-**u** löscht den Text von der Cursorposition bis zum Zeilenanfang

Gerade **Ctrl**-**w** und **Ctrl**-**u** sind in Makros wertvoll, weil sie unabhängig vom konkreten Zeileninhalt arbeiten und trotzdem deterministisch bleiben. Statt im Makro mit Pfeiltasten oder **x** einzelne Zeichen zurückzulöschen, ist ein „Reset“ per **Ctrl**-**w** oder **Ctrl**-**u** deutlich robuster.

Übungen

- Lege eine Zeile an:

  `foo_bar_baz`

  Setze den Cursor ans Zeilenende, gehe mit **a** in den Insert-Mode und teste nacheinander **Backspace**, **Ctrl**-**h** und **Ctrl**-**w**. Beobachte, wie sich die Löschgranularität unterscheidet.
- Erzeuge eine längere Zeile Text, gehe mit **A** in den Insert-Mode und lösche dann mit **Ctrl**-**u** zurück zum Zeilenanfang. Wiederhole die Übung mehrmals und überprüfe, dass die Wirkung unabhängig vom konkreten Textinhalt stabil bleibt.
- Zeichne ein kleines Makro auf, das am Zeilenende ein Wort anhängt, dann mit **Ctrl**-**w** wieder löscht und mit **Esc** endet. Spiele das Makro auf mehreren Zeilen ab und vergleiche die Stabilität mit einer Variante, in der du stattdessen mehrfach **Backspace** verwendest.

2.1.3. Einfügen von Steuerzeichen

Einige Zeichen lassen sich im Insert-Mode nicht „normal“ tippen, sondern müssen als *Literal* eingefügt werden. Dafür dient der *Literal-Eingabemodus*:

- **Ctrl**-**v** **Esc** fügt ein echtes ESC-Zeichen in den Text ein
- **Ctrl**-**v** **Tab** fügt ein echtes Tabulator-Zeichen ein
- **Ctrl**-**v** **0** **1** **3** fügt das Zeichen mit dem ASCII-Code `013` ein

Nach **Ctrl**-**v** erwartet VIM das *nächste* Zeichen entweder direkt von der Tastatur (z. B. **Esc**) oder als Zahlenfolge (Oktal, Dezimal oder Hex). So lassen sich Zeichen einfügen, die sonst eine besondere Bedeutung hätten, etwa Steuerzeichen oder „unsichtbare“ Separatoren.

Hinweis

- Im *Normal-Mode* aktiviert **Ctrl**-**v** den Visual-Block-Mode. Der Literal-Modus funktioniert wie beschrieben im *Insert-Mode*.
- In einigen VIM-Varianten (z. B. Neovim/GVim) kann auch **Ctrl**-**q** für die Literal-Eingabe konfiguriert sein.

Übungen

- Erzeuge eine Zeile wie:

  ESC: [ ]

  Setze den Cursor zwischen die eckigen Klammern, gehe mit [i] in den Insert-Mode und füge mit [Ctrl]-[v][Esc] ein echtes ESC-Zeichen ein. Prüfe mit Exset list, wie VIM das Steuerzeichen darstellt.
- Lege eine Datei an, in der Tabs und Leerzeichen klar unterscheidbar sind (z. B. mit Exset list listchars=tab:^T,trail:_). Füge an verschiedenen Stellen mit [Ctrl]-[v][Tab] echte Tabs ein und vergleiche sie mit normalen Leerzeichen.
- Teste die Eingabe über Zahlencodes: Gehe in den Insert-Mode und tippe nacheinander [Ctrl]-[v][0][1][0], [Ctrl]-[v][0][1][1], [Ctrl]-[v][0][1][2]. Beobachte in einer Hex- oder Exset list-Ansicht, welche Steuerzeichen du eingefügt hast und wie sie sich bei Exset ff=unix bzw. Exset ff=dos verhalten.

2.1.4. Einfügen aus Registern

Auch im Insert-Mode lassen sich Inhalte aus beliebigen Registern einfügen. Das Grundmuster lautet:

[Ctrl]-[r] <reg>

Dabei steht <reg> für den Namen des Registers.
Typische Beispiele:

- [Ctrl]-[r][0] fügt den Inhalt des „Yank-Registers“ "0 ein (zuletzt *kopierter* Text, z. B. durch [y][y] oder [y][i][p])
- [Ctrl]-[r][”] fügt das Standardregister "" ein (meist die letzte Änderung / der letzte Yank)
- [Ctrl]-[r][+] fügt die System-Zwischenablage ein (CLIPBOARD, z. B. per Maus oder anderem Programm kopiert)
- [Ctrl]-[r][*] fügt die X11-PRIMARY-Auswahl ein (unter Linux/X11)

So lassen sich im Insert-Mode beliebige Registerinhalte punktgenau einfügen, ohne den Normal-Mode zu verlassen.

Übungen

- Kopiere eine Zeile mit [y][y], gehe in eine andere Zeile in den Insert-Mode und füge mit [Ctrl]-[r][0] ein. Vergleiche das Verhalten mit [p] aus dem Normal-Mode.

- Markiere im Visual-Mode ein Wort mit v i w und kopiere mit y. Gehe in den Insert-Mode an einer anderen Stelle und füge den Inhalt mit Ctrl-r " ein. Prüfe, dass es sich tatsächlich um das Standardregister handelt.
- Kopiere in einem anderen Programm (Browser, Terminal, Mail) einen Text in die Zwischenablage. Wechsle zu VIM, gehe in den Insert-Mode und füge mit Ctrl-r + (bzw. Ctrl-r * unter X11) ein. Kontrolliere, ob der erwartete Clipboard-Inhalt eingefügt wurde.

2.1.5. Insert-Mode in Makros

Der Insert-Mode ist in Makros einer der störanfälligsten Bereiche — nicht, weil er *unsicher* wäre, sondern weil sich kleine Unterschiede im Text sofort auf die Reproduzierbarkeit auswirken. Kritisch wird es vor allem, wenn:

- der Cursor an variabler Position startet
- der zu bearbeitende Text unterschiedlich lang ist
- man lange im Insert-Mode bleibt und „frei tippt“

Robuste Strategien für Makros:

- A bevorzugen statt i, wenn zeilenweise gearbeitet wird (Ansatz immer am Zeilenende statt irgendwo im Wort)
- Einfügephasen klar begrenzen (nur das Nötige tippen, sofort mit Esc schließen)
- strukturelle Bewegungen und Textobjekte *vor* dem Insert-Mode nutzen (z. B. c i w, c i p statt Pfeiltasten)
- bei Serienänderungen lieber einmal sauber ändern und danach mit . wiederholen, statt denselben Insert-Block im Makro mehrfach aufzuzeichnen

Der Insert-Mode ist in VIM keine neutrale „Eingabestufe“, sondern ein gezielt eingesetztes Werkzeug: Wer Makros stabil halten will, sorgt dafür, dass Einfügen immer an definierten, strukturell stabilen Positionen beginnt und schnell wieder in den Normal-Mode zurückkehrt.

Übungen

- Nimm eine Liste wie:

```
item 1
item 2
item 3
```

Zeichne ein Makro q a auf, das ans Zeilenende [OK] anhängt (A [OK] Esc), und spiele es mit @ a und @ @ auf den restlichen Zeilen ab. Beobachte, wie stabil A im Vergleich zu i ist.

- Lege eine Zeile an:

  `status: pending`

  Zeichne ein Makro, das mit c i w `pending` durch `done` ersetzt und sofort mit Esc endet. Füge weitere Zeilen mit `status: pending` hinzu und wende das Makro dort an. Prüfe, ob unterschiedliche Einrückungen oder zusätzliche Leerzeichen das Makro stören.
- Erzeuge bewusst ein „schlechtes“ Makro, in dem du lange im Insert-Mode bleibst und mit Pfeiltasten korrigierst. Spiele es auf leicht veränderten Zeilen ab und analysiere, an welchen Stellen es bricht. Vergleiche das mit einer Variante, die dasselbe Ziel mit c i w + kurzem Einfügen + Esc erreicht.

2.2. Append

Der Append-Modus ist eine Variante des Insert-Modes. Während i Text *vor* dem Cursor einfügt, beginnt a direkt *hinter* dem aktuellen Zeichen. Append ist damit die direkte Ergänzung zu Insert und wird vor allem dann genutzt, wenn Eingaben an bestehende Inhalte „angehängt“ werden sollen.

2.2.1. Einstieg in den Append-Modus

- a beginnt die Eingabe *hinter* dem Cursor
- A beginnt die Eingabe am *Zeilenende*

Der Unterschied ist entscheidend:

- a ist strikt von der aktuellen Cursorposition abhängig
- A ist positionsunabhängig und springt immer ans Zeilenende

Gerade in Makros ist A meist robuster als a, weil sich die Zeichenlänge einer Zeile ändern darf, ohne dass der Einstiegspunkt im Append-Modus verrutscht.

2.2.2. Typisches Verhalten

Im Append-Modus gelten dieselben Regeln wie im Insert-Mode:

- Esc beendet den Modus
- Ctrl - w löscht das letzte Wort
- Ctrl - u löscht bis zum Zeilenanfang
- Ctrl - r <reg> fügt den Inhalt eines Registers ein

Der Unterschied zum normalen Insert-Einstieg mit i ist die Startposition des Cursors: Append-Kommandos (a / A) beginnen immer *hinter* dem ursprünglichen Zeichen bzw. am Zeilenende. Das ist insbesondere für Makros relevant, weil a -basierte Makros empfindlich auf geänderte Cursorpositionen reagieren, während A -basierte Makros am Zeilenende deutlich stabiler sind.

2.2.3. Wann a sinnvoll ist

a ist die beste Wahl, wenn:

- ein Zeichen oder Wort direkt erweitert werden soll
- zuvor ein Operator wie f X oder t X den Cursor exakt positioniert hat
- Makros bewusst an einer variablen Position anknüpfen sollen

Beispiel:

f a a :

Findet in der aktuellen Zeile das nächste a und hängt direkt danach einen Doppelpunkt an. Die eigentliche Präzision kommt dabei vom Bewegungsbefehl f a . Der anschließende Befehl a nutzt nur die bereits sauber gesetzte Cursorposition, um den Doppelpunkt anzuhängen.

2.2.4. Wann A die bessere Wahl ist

A ist weit robuster und wird für Makros fast immer bevorzugt:

- das Zeilenende ist deterministisch
- unabhängig vom konkreten Zeileninhalt
- funktioniert bei leeren und nicht-leeren Zeilen identisch
- ideal für Formatierungen, Listen und Logging-Zeilen

Beispiel:

A ;

fügt zuverlässig ein Semikolon am Zeilenende ein – egal, wie die Zeile aussieht oder wo der Cursor ursprünglich stand. Genau diese Inhaltsunabhängigkeit macht A zur Standardwahl für makrotaugliche Append-Operationen.

2.2.5. Append in Makros

Makros sollten Append nur nutzen, wenn der Kontext eindeutig ist:

- a ist heikel, wenn der Cursor „im Wort steckt“
- A ist deutlich sicherer, wenn Zeilen vereinheitlicht werden sollen
- für strukturierte Aufgaben ist c + Textobjekt oft stabiler

Faustregel für Makros:

> Wenn du *nicht sicher weißt*, dass der Cursor an der richtigen Stelle steht, nimm A statt a.

Der Append-Modus ist damit kein „Vielzweckwerkzeug“, sondern ein präziser Modus für gezielte Erweiterungen — besonders in klaren, reproduzierbaren Strukturen.

2.3. Neue Zeilen (o, O)

Die Befehle o und O erstellen neue Zeilen und wechseln automatisch in den Insert-Modus. Sie sind oft die stabilste Methode, um Text einzufügen, weil der Cursor präzise und reproduzierbar positioniert wird – unabhängig vom bisherigen Zeileninhalt.

2.3.1. Bedeutung von o und O

- o erstellt eine neue Zeile *unterhalb* der aktuellen Zeile und wechselt in den Insert-Modus
- O erstellt eine neue Zeile *oberhalb* der aktuellen Zeile und wechselt in den Insert-Modus

Beide Befehle erzeugen stets eine vollständig neue, „saubere“ Zeile – egal, wie die aktuelle Zeile aussieht.

2.3.2. Verhalten und Eigenschaften

- die neue Zeile übernimmt die Einrückung der aktuellen Zeile
- der Cursor steht sofort am Anfang der neuen Zeile
- VIM befindet sich direkt im Insert-Modus
- **Esc** beendet den Insert-Modus und kehrt in den Normal-Modus zurück

Gerade die automatische Übernahme der Einrückung macht **o**/**O** im Programmierumfeld besonders nützlich (Codeblöcke, If/Else-Strukturen, Schleifen, Funktionskörper).

2.3.3. Praktische Beispiele

Eine neue Zeile *unterhalb* einfügen:

```
:oThis is a new line
```

Eine neue Zeile *oberhalb* einfügen:

```
:O// Kommentar einfügen
```

Neue Listenpunkte erzeugen:

```
:o- neuer Eintrag
```

Übungen

- Erzeuge eine kleine C- oder Lua-Funktion und füge zwischen zwei bestehenden Zeilen eine neue Logzeile mit **O** ein (z. B. `// TODO: Logging ergänzen`). Beobachte, wie die Einrückung übernommen wird.
- Lege eine Markdown-Liste an und füge mit **o** mehrere weitere Einträge hinzu:

  ```
  - Punkt 1
  - Punkt 2
  ```

  Ergänze „Punkt 3“, „Punkt 4“ nur mit **o** und Text – ohne die Einrückung manuell anzupassen.
- Nimm ein Makro auf, das mit **O** oberhalb der aktuellen Zeile einen Kommentar einfügt (z. B. `// MARK`) und mit **Esc** sauber in den Normal-Modus zurückkehrt. Spiele das Makro in unterschiedlichen Codebereichen ab und prüfe, ob die Einrückung überall korrekt ist.

2.3.4. Einsatz in Makros

o und O gehören zu den stabilsten Makro-Bausteinen überhaupt, weil sie völlig unabhängig von Inhalt oder exakter Cursorposition funktionieren.

Vorteile:

- reproduzierbares Verhalten
- keine Abhängigkeit von Zeilenlänge
- keine Gefahr des „Cursor-mitten-im-Wort“-Effekts
- immer kontrollierter Übergang in den Insert-Modus

Typische Makro-Strategien:

- neue Log-Zeilen oder Protokolleinträge erzeugen
- automatisch Listen oder Tabellenzeilen erweitern
- Formatierungen, Header oder Kommentarblöcke einfügen

Beispiel (Kommentarkopf oberhalb einfügen):

O `// TODO: <Text>` Esc

Dieser Ablauf ist deterministisch: Die neue Zeile entsteht immer oberhalb der aktuellen und übernimmt deren Einrückung.

Übungen

- Nimm ein Makro auf, das mit O oberhalb der aktuellen Zeile einen Kommentar `// MARK` einfügt und mit Esc sauber beendet. Spiele es in unterschiedlichen Codebereichen ab und prüfe, ob Einrückung und Position immer stimmen.
- Erzeuge eine einfache Log-Tabelle und ergänze mit einem o-basierten Makro automatisch neue Einträge (Datum, Status, Nachricht) in einer neuen Zeile unterhalb.

2.3.5. Unterschiede zu Insert/Append

- o/O erzeugen garantiert neue Zeilen — i/a nie
- der Cursor steht immer am Anfang der neuen Zeile
- die Einrückung der Umgebung wird automatisch übernommen
- die Makro-Stabilität ist deutlich höher als bei i/a

Damit sind o und O die erste Wahl, wenn Zeilen strukturell neu eingefügt oder umbrochen werden sollen — insbesondere in automatisierten Abläufen und Makros, die in ganzen Dateien oder größeren Codeblöcken laufen.

2.4. Replace Mode (r, R)

Der Replace-Modus bietet eine alternative Art der Eingabe: Text wird nicht eingefügt, sondern bestehende Zeichen werden *überschrieben*. Das unterscheidet ihn klar vom normalen Insert-Modus.

VIM kennt zwei Varianten:

- r einzelnes Zeichen ersetzen
- R kontinuierliches Ersetzen (Replace Mode)

2.4.1. Einzelnes Ersetzen mit r

Mit dem Befehl r wird genau ein Zeichen ersetzt:

`foo` ⇒ r X ⇒ `Xoo`

Eigenschaften:

- ersetzt exakt *ein* Zeichen an der Cursorposition
- der Cursor bleibt auf derselben Position stehen
- ideal für punktuelle Korrekturen (*Tippfehler, einzelne Zeichen*)
- sehr makrotauglich, weil das Verhalten deterministisch ist

Übungen

- Lege eine Zeile `abcdef` an, positioniere den Cursor nacheinander auf jedem Buchstaben und ersetze mit r X. Beobachte, dass immer genau ein Zeichen überschrieben wird.
- Teste den Unterschied zu c w: Ersetze ein Zeichen einmal mit r X und einmal mit c w + Text. Vergleiche Cursorposition und Undo-Verhalten.

2.4.2. Replace Mode mit R

Mit R wechselt VIM in einen kontinuierlichen Überschreibmodus. Statt Text einzufügen, werden bestehende Zeichen überschrieben.

Ablauf:

R Text eingeben **Esc**

Beispiel:

Ausgang: `Status: FAIL` (Cursor auf dem `F` von `FAIL`)
Tastfolge: **R** DONE **Esc**
Ergebnis: `Status: DONE`

Eigenschaften:
- überschreibt vorhandene Zeichen statt neue einzufügen
- Cursor bewegt sich wie im Insert-Modus nach rechts
- Zeilenlänge bleibt unverändert, solange die Eingabe genauso lang ist
- ideal für feste Spalten, Tabellen und ausgerichteten Code

2.4.3. Verhalten am Zeilenende

Sobald im Replace Mode mehr Zeichen eingegeben werden, als in der Zeile vorhanden sind, schaltet VIM intern auf „Einfügen“ um:
- solange noch Originalzeichen existieren → reines Überschreiben
- sind alle Originalzeichen verbraucht → zusätzliche Zeichen werden angehängt

Dadurch kann sich die Zeilenlänge am Ende doch verändern, wenn mehr Text eingegeben wird als ursprünglich vorhanden war.

2.4.4. Undo-Verhalten

Eine komplette Replace-Session (von **R** bis **Esc**) gilt als *ein* zusammenhängender Undo-Block:
- **u** macht den gesamten Replace-Vorgang rückgängig
- **Ctrl**-**r** stellt ihn komplett wieder her

Feinkörniges Undo innerhalb der Session gibt es nicht.

2.4.5. Makrotauglichkeit

Der Replace Mode ist grundsätzlich gut für Makros geeignet:

- deterministische Cursorbewegung nach rechts
- Undo sauber gebündelt in einem Schritt
- kein seitliches „Wandern“ durch Einfüge-Operationen

Schwachstelle:

- bei Zeilen unterschiedlicher Länge kann das Ergebnis abweichen, wenn in einigen Zeilen früher das Zeilenende erreicht wird als in anderen

Faustregel: **R** ist robust, solange alle betroffenen Zeilen eine vergleichbare Struktur und mindestens die notwendige Länge haben.

2.5. Spezialbefehle im Insert Mode

Neben der normalen Texteingabe bietet der Insert-Modus eine Reihe von Steuerbefehlen, die direkt während der Eingabe ausgeführt werden können. Diese Befehle sind oft schneller und stabiler als vergleichbare Normal-Mode-Operationen, insbesondere in Makros.

2.5.1. Löschbefehle im Insert-Modus

Diese Befehle ermöglichen effizientes Korrigieren schon während der Eingabe:

- **Ctrl**-**h** wirkt wie Backspace (ein Zeichen löschen)
- **Ctrl**-**w** löscht das vorherige Wort
- **Ctrl**-**u** löscht den Text von der Cursorposition bis zum Zeilenanfang

Vorteile gegenüber expliziten Normal-Mode-Befehlen:

- *unabhängig vom aktuellen Textlayout* (kein exaktes Positionieren des Cursors nötig)
- *immer deterministisch* — dasselbe Makro verhält sich bei unterschiedlichem Inhalt identisch
- *sehr makrotauglich*, weil der Insert-Modus nicht ständig verlassen und wieder betreten werden muss

Übungen

- Lege eine Zeile an wie

  ```
  foo bar baz qux
  ```

  Setze den Cursor ans Zeilenende, gehe in den Insert-Modus (**A**) und tippe ein paar zusätzliche Zeichen. Nutze dann **Ctrl**-**w**, um schrittweise Wörter

zurückzunehmen. Beobachte, dass die Löschung unabhängig von der exakten Cursorposition im Wort funktioniert.

- Schreibe in einer neuen Zeile einen längeren Satz, z. B.:

  ```
  Dies ist ein Testsatz mit zu viel Text
  dahinter.
  ```

  Wechsle ans Zeilenende (**$** oder **A**) und nutze dann im Insert-Modus **Ctrl**-**u**. Kontrolliere, dass der komplette Zeileninhalt bis zum Anfang gelöscht wird, ohne dass du zuvor in den Normal Mode wechseln musst.
- Nimm eine bestehende Konfigurationszeile (z. B. aus einer echten `.vimrc`) und korrigiere Tippfehler ausschließlich mit **Ctrl**-**h**, **Ctrl**-**w** und **Ctrl**-**u** im Insert-Modus. Ziel: möglichst keine Normal-Mode-Löschbefehle wie **x** oder **d** **w** verwenden und trotzdem präzise korrigieren.

2.5.2. Einfügen von Registerinhalten

Während der Eingabe kann beliebiger Registerinhalt eingefügt werden:

`Ctrl-r<reg>`

Beispiele:

- **Ctrl**-**r** **0** fügt den zuletzt gelöschten Text ein
- **Ctrl**-**r** **”** fügt das Standardregister ein
- **Ctrl**-**r** **+** fügt die System-Zwischenablage (Clipboard) ein

Dies ist einer der stabilsten Wege, Text automatisiert aus Registern einzufügen, insbesondere in Makros.

2.5.3. Einfügen von wörtlichen Zeichen

Einige Zeichen müssen „wörtlich“ eingefügt werden, weil sie sonst als Steuerzeichen interpretiert würden. Dazu dient:

`Ctrl-v<key>`

Wichtige Beispiele:

- `Ctrl-v Esc` fügt ein echtes ESC-Zeichen ein
- `Ctrl-v Tab` fügt einen echten Tabulator ein
- `Ctrl-v 013` fügt ein Zeichen über den ASCII-Code ein
- `Ctrl-v \` fügt einen Backslash ein

Das ist besonders nützlich in Konfigurationsdateien, Shell-Skripten oder anderen Kontexten, in denen Sonderzeichen exakt kontrolliert werden müssen.

2.5.4. Navigieren im Insert-Modus

Auch im Insert-Modus ist begrenzte Navigation möglich:

- **Ctrl**-**o** führt *einen* Normal-Mode-Befehl aus und kehrt danach direkt in den Insert-Modus zurück.

Beispiel:

`:\Ctrl {o} \Key {d}\KeySep \Key {d}`

löscht die aktuelle Zeile komplett, ohne den Insert-Modus dauerhaft zu verlassen – nach **d** **d** befindet sich der Cursor wieder im Insert-Modus an der neuen Cursorposition.

Wichtige Punkte:

- **Ctrl**-**o** ist makrotauglich und sehr gut kontrollierbar
- „Ein Befehl“ bedeutet: genau *eine* Normal-Mode-Operation (z. B. **d** **d**, **w**, **{**), auch wenn sie aus mehreren Tasten besteht
- weitere Tasten nach dem ersten Normal-Mode-Befehl werden wieder als normale Eingabe im Insert-Modus gewertet

Übungen

- Schreibe eine Zeile Text, gehe in den Insert-Modus und teste: `:\Ctrl {o} \Key {0}` Beobachte, dass der Cursor kurz an den Zeilenanfang springt und du danach weiter tippen kannst, ohne den Insert-Modus verlassen zu haben.
- Schreibe mehrere Zeilen untereinander und probiere: `:\Ctrl {o} \Key {d}\KeySep \Key {d}` Kontrolliere, dass die aktuelle Zeile gelöscht wird und du direkt wieder im Insert-Modus stehst.

2.5.5. Auto-Completion im Insert-Modus

VIM stellt im Insert-Modus eine Reihe leistungsfähiger Vervollständigungsfunktionen bereit. Sie greifen auf den aktuellen Puffer, Dateinamen oder sprachspezifische Informationen zurück.

Wichtige Befehle:

- **Ctrl**-**n** nächster Vorschlag aus dem aktuellen Kontext

- Ctrl - p vorheriger Vorschlag
- Ctrl - x Ctrl - l komplette Zeile vervollständigen
- Ctrl - x Ctrl - f Dateinamen vervollständigen
- Ctrl - x Ctrl - o „omni completion“ (kontextabhängig, z. B. via LSP / Syntax)

Diese Befehle sind besonders nützlich in Code-Projekten, Konfigurationsdateien und langen Texten, da sie Tippaufwand reduzieren und Tippfehler minimieren.

Übungen

- Lege eine Datei an und schreibe einige wiederkehrende Begriffe (z. B. Funktionsnamen oder längere Bezeichner). Tippe nur den Anfang eines Wortes und verwende dann Ctrl - n und Ctrl - p, um die Vorschläge durchzublättern.
- Erzeuge mehrere ähnliche Zeilen und nutze Ctrl - x Ctrl - l, um komplette Zeilen zu vervollständigen. Beobachte, wann VIM passende Vorschläge anbietet und wann nicht.
- Tippe in einem Verzeichnis mit mehreren Dateien einen Teilpfad (z. B. `./`) und verwende Ctrl - x Ctrl - f, um Dateinamen automatisch zu ergänzen.

2.5.6. Spezialfunktionen

Einige zusätzliche Steuermöglichkeiten im Insert-Modus:

- Ctrl - t Einrücktiefe der aktuellen Zeile erhöhen
- Ctrl - d Einrücktiefe der aktuellen Zeile verringern
- Ctrl - y Zeichen aus der Zeile *darüber* einfügen, ohne den Cursor zu bewegen
- Ctrl - e Zeichen aus der Zeile *darunter* einfügen, ohne den Cursor zu bewegen

Diese Befehle werden selten gelehrt, sind aber im Programmieralltag und bei strukturierten Textformaten extrem effizient, weil sie Einrückung und Kontext anpassen, ohne den Insert-Modus zu verlassen.

Übungen

- Lege einen kleinen Codeblock mit verschiedenen Einrückungen an und nutze Ctrl - t und Ctrl - d, um gezielt die Einrückung zu verändern, ohne in den Normal Mode zu wechseln.

- Erzeuge zwei Zeilen, in denen sich nur einzelne Zeichen unterscheiden. Positioniere den Cursor in der unteren Zeile und nutze **Ctrl**-**y** bzw. **Ctrl**-**e**, um Zeichen aus der benachbarten Zeile zu ziehen, ohne den Cursor zu verschieben.

2.5.7. Makrotauglichkeit der Insert-Spezialbefehle

Besonders stabil in Makros sind:

- **Ctrl**-**w**, **Ctrl**-**u** deterministisches Löschen im Insert-Modus
- **Ctrl**-**r** <reg> Registereinfügen
- **Ctrl**-**o** gefolgt von einem Normal-Mode-Kommando
- **Ctrl**-**t**, **Ctrl**-**d** Einrückungen gezielt anpassen

Weniger stabil sind:

- Auto-Completion (**Ctrl**-**n**, **Ctrl**-**p**) — hängt stark vom aktuellen Kontext ab
- Navigieren mit Pfeiltasten — erzeugt schwer reproduzierbare Cursorwege

Die Spezialbefehle machen den Insert-Modus zu einem präzisen Werkzeug für automatisierte, reproduzierbare Arbeit, sofern man sich auf die stabilen, kontextunabhängigen Varianten konzentriert.

3. Navigation

3.1. Wort- und Satznavigation

VIM unterscheidet präzise zwischen zwei Kategorien von „Wörtern“: `word` und `WORD`. Zusätzlich existieren Bewegungen auf Satz- und Absatzebene. Diese Navigation ist für viele Operator–Motion-Kombinationen essenziell, weil sie nicht nur einzelne Zeichen, sondern die semantische Struktur eines Textes nutzt.

- `word` zusammenhängende Folge aus Buchstaben, Ziffern und Unterstrich (`a–z`, `A–Z`, `0–9`, `_`); Interpunktion trennt.
- `WORD` alles, was nicht Leerraum ist; getrennt wird nur durch Whitespace.

Im Alltag arbeitest du meist mit `word`-Bewegungen, während `WORD`-Bewegungen (**W**, **E**, **B**) vor allem in Code mit vielen Sonderzeichen oder Pfaden (`/usr/local/bin/...`) ihre Stärken ausspielen.

3.1.1. Wortbasierte Navigation (w, e, b)

Für die feine Navigation innerhalb einer Zeile sind die „word“-Bewegungen zentral. Sie definieren, wie sich der Cursor über normale Wörter und Identifier hinweg bewegt.

Ein `word` ist eine alphanumerische Sequenz inklusive Unterstrich; Interpunktion wie `,` oder `.` beendet das `word` und beginnt ggf. ein neues.

- **w** springt zum Anfang des nächsten `word`.
- **e** springt zum Ende des aktuellen bzw. nächsten `word`.
- **b** springt zum Anfang des vorherigen `word`.

Diese Bewegungen sind elementar für präzise Bearbeitungen wie **d** **w**, **d** **e**, **c** **b** oder **y** **e**. Statt mühsam zeichenweise zu navigieren, beschreibst du mit **w**, **e**, **b** direkt logische Einheiten im Text.

Übungen

- Lege eine Testzeile an, z. B.:

```
foo_bar, foo.bar; foo-bar
```

Bewege den Cursor mit **w**, **e**, **b** über die Zeile und beobachte, wo VIM beginnt und endet. Notiere dir, was als `word` behandelt wird und was als Trenner dient.

- Wiederhole die gleiche Übung mit einer Zeile wie:

```
/usr/local/bin/script.sh --option=value
```

und vergleiche **w**, **e**, **b** mit den WORD-Varianten **W**, **E**, **B**. Achte darauf, wie sich die Sprungweiten unterscheiden.

- Kombiniere Operatoren mit Wortbewegungen:
  - **d** **w** löscht vom Cursor bis zum Beginn des nächsten `word`.
  - **d** **e** löscht bis zum Ende des aktuellen `word`.
  - **c** **b** ändert das vorherige `word`.

  Spiele diese Kombinationen systematisch an einer Testzeile durch, bis du „im Kopf“ hast, welcher Bereich jeweils betroffen ist.

3.1.2. WORD-basierte Navigation (W, E, B)

WORD bezeichnet eine durch Leerzeichen abgegrenzte Einheit. Interpunktion trennt hier *nicht*: Alles, was zwischen zwei Whitespace-Bereichen steht, gilt als ein einziges WORD.

- **W** springt zum Anfang des nächsten WORD.
- **E** springt zum Ende des aktuellen bzw. nächsten WORD.
- **B** springt zum Anfang des vorherigen WORD.

Diese Variante ist besonders hilfreich in Programmcode, Konfigurationsdateien oder Protokollen, in denen viele Sonderzeichen vorkommen (z. B. Pfade, Optionen, Logzeilen). Statt an jedem Sonderzeichen hängen zu bleiben, springst du mit **W**, **E**, **B** über ganze Blöcke hinweg.

Übungen

- Lege eine Zeile mit vielen Sonderzeichen an, z. B.:

```
/usr/local/bin/script.sh
--option=value,foo=bar;rc=0
```

Vergleiche das Verhalten von **w**, **e**, **b** mit **W**, **E**, **B**. Notiere dir, welche Bereiche als `word` und welche als WORD behandelt werden.

- Öffne eine reale Konfigurationsdatei (z. B. `.vimrc`, `nginx.conf`, `sshd_config`) und bewege dich mit **w**/**b** sowie **W**/**B** durch eine Zeile mit Optionen. Achte darauf, wann WORD-Bewegungen deutlicher schneller ans Ziel bringen.
- Kombiniere Operatoren mit WORD-Bewegungen:
  - **d** **W** löscht vom Cursor bis zum Beginn des nächsten WORD.
  - **c** **E** ändert vom Cursor bis zum Ende des aktuellen WORD.
  - **y** **2** **W** kopiert die nächsten zwei WORDs.

  Spiele diese Kombinationen in einer Testdatei durch und vergleiche sie mit den entsprechenden `word`-Varianten (**d** **w**, **d** **e**, **y** **2** **w**).

3.1.3. Vergleich von `word` und WORD

- `word`: fein granular, trennt an Interpunktion. Beispiel: `user.name = value;` besteht aus mehreren `word`-Einheiten.
- WORD: grob granular, trennt nur an Leerzeichen. Dasselbe Beispiel wird als ein einziges WORD betrachtet.

Dadurch lassen sich Operatoren sehr zielgerichtet anwenden:

- **d** **w** löscht nur das nächste „Teilwort“ (bis zur nächsten `word`-Grenze).
- **d** **W** löscht den gesamten Block bis zum nächsten Leerzeichen (eine WORD-Einheit).

Übungen

- Lege eine Testzeile an, z. B.:

  ```
  user.name = some_value.with.dots;
  ```

  Setze den Cursor an den Zeilenanfang und vergleiche: **d** **w**, **d** **W**, **d** **e**, **d** **E**. Notiere dir, welche Bereiche jeweils verschwinden.
- Wiederhole die Übung mit einer Zeile wie:

  ```
  path=/usr/local/bin:/usr/bin:/bin
  ```

  Teste **w**/**W** und **d** **w**/**d** **W**, um das unterschiedliche Verhalten von `word` und WORD in Präsenz von Doppelpunkten und Slashes nachvollziehen zu können.

3.1.4. Satznavigation ((und))

VIM definiert einen „Satz“ typischerweise über Satzzeichen wie `.`, `!` oder `?`, gefolgt von mindestens einem Leerzeichen oder einem Zeilenende.

- **)** springt zum Anfang des nächsten Satzes.
- **(** springt zum Anfang des vorherigen Satzes.

Diese Bewegungen sind besonders nützlich in Fließtext, Dokumentation oder Kommentaren, vor allem in Kombination mit Operatoren wie **d** **)** oder **c** **(**, um ganze Sätze gezielt zu löschen oder zu ändern.

Übungen

- Lege einen kurzen Absatz mit mehreren Sätzen an, z. B. drei bis vier Sätze mit unterschiedlichen Längen. Bewege den Cursor mit **(** und **)** durch den Text und beobachte, wie VIM die Satzgrenzen interpretiert.
- Teste **d** **)** und **c** **)** an verschiedenen Positionen innerhalb eines Satzes: mitten im Satz, am Anfang, am Ende. Notiere dir, wie viel jeweils gelöscht bzw. zur Bearbeitung freigegeben wird.
- Füge absichtlich ungewöhnliche Interpunktion ein (z. B. Abkürzungen wie `z. B.` oder `u. a.`) und prüfe, ob die Satznavigation an diesen Stellen noch sinnvoll reagiert oder ob du manuell nachsteuern musst.

3.1.5. Praktische Beispiele

- **d** **w** löscht nur das nächste Wortfragment (ein `word` im Sinne von VIM).
- **d** **W** löscht ein ganzes Argument oder Kommando (ein komplettes `WORD`, z. B. einen Pfad oder eine Option mit Sonderzeichen).
- **c** **)** ändert den gesamten aktuellen Satz.
- **y** **E** kopiert bis zum Ende des aktuellen `WORD`.

Diese Beispiele zeigen, wie stark sich das Verhalten ändert, je nachdem, ob du `word`- oder `WORD`-Bewegungen verwendest oder auf Satzebene navigierst.

3.1.6. Warum diese Navigation so mächtig ist

Wort- und Satzbewegungen kombinieren strukturelle Informationen mit dem Operator–Motion-Konzept. Statt nur Zeichen zu zählen, arbeitest du mit *Bedeutungs-*

einheiten: Wörtern, Blöcken (WORDs) und Sätzen. Dadurch lassen sich komplexe Operationen in einer einzigen, reproduzierbaren Tastenkombination ausdrücken.

Wer diese Navigation sicher beherrscht, kann Text extrem effizient manipulieren, ohne visuelle Selektion oder Mausgebrauch. Gerade in Kombination mit Operatoren (d, c, y) und Counts (3 w, 2 W, 3)) entsteht ein sehr dichtes, aber dennoch logisches „Vokabular“ für Bearbeitungsaktionen.

Übungen

- Lege eine Zeile mit gemischten Tokens an, z. B.:

  ```
  user.name = foo_bar, /usr/local/bin/script.sh
  --opt=val;
  ```

  Vergleiche systematisch d w mit d W an verschiedenen Positionen. Notiere dir, welche Bereiche jeweils verschwinden.
- Schreibe zwei bis drei Sätze Fließtext untereinander. Spiele Kombinationen wie c), d), y) an verschiedenen Cursorpositionen durch. Achte darauf, wie VIM Satzgrenzen interpretiert (insbesondere bei Abkürzungen).
- Kombiniere Counts mit Wort- und Satzbewegungen, z. B.: 3 d w, 2 d W, 2). Überprüfe, ob das Ergebnis genau deiner Erwartung entspricht und passe deine mentale Modelle an, falls es Abweichungen gibt.

3.2. Absatz- und Dateinavigation

Zusätzlich zu Wort- und Satzbewegungen bietet VIM präzise Navigation über Absätze und Bildschirmbereiche hinweg. Diese Bewegungen sind besonders wichtig für Makros und strukturierte Bearbeitungen, weil sie unabhängig von konkreter Zeilenlänge oder Formatierung funktionieren.

3.2.1. Absatznavigation ({ und })

VIM erkennt Absätze als Textblöcke, die durch mindestens eine Leerzeile voneinander getrennt sind. Die Navigation orientiert sich dabei ausschließlich an diesen Leerzeilen, nicht an Einrückung oder Formatierung.

- } springt zum Anfang des *nächsten* Absatzes.
- { springt zum Anfang des *vorherigen* Absatzes.

Diese Bewegungen sind besonders robust für Makros, weil sie nicht von Zeilen- oder Wortlängen abhängen. Statt einzelne Zeilen zu zählen, arbeitest du mit ganzen Blöcken.

Typische Kombinationen:

- **d** **}** löscht ab der aktuellen Cursorposition bis zum Anfang des nächsten Absatzes.
- **y** **{** kopiert den vorangegangenen Absatz.
- **c** **}** ändert den Rest des aktuellen Absatzes bis zum nächsten Absatzanfang.

Gerade in Fließtext, Logs oder Konfigurationsblöcken mit Leerzeilen als Trenner sind diese Bewegungen deutlich stabiler als reine Zeilensprünge.

Übungen

- Lege drei bis vier kurze Absätze an, jeweils durch eine Leerzeile getrennt. Bewege den Cursor mit **{** und **}** durch den Text und prüfe, ob VIM die Absatzgrenzen so interpretiert, wie du es erwartest.
- Starte in der Mitte eines Absatzes und probiere **d** **}** und **c** **}** aus. Beobachte genau, welcher Bereich verschwindet bzw. zur Eingabe freigegeben wird.
- Kopiere mit **y** **{** einen Absatz und füge ihn mehrfach mit **p** ein. Kontrolliere, ob die Absatzstruktur (Leerzeilen als Trenner) erhalten bleibt und wie sich das auf die Navigation mit **{** und **}** auswirkt.

3.2.2. Navigation innerhalb des sichtbaren Bereichs

VIM kann relativ zum aktuell sichtbaren Fensterbereich navigieren. Statt in Zeilennummern zu denken, orientierst du dich hier an der Bildschirmhöhe.

- **H** springt zur obersten sichtbaren Zeile des aktuellen Fensters.
- **M** springt zur Zeile in der Mitte des Fensters.
- **L** springt zur untersten sichtbaren Zeile des Fensters.

Diese Bewegungen sind besonders nützlich, wenn du nach dem Scrollen einen bestimmten Textbereich schnell an den Anfang, in die Mitte oder ans Ende des Fensters holen willst. In Kombination mit Operatoren (**d** **H**, **y** **L** etc.) lassen sich so komplette Bildschirmbereiche gezielt bearbeiten, ohne die exakten Zeilennummern zu kennen.

Übungen

- Öffne eine Datei mit mehr Zeilen, als auf den Bildschirm passen. Scrolle mit Ctrl-f und Ctrl-b und teste anschließend H, M, L. Beobachte, wie sich die Cursorposition relativ zum sichtbaren Fenster verändert.
- Kombiniere Operatoren mit Fensterbewegungen:
  - y H kopiert vom aktuellen Cursor bis zur obersten sichtbaren Zeile.
  - d L löscht vom Cursor bis zur untersten sichtbaren Zeile.

  Spiele diese Kombinationen auf einem Testausschnitt durch und prüfe, ob der betroffene Bereich deiner Erwartung entspricht.
- Aktiviere Splits (z. B. mit `:vsplit` oder `:split`) und wiederhole die Übungen in beiden Fenstern. Achte darauf, dass H, M, L immer nur den jeweils aktiven Split betreffen.

3.2.3. Scrollbasierte Navigation

- z t positioniert die aktuelle Zeile am oberen Rand des Fensters (*top*).
- z z zentriert die aktuelle Zeile im Fenster.
- z b positioniert die aktuelle Zeile am unteren Rand des Fensters (*bottom*).

Im Gegensatz zu H, M, L verändern diese Befehle den sichtbaren Bereich, *ohne* die logische Cursorposition in der Datei zu ändern: Die aktuelle Zeile bleibt dieselbe, wird aber im Fenster nach oben, mittig oder nach unten verschoben. Damit eignen sie sich besonders für Makros, bei denen der Cursor auf derselben Zeile bleiben soll, der Kontext drumherum aber anders eingerahmt wird.

Übungen

- Öffne eine längere Datei und platziere den Cursor irgendwo in der Mitte des Textes. Wechsle mehrfach zwischen z t, z z, z b und beobachte, wie sich nur der Bildausschnitt verändert, nicht aber die Cursorzeile.
- Kombiniere Scrollbefehle mit Fensterbewegungen:
  - z t → M: aktuelle Zeile zunächst nach oben scrollen, dann zur Mitte springen.
  - z z → H: Zeile zentrieren und danach an den oberen Fensterrand springen.

  Prüfe, wie sich die Kombination auf Lesbarkeit und Orientierung auswirkt.

- Zeichne ein kurzes Makro auf, das:
  1. die aktuelle Zeile mit **z** **z** zentriert,
  2. eine kleine Änderung (z. B. **A** Kommentar) vornimmt,
  3. mit **Ctrl**-**d** eine halbe Seite weiterscrollt.

  Spiele das Makro mehrmals ab und achte darauf, dass du immer strukturiert von Block zu Block arbeitest, ohne die Zeilennummern zu kennen.

3.2.4. Dateiweite Navigation

Für grobe Sprünge an feste Positionen in der Datei stellt VIM einige zentrale Befehle bereit:

- **g** **g** springt zur *ersten Zeile* und dort in die *erste Spalte*.
- **G** springt zur *letzten Zeile* und dort in die *erste Spalte* (nicht zwingend ans physische Dateiende).
- `<n>` **G** springt zur Zeile `<n>` und setzt den Cursor in die erste Spalte.

Diese Bewegungen sind besonders wichtig, wenn du gezielt an bekannte Zeilennummern springen möchtest (z. B. Fehlermeldungen, Compiler-Output, Logreferenzen).

Hinweis zur Präzision

- **G** **$** springt zur letzten Zeile *und* an deren Ende.
- **g** **g** **0** springt an den absoluten Anfang der Datei (erste Zeile, erste Spalte).

Damit kannst du sehr genau steuern, ob du nur an den Beginn einer Zeile oder wirklich an das letzte Zeichen der Datei springen willst.

Übungen

- Aktiviere Zeilennummern mit `:set number` und springe gezielt zu mehreren unterschiedlichen Zeilen, z. B. **1** **0** **G**, **4** **2** **G**, **1** **0** **0** **G**. Kontrolliere jeweils, ob der Cursor wirklich in der ersten Spalte der gewünschten Zeile landet.
- Navigiere mit **g** **g**, **G**, **G** **$**, **g** **g** **0** durch eine längere Datei und beobachte genau, wie sich die Position des Cursors unterscheidet (Beginn der Zeile vs. Zeilenende vs. absoluter Dateianfang).
- Zeichne ein kurzes Makro auf, das:

1. mit **g** **g** zum Dateianfang springt,
2. ein Suchmuster mit `:/pattern` ausführt,
3. nach dem Treffer mit **G** ans Dateiende springt,
4. dort eine kleine Markierung (z. B. Kommentar) anhängt.

Spiele das Makro mehrfach ab und prüfe, ob die Sprünge unabhängig von der aktuellen Position immer reproduzierbar sind.

3.2.5. Warum diese Navigation besonders makrotauglich ist

Absatz- und Dateisprünge sind nicht von sichtbarer Formatierung abhängig. Bewegungen wie Absatzsprünge (`{` / `}`), seitenweises Scrollen (**Ctrl**-**f**, **Ctrl**-**b**, **Ctrl**-**d**, **Ctrl**-**u**) oder Dateiweite Sprünge (**g** **g**, **G**, `<n>` **G**) funktionieren zuverlässig in:

- Logfiles
- Konfigurationsdateien
- Quellcode
- Markdown- und LaTeX-Dokumenten
- beliebig formatierten Texten

Für Makros ist das entscheidend: Diese Bewegungen hängen weder von konkreten Zeilenumbrüchen noch von Satzlängen oder wechselnder Formatierung ab. Sie adressieren immer dieselben strukturellen Ankerpunkte (Absatzanfang, Dateianfang, Dateiende, Bildschirmoberkante etc.). Dadurch bleiben Makros reproduzierbar und stabil, selbst wenn sich der Textumfang oder die Aufteilung in Zeilen später ändert.

Übungen

- Lege eine Datei mit mehreren klar getrennten Absätzen an (Blöcke durch Leerzeilen getrennt). Zeichne ein Makro auf, das:
  1. vom aktuellen Cursor mit `}` zum nächsten Absatzanfang springt,
  2. dort eine Markierung (z. B. `TODO:`) einfügt,
  3. dann mit `}` zum nächsten Absatz geht.

  Spiele das Makro mehrfach ab und beobachte, dass es unabhängig von Zeilenlängen zuverlässig von Absatz zu Absatz arbeitet.
- Erzeuge eine längere Datei (z. B. Logauszug) und zeichne ein Makro auf, das:
  1. mit **g** **g** zum Dateianfang springt,
  2. mit **Ctrl**-**f** eine Seite weiterblättert,
  3. am oberen Rand (**H**) eine Markierung oder eine Suche ausführt.

Ergänze oder lösche anschließend Zeilen im Log und prüfe, ob das Makro weiterhin stabil läuft.

- Kombiniere Absatzsprünge und Dateiweite Navigation:
  - **g** **g** →mehrere }-Sprünge → **G**.

  Überlege dir ein eigenes Makro, das in jedem Absatz eine kleine Transformation durchführt (z. B. eine Präfix-Zeichenkette einfügt) und dann zum nächsten Absatz weiterläuft.

3.3. Scrollbefehle

Scrollbefehle steuern die Bewegung des sichtbaren Textbereichs im Fenster. Im Unterschied zu Navigationsbefehlen bewegen Scrollkommandos entweder *nur den Bildausschnitt* oder *Cursor und Bild gemeinsam*. Für Makros ist diese Unterscheidung essenziell, weil sich sonst bei Layouteänderungen (Fensterhöhe, Splits, Statuszeilen) die Cursorposition unvorhersehbar verschieben kann.

In diesem Abschnitt betrachten wir zuerst die Scrollbefehle, die den Cursor mitbewegen. Im Anschluss werden die Varianten betrachtet, die nur den sichtbaren Bereich verschieben und den Cursor an seiner Position belassen.

3.3.1. Scrollen mit Cursorbewegung

Diese Befehle verschieben den sichtbaren Bereich und bewegen den Cursor gleichzeitig. Sie sind für das *manuelle* Lesen und Überfliegen großer Dateien sehr angenehm, für Makros jedoch nur eingeschränkt geeignet.

- **Ctrl**-**f** eine Bildschirmseite vorwärts scrollen
- **Ctrl**-**b** eine Bildschirmseite rückwärts scrollen
- **Ctrl**-**d** eine halbe Seite vorwärts scrollen
- **Ctrl**-**u** eine halbe Seite rückwärts scrollen

Diese Befehle orientieren sich am aktuellen Fensterlayout, nicht am Textinhalt. Sie hängen daher von Faktoren wie Terminalgröße, Statuszeilen, Splits oder Folds ab. In interaktiven Sessions ist das komfortabel, in Makros kann es dagegen zu scheinbar „zufälligen“ Cursorpositionen führen, sobald sich die Umgebung ändert.

Typischer Einsatzbereich: Große Logfiles überfliegen, in langen Dokumenten schnell blättern oder sich einen visuellen Überblick verschaffen, ohne konkrete Zielpositionen im Text zu adressieren.

Übungen

- Öffne eine längere Datei (z. B. ein Log oder dieses Handbuch) und wechsle zwischen **Ctrl**-**f**, **Ctrl**-**b**, **Ctrl**-**d**, **Ctrl**-**u** hin und her. Beobachte:
  - Wie weit sich der Text tatsächlich bewegt.
  - Wie sich die Cursorposition relativ zum Fenster verhält.
- Zeichne ein Makro auf, das mit **Ctrl**-**f** eine Seite weiterblättert und dann eine Operation auf der aktuellen Zeile ausführt (z. B. ein Zeichen einfügt oder die Zeile mit einem Kommentarzeichen versieht). Ändere danach die Fensterhöhe oder füge Zeilen ein und spiele das Makro erneut ab. Notiere dir, an welchen Stellen das Makro nicht mehr das tut, was du erwartet hast.
- Vergleiche dieses Verhalten mit einem Makro, das sich nur über inhaltsbezogene Bewegungen (**g** **g**, **G**, Absatzsprünge { / }, Suchen) orientiert. Beurteile, welche Variante für wiederholbare Automatisierung stabiler ist.

3.3.2. Scrollen ohne Cursorbewegung

Im Gegensatz zu den zuvor behandelten Befehlen verändern die folgenden Kommandos *nur* den sichtbaren Bereich, der Cursor bleibt dabei auf derselben Textposition. Das ist besonders hilfreich, wenn du die aktuelle Zeile beibehalten möchtest, aber mehr Kontext nach links oder rechts sehen willst.

- **z** **H** scrollt den Text eine halbe Bildschirmbreite nach links
- **z** **L** scrollt den Text eine halbe Bildschirmbreite nach rechts

Diese horizontale Scrollgruppe ist im normalen Fließtext eher selten erforderlich, wird aber in Praxisumgebungen wichtig, in denen lange Zeilen vorkommen, zum Beispiel:

- Logfiles mit sehr breiten Einträgen
- Tabellen mit vielen Spalten
- Code mit langen Funktionssignaturen oder verschachtelten Aufrufen
- Konfigurationsdateien mit vielen Parametern in einer Zeile

In Kombination mit deaktiviertem automatischem Zeilenumbruch (`:set nowrap`) erlauben diese Befehle einen kontrollierten Blick auf breite Zeilen, ohne den Cursor aus seiner Arbeitsposition zu reißen.

Übungen

- Erzeuge eine künstlich lange Zeile, z. B. durch wiederholtes Anhängen von Text oder durch Einfügen eines langen Logeintrags. Deaktiviere den automatischen Zeilenumbruch mit `:set nowrap` und nutze z H und z L, um die Zeile nach links und rechts zu scrollen. Achte darauf, dass der Cursor an derselben Textposition bleibt.
- Vergleiche das Verhalten von z H / z L mit horizontalen Bewegungen wie 0, $ oder w / W. Notiere dir den Unterschied zwischen *Cursorbewegung* und *Fensterscrollen.*
- Zeichne ein Makro auf, das eine Operation auf der aktuellen Spalte ausführt (z. B. ein Zeichen ersetzt) und zwischendurch den sichtbaren Bereich mit z L verschiebt. Spiele das Makro an verschiedenen Fensterbreiten ab und beobachte, dass die Cursorposition stabil bleibt, obwohl sich der sichtbare Ausschnitt ändert.

3.3.3. Ausrichten der aktuellen Zeile im Fenster

Diese Befehle verschieben den sichtbaren Bereich so, dass die aktuelle Cursorzeile eine bestimmte Position im Fenster einnimmt. Die *Textposition* des Cursors (Zeile/Spalte) bleibt dabei unverändert; nur der Fensterausschnitt wird verschoben.

- z t setzt die aktuelle Zeile an den oberen Fensterrand (*top*)
- z z zentriert die aktuelle Zeile im Fenster
- z b setzt die aktuelle Zeile an den unteren Fensterrand (*bottom*)

Diese Befehle sind besonders makrotauglich, weil sie eine definierte visuelle Orientierung herstellen, ohne die logische Cursorposition im Text zu verändern. So bleiben Makros reproduzierbar, auch wenn sich Fenstergröße oder Schriftgröße ändern.

Übungen

- Öffne eine mehrseitige Datei und positioniere den Cursor irgendwo im mittleren Bereich. Wechsle mehrfach zwischen z t, z z und z b. Beobachte, dass der Cursor im Text identisch bleibt, sich aber seine *Position im Fenster* ändert.
- Zeichne ein Makro auf, das zuerst z z ausführt und dann eine einfache Operation (z. B. A // TODO Esc). Spiele dieses Makro an unterschiedli-

chen Bildschirmhöhen ab (kleines vs. großes Terminalfenster) und überprüfe, dass die Operation trotzdem korrekt auf derselben Zeile landet.
- Kombiniere vertikale Bewegungen (Ctrl-f, Ctrl-b, j, k) mit anschließender Ausrichtung über z t / z z / z b. Notiere dir, in welchen Situationen du welche Ausrichtung bevorzugst (z. B. Kontext oberhalb vs. unterhalb der aktuellen Zeile sehen).

3.3.4. Scrollen in Verbindung mit relativer Navigation

Für präzisere Scrollbewegungen lassen sich Counts mit den Scrollbefehlen kombinieren:

- 3 Ctrl-d scrollt drei halbe Seiten vorwärts
- 5 Ctrl-u scrollt fünf halbe Seiten zurück

Die tatsächliche Zeilenanzahl pro Ctrl-d / Ctrl-u hängt von der Fensterhöhe ab. Über den Count skalierst du diese Bewegung, ohne ein eigenes Makro schreiben zu müssen. Das ist besonders hilfreich in sehr großen Dateien wie Logfiles oder Backtraces.

Übungen

- Öffne eine große Logdatei und vergleiche Ctrl-d mit 3 Ctrl-d. Achte darauf, wie viel Kontext du jeweils noch siehst und welche Variante sich für die Fehlersuche besser anfühlt.
- Zeichne ein Makro auf, das zunächst mit 2 Ctrl-d nach unten scrollt und dann eine einfache Operation (z. B. n für die nächste Suchtrefferstelle) ausführt. Spiele das Makro mehrfach ab und beobachte, ob die Kombination aus Count und Scrollen dein Navigationsmuster sinnvoll beschleunigt.
- Variiere die Terminalgröße (kleines vs. großes Fenster) und prüfe, ob 3 Ctrl-d bzw. 5 Ctrl-u sich noch kontrolliert anfühlen oder ob du lieber kleinere Counts verwendest. Notiere dir eine für dich passende Standardkombination.

3.3.5. Makrohinweise zum Scrollen

Scrollbefehle können Makros unberechenbar machen, wenn der Bildschirm in einer anderen Größe oder auf einem anderen Terminal dargestellt wird. Um Makros stabil zu halten, gilt grundsätzlich:

- j, k, w, $ sind robust und unabhängig vom Fensterlayout.
- Ctrl-f, Ctrl-b, Ctrl-d, Ctrl-u hängen stark von Fenstergröße und Schriftbild ab.
- z t, z z, z b sind robust, weil der Cursor an seiner Position bleibt und nur der Bildausschnitt ausgerichtet wird.

Besonders wichtig ist:

> **Makros sollten nach Möglichkeit keine seiten- oder halbseitenweisen Scrollbefehle wie Ctrl-f, Ctrl-b, Ctrl-d, Ctrl-u enthalten.**

Sie funktionieren zwar, aber der Bildausschnitt auf einem anderen System, Terminal oder in einem anders großen Editorfenster kann zu völlig anderem Verhalten führen. Makros, die auf solchen Befehlen basieren, brechen oft genau dann, wenn man sie auf einem anderen Setup am dringendsten bräuchte.

Übungen

- Nimm ein einfaches Makro auf, das Ctrl-f verwendet (z. B. q q Ctrl-f j q) und spiele es in derselben Datei bei unterschiedlicher Fenstergröße ab. Beobachte, wie sich das Verhalten ändert.
- Erzeuge ein alternatives Makro, das statt Ctrl-f mit einer Kombination aus n, j, k, w arbeitet und die gleiche logische Aufgabe erfüllt (z. B. zum nächsten Treffer springen und eine Zeile bearbeiten). Vergleiche die Stabilität beider Varianten.
- Nutze z t oder z z in einem Makro, um vor einer Bearbeitung den Cursor sichtbar auszurichten, ohne seine Position zu verändern. Prüfe, ob das Makro sich auf verschiedenen Terminalgrößen gleich verhält.

3.3.6. Zusammenfassung

Scrollbefehle in VIM trennen klar zwischen Sichtbewegung und Cursorbewegung. Für Makros gelten strenge Stabilitätsanforderungen: Befehle wie Ctrl-f,

Ctrl-b, Ctrl-d, Ctrl-u sind zwar bequem für das manuelle Überfliegen großer Dateien, führen in Makros aber leicht zu unvorhersehbaren Cursorpositionen.

Zentrierende Befehle wie zz, zt, zb sind oft die bessere Wahl, weil sie den Cursor nicht verschieben, sondern nur den Bildausschnitt anpassen. Für interaktives Arbeiten bieten die seitenweisen Scrollbefehle hingegen eine sehr effiziente Möglichkeit, schnell durch große Dateien wie Logfiles, Backtraces oder lange Konfigurationsdateien zu navigieren.

Wer bei Makros bewusst auf robuste Bewegungen (j, k, w, $, Absatz- und Wortnavigation) setzt und fensterabhängige Scrollkommandos nur gezielt einsetzt, erhält zuverlässige, wiederverwendbare Automatisierungen.

3.4. Klammernavigation

Klammernavigation ist ein wesentlicher Bestandteil beim Arbeiten mit strukturiertem Text, insbesondere in Programmcode, LaTeX, JSON, Markdown oder Konfigurationsdateien. VIM bietet präzise Bewegungen, um zwischen zusammengehörigen Klammern zu springen oder ganze Blockbereiche sicher zu bearbeiten. In Kombination mit Textobjekten (i(, a{ usw.) entstehen sehr mächtige Operator–Motion-Kombinationen.

3.4.1. Springen zur passenden Klammer (%)

Der Befehl % springt zur jeweils passenden, korrespondierenden Klammer. Unterstützt werden die folgenden Paare:

- ( )
- { }
- [ ]

Der Cursor muss dabei auf einer der beiden korrespondierenden Klammern stehen. Bei Ausführung von % springt VIM exakt zur passenden Gegenseite. Befindet sich die Datei in einem syntaktisch konsistenten Zustand, kannst du so schnell durch verschachtelte Strukturen navigieren.

Besonders stark wird % in Verbindung mit Operatoren und Textobjekten:

- d% löscht vom Cursor bis zur passenden Klammer.
- y% kopiert den Inhalt inklusive beider Klammern.

- **c** **%** ändert alles zwischen aktueller und passender Klammer.

Übungen

- Erzeuge einen verschachtelten Ausdruck, z. B.:

  ```
  if (foo(bar[ i ])) { return baz(i); }
  ```

  Setze den Cursor nacheinander auf die verschiedenen Klammern `(`, `[`, `{` und nutze **%**, um zwischen den Paaren hin- und herzuspringen. Beobachte, wie VIM auch bei Verschachtelung die korrekten Partner findet.
- Wiederhole die Übung mit bewusst „schräg“ formatiertem Code (Einrückungen, Leerzeilen, lange Zeilen). Prüfe, ob **%** weiterhin zuverlässig die passenden Paare ansteuert.
- Teste Operator-Kombinationen:
  - **d** **%** auf einer öffnenden Klammer `(` → der gesamte umklammerte Block verschwindet.
  - **y** **%** auf einer öffnenden `{` → der komplette Block wird in ein Register kopiert.
  - **c** **%** auf einer öffnenden `(` → der umklammerte Ausdruck wird durch neuen Text ersetzt.

  Spiele diese Varianten mit unterschiedlichen Klammerarten durch, bis du sicher im Kopf hast, welchen Bereich **%** genau umfasst.
- Kombiniere **%** mit Textobjekten, z. B. in einer Zeile wie:

  ```
  result = foo(bar(baz));
  ```

  Nutze **v** **i** **(**, **v** **a** **(**, **d** **i** **(** und danach **%**, um zu verstehen, wann Textobjekte und wann die reine Klammernavigation sinnvoller sind.

Besonderheiten

- **%** funktioniert auch in tief verschachtelten Strukturen.
- Die Treffergenauigkeit steigt, wenn eine passende Syntaxdatei geladen ist (z. B. für C, JSON, Lua).
- Ohne oder mit unvollständiger Syntax kann VIM nur einfache Zeichenpaare matchen; komplexere Sprachkonstrukte werden dann ggf. nicht korrekt erkannt.
- Die Funktion arbeitet unabhängig von Leerzeichen, Einrückungen und Zeilenumbrüchen – Layoutänderungen ändern das Matching nicht.

3.4.2. Navigation im Klammerkontext

Auch ohne % existieren Bewegungen, die im Umfeld von Klammern sehr hilfreich sind. Sie springen zu strukturell relevanten Stellen im Code und sind besonders nützlich in größeren Dateien:

- [(springt rückwärts zur vorherigen *öffnenden* Klammer
-]) springt vorwärts zur nächsten *schließenden* Klammer
- [{ springt rückwärts zum vorherigen Blockbeginn
-] } springt vorwärts zum nächsten Blockende

Diese Bewegungen sind vor allem in Codeblöcken hilfreich, in denen viele Klammern auftreten und die visuelle Struktur schwer zu überblicken ist. Sie lassen sich mit Operatoren kombinieren, etwa:

- d] löscht vom Cursor bis zur nächsten schließenden Klammer
- y [{ kopiert vom Cursor bis zum vorherigen Blockbeginn

Übungen

- Erzeuge einen Beispielcode mit mehreren Blöcken, z. B.:

```
if (cond) {
while (x[ i ] <max) {
do_something(i);
}
}
```

  Setze den Cursor an verschiedene Stellen im Block und nutze [(,]), [{ und] }, um zu den jeweiligen Block- oder Klammergrenzen zu springen. Notiere dir, welche Bewegungen welche Strukturebene ansteuern.
- Kombiniere die Bewegungen mit Operatoren:
  - d [{ löscht vom Cursor bis zum Beginn des umgebenden Blocks.
  - y] } kopiert vom Cursor bis zum Ende des nächsten Blocks.

  Spiele diese Kombinationen in mehreren unterschiedlich eingerückten Codebeispielen durch.
- Öffne eine realistische Datei (z. B. eine vorhandene Konfigurationsdatei oder ein LaTeX-Dokument) und nutze nur %, [(,]), [{ und] }, um dich von Block zu Block zu bewegen. Achte darauf, wie gut sich logische Strukturen allein über diese Bewegungen erfassen lassen.

3.4.3. Klammerobjekte (für Operatoren)

Klammern lassen sich in VIM als Textobjekte verwenden. Dadurch können Operatoren wie d, c oder y exakt auf den Inhalt eines Blocks angewendet werden – wahlweise nur auf den inneren Inhalt oder auf den gesamten Bereich inklusive Klammern.

- i (innerer Inhalt von `( ... )`
- a (Block inklusive der runden Klammern
- i { innerer Block einer geschweiften Klammer
- a { Block inklusive der geschweiften Klammern
- i [innerer Inhalt eines eckigen Blocks
- a [Block inklusive der eckigen Klammern

Diese Textobjekte sind ideal für Operator–Motion-Kombinationen wie:

- d i (löscht nur den Inhalt der runden Klammer
- c i { ersetzt den Inhalt einer geschweiften Klammer
- y a [kopiert einen eckigen Block inklusive Klammern

Übungen

- Lege eine Zeile an wie:

  ```
  printf("Wert: %d (OK)", value);
  ```

  Setze den Cursor in verschiedene Bereiche und teste: d i (, c i (, y a (. Beobachte genau, was jeweils als „innerer Inhalt“ behandelt wird.
- Erzeuge einen verschachtelten Block:

  ```
  if (cond) {
  do_something(arr[i]);
  }
  ```

  – Setze den Cursor irgendwo innerhalb des Blocks und probiere d i { vs. d a {.
  – Achte darauf, dass d i { nur den Inhalt löscht, während d a { auch die Klammern entfernt.
- Nimm eine reale Code- oder Konfigurationsdatei und arbeite bewusst nur mit Klammerobjekten: c i (, c i { und c i [. Ziel: Du änderst

Werte und Ausdrücke, ohne jemals mühsam Zeichenweise zu navigieren oder visuell zu selektieren.

3.4.4. Klammern in Makros

Beim Arbeiten mit Makros ist Klammernavigation besonders wertvoll, weil:

- % immer exakt zur passenden Gegenklammer springt,
- Textobjekte wie i (unabhängig von Zeilenumbrüchen funktionieren,
- Makros dadurch strukturell stabil bleiben, selbst wenn sich der Text zwischen Aufzeichnung und späterer Ausführung verändert.

Ein Makro, das mit Pfeiltasten oder Zeichenbewegungen (h, l) aufgezeichnet wird, bricht leicht, sobald sich die Zeilenlänge ändert. Ein Makro, das stattdessen % oder Textobjekte wie i (nutzt, bleibt stabil — selbst bei komplett veränderter Formatierung.

Übungen

- Lege mehrere Zeilen mit Funktionsaufrufen an, z. B.:

  ```
  log_info("Start", code);
  log_info("Weiter", value);
  log_info("Ende", status);
  ```

  Nimm ein Makro auf, das:
  1. zur öffnenden Klammer springt (f (oder Cursor passend setzen),
  2. mit c i (den Klammerinhalt ändert,
  3. mit @ q (oder einem anderen Register) erneut ausgeführt wird.

  Ändere danach die Länge einzelner Zeilen (z. B. extra Parameter) und prüfe, ob das Makro mit c i (weiterhin korrekt arbeitet.
- Zeichne zum Vergleich ein zweites Makro auf, das nur mit h, l oder Pfeiltasten in die Klammer navigiert. Verschiebe oder verlängere anschließend die Zeile und teste das Makro erneut. Beobachte, wie schnell solche Makros „daneben greifen“.

3.4.5. Zusammenfassung

Klammernavigation in VIM ist ein zentraler Bestandteil strukturellen Arbeitens. Während `%` für das schnelle Springen zwischen passenden Klammern zuständig ist, bieten Textobjekte wie `i` `(` und `a` `{` mächtige Werkzeuge für präzise Bearbeitung.

Besonders für Makros sind diese Bewegungen unverzichtbar, weil sie unabhängig von Layout, Zeilenumbrüchen und variabler Struktur funktionieren. Wer Klammernavigation konsequent mit Operator–Motion kombiniert, erhält robuste, wiederverwendbare Bearbeitungsschritte statt fragiler Einmal-Aktionen.

4. Editieren (Operatoren)

4.1. Löschen (d)

Der Operator **d** ist einer der zentralen Operatoren in VIM. Er löscht den Bereich, der durch eine Motion oder ein Textobjekt definiert wird. Da sich **d** frei mit Bewegungen kombinieren lässt, entstehen sehr präzise Löschoperationen. Diese Modularität macht **d** besonders makrotauglich – du musst nur den Zielbereich per Bewegung beschreiben, nicht umständlich visuell markieren.

4.1.1. Grundprinzip

Das Muster folgt immer dem Operator–Motion–Konzept:

d + `Motion`

Beispiele:

- **d** **w** löscht vom Cursor bis zum Anfang des nächsten Wortes
- **d** **e** löscht bis zum Ende des aktuellen Wortes
- **d** **$** löscht bis zum Zeilenende
- **d** **0** löscht bis zum absoluten Zeilenanfang
- **d** **}** löscht bis zum Anfang des nächsten Absatzes

Der eigentliche Löschoperator (**d**) bleibt immer gleich – nur die Motion bestimmt, wie groß und wie «geschnitten» der zu löschende Bereich ist. Genau das macht **d** so gut mit Makros kombinierbar: Änderst du später den Text, aber nicht die Struktur der Bewegungen, funktionieren dieselben **d**+Motion–Kombinationen immer noch.

Übungen

- Lege einen kurzen Absatz an, z. B.:

```
Dies ist ein kurzer Testsatz mit mehreren
Wörtern.
```

Probiere systematisch d w, d e, d b und beobachte, welche Wortteile jeweils verschwinden. Nutze u, um die Änderungen zurückzunehmen.

- Erzeuge mehrere Absätze mit Leerzeilen dazwischen. Teste d }, während der Cursor in der Mitte eines Absatzes steht. Beobachte, wie VIM den Rest des Absatzes bis zur nächsten Leerzeile entfernt.
- Kombiniere Counts mit d, z. B.:
  - 3 d w löscht drei aufeinanderfolgende Wörter,
  - 2 d } löscht bis über zwei Absatzgrenzen hinweg.

  Ziel der Übung: Du kannst aus dem Kopf sagen, welcher Bereich bei einer gegebenen d+Motion–Kombination betroffen ist, bevor du die Taste drückst.

4.1.2. Löschen kompletter Zeilen

Wird der Operator direkt mit einer Zeilenbewegung kombiniert – oder zweimal hintereinander eingegeben – wirkt er auf komplette Zeilen einschließlich ihres Zeilenumbruchs.

- d d löscht die aktuelle Zeile (inklusive Newline)
- 2 d d löscht die aktuelle und die folgende Zeile
- d 2 j löscht die aktuelle Zeile und *zwei weitere* Zeilen darunter (insgesamt drei Zeilen)

Semantisch gilt weiterhin das Operator–Motion–Muster:

- d d ⇒ d + (implizite Zeilenbewegung nach unten)
- 2 d d ⇒ d + 2 j
- d 2 j ⇒ d + 2 j

Wichtig ist, dass VIM bei zeilenweisen Operationen nicht nur den Inhalt, sondern die gesamte Zeile inklusive des Zeilenumbruchs entfernt. Deshalb „rutschen“ nachfolgende Zeilen nach oben.

Übungen

- Erzeuge fünf nummerierte Zeilen:

```
Zeile 1
Zeile 2
```

```
Zeile 3
Zeile 4
Zeile 5
```

Stelle den Cursor auf „Zeile 3“ und führe nacheinander d d, u, 2 d d, u, d 2 j aus. Notiere dir jeweils, welche Zeilen verschwinden.

- Wiederhole die Übung in einer Datei mit Leerzeilen zwischen den Textzeilen. Beobachte, dass d d immer die komplette Zeile samt Leerzeilenstatus entfernt – niemals nur „bis zum Textende“.
- Nimm ein kurzes Makro auf, das mit d d arbeitet (z. B. „lösche jede zweite Zeile“). Spiele es mehrmals ab und prüfe, ob das Ergebnis der erwarteten Struktur entspricht. Ziel: Du kannst die Wirkung von zeilenweisen Löschungen im Kopf vorwegnehmen.

4.1.3. Löschen mit Textobjekten

Textobjekte bieten besonders präzise Steuerung, weil sie ganze semantische Einheiten (Wörter, Klammerblöcke, Absätze) beschreiben, anstatt nur rohe Zeichenbereiche.

Typische Kombinationen mit dem Operator d:

- d i w löscht das aktuelle *innere Wort* (ohne nachfolgenden Leerraum)
- d a w löscht das aktuelle Wort *inklusive* angrenzenden Leerraums
- d i (löscht nur den Inhalt der runden Klammer (...)
- d a { löscht den gesamten Block inklusive der geschweiften Klammern { ... }
- d i p löscht den aktuellen Absatz (innerer Absatz)

Die Kombination d + Textobjekt folgt weiterhin dem Operator–Motion–Muster, nur dass die „Motion“ hier ein strukturelles Objekt ist (Wort, Klammer, Absatz) statt einer reinen Positionsbewegung. Genau das macht diese Befehle für Makros extrem stabil: Sie funktionieren unabhängig von Zeilenumbrüchen, Einrückung oder Zeilenlänge.

Übungen

- Lege eine Zeile wie

```
int current_value = previous_value + 1;
```

an. Setze den Cursor in die Mitte von `current_value` und vergleiche systematisch: d w, d i w, d a w. Notiere dir, was jeweils exakt gelöscht wird.

- Erzeuge ein Konstrukt wie

  ```
  func(foo, (bar + baz), qux);
  ```

  Setze den Cursor in den inneren Klammerausdruck `(bar + baz)` und spiele durch: d i (, d a (, d i), d a). Beobachte den Unterschied zwischen „inner“ (i) und „around“ (a).
- Schreibe drei kurze Absätze (jeweils durch eine Leerzeile getrennt). Stelle den Cursor mitten in den zweiten Absatz und führe d i p aus. Prüfe, welche Leerzeilen bestehen bleiben und wie sich die Absatzstruktur verändert.
- Nimm ein Makro auf, das mit d i w oder d a w ein Wort ersetzt (z. B. durch TODO). Spiele es auf mehreren unterschiedlichen Zeilen ab und verifiziere, dass es trotz verschiedener Wortlängen und Einrückungen stabil bleibt.

4.1.4. Löschen mit Counts

Counts steuern, *wie oft* eine Operation ausgeführt wird. Sie können sowohl vor dem Operator als auch vor der Motion stehen – der Effekt ist identisch:

- 3 d w — löscht drei `word`-Bereiche
- d 3 w — gleicher Effekt wie 3 d w
- 5 d d — löscht fünf Zeilen ab der aktuellen
- d 2 } — löscht zwei Absatzblöcke bis zum zweiten nächsten Absatzanfang

Counts sind ein zentraler Baustein für makrotaugliche Befehle, weil sie Durchläufe exakt steuerbar machen, ohne dass Schleifen oder zusätzliche Logik nötig wären.

Übungen

- Erzeuge fünf ähnliche Textzeilen untereinander. Setze den Cursor in die erste Zeile und vergleiche: d d, 2 d d, 5 d d. Beobachte, wie sich die Anzahl der gelöschten Zeilen verändert.
- Schreibe eine längere Zeile mit mehreren Wörtern. Teste: 3 d w und d 3 w. Vergewissere dich, dass beide Befehle dasselbe Ergebnis liefern.

- Lege drei Absätze an (jeweils durch Leerzeilen getrennt) und setze den Cursor in den ersten Absatz. Führe **d** **}** und anschließend **d** **2** **}** aus. Vergleiche den Unterschied im gelöschten Umfang.
- Nimm ein Makro auf, das mit einem Count arbeitet, z. B.: **3** **d** **w** gefolgt von **i** TODO **Esc** Spiele das Makro auf mehreren Zeilen ab und prüfe, ob es überall den erwarteten Bereich ersetzt.

4.1.5. Löschen bis zu Suchergebnissen

Statt eine feste Motion wie **w** oder **$** zu verwenden, kannst du **d** auch mit einer Suche kombinieren. Die Suche fungiert dann als „bewegliches Ziel“:

- **d** **/** `string` löscht vom Cursor bis *vor* den nächsten Treffer von `string`
- **d** **?** `string` löscht vom Cursor rückwärts bis *vor* den vorherigen Treffer

Wichtig:

- Das Suchmuster selbst bleibt erhalten – gelöscht wird nur der Bereich *bis unmittelbar davor*.
- Wird kein Treffer gefunden, bricht VIM mit einer Fehlermeldung (z. B. `E486: Pattern not found`) ab und löscht nichts.
- Das Suchmuster folgt den normalen Regeln der **/**- bzw. **?**-Suche (inklusive regulärer Ausdrücke).

Diese Technik ist besonders nützlich in Logfiles, Konfigurationsdateien oder Code, wenn du bis zu einem strukturellen Marker (z. B. einem Keyword, einem Timestamp oder einer Klammer) löschen möchtest, ohne die Markierung selbst anzutasten.

Übungen

- Erzeuge eine Testzeile wie:

  ```
  INFO start ... DEBUG check ... ERROR stop
  ```

  Setze den Cursor vor `INFO` und teste: **d** **/** DEBUG **Enter**. Kontrolliere, dass alles bis direkt vor DEBUG gelöscht wird.
- Wiederhole die Übung rückwärts: Setze den Cursor auf ERROR und verwende **d** **?** DEBUG **Enter**. Beobachte, welcher Bereich rückwärts entfernt wird.
- Kombiniere **d** **/** `string` mit **.** (Wiederholungsoperator), um in einem Logblock mehrfach bis zum gleichen Marker zu löschen.

- Nutze reguläre Ausdrücke, z. B. d / ERROR\ | WARN Enter in einem Logfile. Prüfe, wie sich das Löschen verhält, wenn mehrere Muster alternativ (\ |) gesucht werden.

4.1.6. Einfügen nach Löschoperationen

Jede Löschoperation mit d oder d d schreibt den entfernten Inhalt in das Standardregister (" "). Über dieses Register kannst du den gelöschten Text direkt wieder einfügen:

- p fügt *nach* dem Cursor ein (bei Zeilenlöschungen: *unterhalb* der aktuellen Zeile)
- P fügt *vor* dem Cursor ein (bei Zeilenlöschungen: *oberhalb* der aktuellen Zeile)

Wird d d verwendet, landen die gelöschten Zeilen vollständig im Standardregister. Ein anschließendes p stellt sie wieder als ganze Zeilen her – inklusive Zeilenumbrüchen. Mit einem Count kannst du Mehrfacheinfügungen steuern, z. B. 3 p für dreifaches Einfügen.

Übungen

- Erzeuge drei Testzeilen, z. B.:

  ```
  Zeile 1
  Zeile 2
  Zeile 3
  ```

  Lösche d d auf „Zeile 2“ und füge mit p wieder ein. Beobachte, dass die Zeile erneut unterhalb der aktuellen Position eingefügt wird.
- Wiederhole die Übung mit P: Lösche eine Zeile mit d d, bewege den Cursor an eine andere Stelle und füge mit P ein. Prüfe, wie sich die Position relativ zum Cursor unterscheidet.
- Kombiniere d d und 3 p: Lösche eine Zeile und füge sie dreimal hintereinander ein. Achte darauf, dass die Reihenfolge der eingefügten Zeilen stabil bleibt.
- Nutze . (Wiederholungsoperator), um das Muster „Zeile löschen und an anderer Stelle wieder einfügen“ mehrfach zu wiederholen. Beobachte, wie sich p und P im Zusammenspiel mit . verhalten.

4.1.7. Makrohinweise für Löschoperationen

Für Makros gelten bei Löschoperationen einige robuste Grundsätze:

- **Nutze Textobjekte**, wenn immer möglich: d i w, d a w, d i p, d i (, d a { usw. Sie sind unabhängig von Zeilenumbrüchen und Einrückung.
- **Vermeide Pfeiltasten** in Makros – sie hängen vom konkreten Bildschirm-Layout ab und machen Makros fragil.
- **Nutze Counts**, wenn der Umfang berechenbar ist: 3 d d, 2 d }, 5 d w statt mehrfacher Einzelaufrufe.
- **Bevorzuge Wort-, Absatz- und Suchbewegungen** gegenüber rein zeilenweisen Löschungen: d w, d W, d }, d / `string` Enter sind meist stabiler als „irgendwie mit j / k hinlaufen und x spammen“.
- **Nutze . konsequent**, um eine einmal definierte Löschoperation exakt zu wiederholen, statt sie im Makro mehrfach manuell zu kodieren.

Gerade d ist ein Paradebeispiel für die Stärke des Operator–Motion–Konzepts: Mit einer einzigen Taste plus Motion oder Textobjekt lassen sich sehr unterschiedliche Bereiche zielgenau entfernen, ohne dass das Makro an konkrete Zeilenlängen gebunden ist.

Übungen

- Nimm ein einfaches Testdokument und zeichne ein Makro auf Register q, das mit d i w das aktuelle Wort löscht und mit l zum nächsten startet. Spiele das Makro mit @ q und @ @ ab und beobachte, wie stabil es auf unterschiedlichen Zeilenlängen bleibt.
- Zeichne ein Makro, das mit d i p einen Absatz löscht und mit j zum nächsten Absatz springt. Teste es in einem Text mit unterschiedlich langen Absätzen. Ersetze dann j durch }, und vergleiche die Stabilität.
- Erzeuge einen Logblock mit mehreren Zeilen, die ERROR oder WARN enthalten. Zeichne ein Makro, das mit d / ERROR Enter bis zum nächsten Fehler löscht und dann mit . wiederholt. Beobachte, wie sich das Makro verhält, wenn du zusätzliche Zeilen einfügst oder entfernst.
- Baue bewusst ein fragiles Makro mit Pfeiltasten (z. B. mit den Cursorpfeilen über mehrere Zeilen laufen und x benutzen) und vergleiche sein Verhalten

mit einem Makro, das die gleichen Löschungen über d w, d W oder Textobjekte durchführt.

4.1.8. Zusammenfassung

Der Operator d ist der zentrale Löschmechanismus in VIM. Durch die Kombination mit Bewegungen, Counts und Textobjekten entsteht ein sehr präzises und flexibles Werkzeug:

- d + Motion (d w, d e, d $, d }) für bereichsorientiertes Löschen,
- d d und Counts (3 d d, 5 d }) für zeilenweise Löschungen,
- Textobjekte wie d i w, d a { oder d i p für semantisch sinnvolle Blöcke,
- Suchbasierte Varianten wie d / `string` Enter und d ? `string` Enter für Logfiles und strukturierte Marker.

In manuellen Sessions sorgt d für schnelles, zielgerichtetes Aufräumen. In Makros bildet er zusammen mit stabilen Bewegungen und Textobjekten das Rückgrat reproduzierbarer, robusten Editierabläufe.

4.2. Ändern (c)

Der Operator c (*change*) kombiniert Löschen mit einem sofortigen Wechsel in den Insert Mode. Er verhält sich strukturell wie d, öffnet aber nach der Motion direkt den Eingabemodus, sodass der entfernte Bereich unmittelbar überschrieben werden kann.

Typisch ist das Muster:

`c + Motion`

also z. B. c w, c $ oder c i (.

4.2.1. Grundprinzip

c arbeitet immer nach dem Operator–Motion–Schema: Du beschreibst zuerst, „was“ passieren soll (c), dann „wo“ es passieren soll (Motion oder Textobjekt). Anschließend wechselt VIM in den Insert Mode.

Beispiele:

- c w ändert das nächste Wortfragment ab der Cursorposition
- c e ändert bis zum Ende des aktuellen `word`
- c $ ändert vom Cursor bis zum Zeilenende
- c 0 ändert vom Cursor bis zum Zeilenanfang

Nach Ausführen der Motion steht der Cursor bereits im Insert Mode; der gelöschte Bereich kann direkt neu eingegeben werden. Mit <Esc> kehrst du anschließend in den Normal Mode zurück.

Hinweis VIM behandelt c w in vielen Situationen praktisch wie c e („change word bis Wortende"). Der genaue Unterschied zeigt sich vor allem an Wortgrenzen und Interpunktion – ideal als Übungsfeld, siehe unten.

4.2.2. Zeilenbasierte Änderungen

Wie bei d gibt es auch bei c zeilenbasierte Varianten:

- c c ändert die gesamte aktuelle Zeile
- 2 c c ändert zwei aufeinanderfolgende Zeilen
- c 2 j ändert die aktuelle Zeile und zwei weitere Zeilen

Intern entspricht c c einer Kombination aus „Zeile leeren" und Wechsel in den Insert Mode. Praktisch ist c c damit die Standardmethode, um eine Zeile komplett neu zu schreiben.

Übungen

- Lege eine Testzeile an, z. B.:

  ```
  foo_bar, baz.qux; end
  ```

  Setze den Cursor auf den Anfang von `foo_bar` und vergleiche c w und c e. Achte darauf, wie viel genau jeweils geändert wird und wo der Cursor im Insert Mode landet.
- Wiederhole die Übung mit einem Wort in der Mitte der Zeile (z. B. `baz`) und untersuche, wie sich c w an Wortgrenzen und vor Interpunktion verhält.
- Teste zeilenbasierte Änderungen:
  - Verwende c c, um eine komplette Zeile durch neuen Text zu ersetzen.

  - Nutze 2 c c auf einer zweizeiligen Logmeldung oder einem Konfigurationsblock und gib anschließend neuen Inhalt ein.
  - Vergleiche c c mit S (Substitute) – beide verhalten sich für ganze Zeilen praktisch identisch.
- Nimm ein Makro auf, das mit c w ein bestimmtes Wortmuster ersetzt (z. B. TODO durch DONE) und mit n zum nächsten Treffer springt. Spiele das Makro mit @ q und @ @ ab und beobachte, wie stabil die Kombination aus c und Motion ist.
- Kombiniere Counts mit c, z. B. 3 c w oder c 3 w, und überprüfe in einer Testzeile, welchen Bereich du tatsächlich überschreibst. Ziel ist, ein Gefühl dafür zu bekommen, ab wann ein Count sinnvoller ist als mehrere einzelne Änderungen.

Technisch entspricht `cc` der Sequenz:

`c^`
(vom ersten Nichtleerzeichen bis zum Zeilenende)

4.2.3. Ändern mit Textobjekten

Textobjekte ermöglichen es, strukturelle Einheiten gezielt zu ersetzen, statt nur rohe Zeichenbereiche zu bearbeiten. c wird dabei mit einem Textobjekt kombiniert:

- c i w ändert das aktuelle „innere“ Wort (ohne umliegenden Leerraum)
- c a w ändert das aktuelle Wort inklusive nachfolgendem Leerraum
- c i (ändert nur den Inhalt einer runden Klammer (...)
- c a { ändert einen gesamten Block inklusive geschweifter Klammern
- c i p ändert den aktuellen Absatz (innerer Absatzblock)

Diese Befehle sind hochgradig makrotauglich, weil sie sich an logischen Strukturen (Wort, Block, Absatz) orientieren und nicht von Zeilenlänge, Einrückung oder Umbrüchen abhängen.

Übungen

- Lege eine Testzeile an, z. B.:

```
int value = foo_bar; /* TODO: fix */
```

Setze den Cursor auf `foo_bar` und vergleiche **c** **i** **w** und **c** **a** **w**. Achte darauf, ob der nachfolgende Leerraum mit ersetzt wird oder nicht.

- Erzeuge eine Klammerkonstruktion wie:

  ```
  log_debug("status=%d", status);
  ```

  Teste nacheinander **c** **i** **(** und **c** **a** **(**. Untersuche, ob nur der Inhalt oder auch die Klammern selbst überschrieben werden.
- Schreibe zwei bis drei Absätze, getrennt durch Leerzeilen. Setze den Cursor irgendwo in einen Absatz und verwende **c** **i** **p**, um den gesamten Absatz durch neuen Text zu ersetzen. Wiederhole das mit **c** **a** **p** und beobachte den Unterschied, falls am Rand zusätzliche Leerzeilen stehen.
- Nimm ein Makro auf, das mit **c** **i** **w** ein bestimmtes Schlagwort (z. B. TODO) ersetzt und anschließend mit **n** zum nächsten Suchtreffer springt. Spiele das Makro an mehreren Stellen ab und prüfe, wie stabil sich **c** **i** **w** in unterschiedlichen Zeilenlängen verhält.

4.2.4. Counts mit c

Wie bei allen Operatoren kannst du auch beim Ändern einen Count voranstellen. Damit steuerst du, wie viele Einheiten in einem Schritt geändert werden:

- **3** **c** **w** ändert die nächsten drei Wortfragmente
- **c** **3** **w** identischer Effekt wie **3** **c** **w**
- **5** **c** **c** ändert fünf aufeinanderfolgende Zeilen

Counts können vor dem Operator oder vor der Motion stehen – semantisch ist das gleich. Wichtig ist nur: Nach Ausführung der Motion wechselt VIM in den Insert Mode und du ersetzt den gelöschten Bereich direkt.

Übungen

- Lege eine Testzeile an wie

  ```
  eins zwei drei vier fünf sechs sieben
  ```

  Setze den Cursor auf `eins` und teste **3** **c** **w**. Ersetze die drei Wörter durch einen neuen Text und prüfe, wie weit VIM tatsächlich löscht.
- Erzeuge drei kurze Zeilen untereinander und setze den Cursor auf die erste. Verwende **3** **c** **c** und tippe einen neuen Satz. Kontrolliere, dass genau drei Zeilen ersetzt wurden.

- Wiederhole dieselben Experimente mit c 3 w statt 3 c w, um zu verinnerlichen, dass Count vor Operator oder Motion denselben Effekt hat.

4.2.5. Unterschied zwischen cw und ce

Obwohl c w und c e auf den ersten Blick ähnlich wirken, gibt es wichtige Unterschiede – vor allem, wenn der Cursor *nicht* am Wortanfang steht.

- c w ändert das *folgende Wort* ab der aktuellen Cursorposition.
- c e ändert bis zum *Ende des aktuellen Wortes*.

Steht der Cursor bereits auf dem ersten Zeichen eines Wortes, verhalten sich c w und c e in vielen Situationen ähnlich, aber der mentale Merksatz ist unterschiedlich:

- c w: „ändere dieses Wort weiter bis zum *nächsten sinnvollen Einstieg*“
- c e: „ändere dieses Wort bis zum *Wortende*“

Sobald der Cursor mitten im Wort steht, zeigt sich der Unterschied deutlicher: c w schneidet ab Cursorposition, c e orientiert sich strikt am Wortende.

Übungen

- Lege eine Testzeile an, z. B.:

  `funktionstest`

  Setze den Cursor bewusst *in die Mitte* des Wortes, etwa auf das `k`, und probiere nacheinander: c w und c e. Notiere dir, ab welcher Position jeweils geändert wird.
- Lege eine zweite Testzeile an:

  `foo bar baz`

  - Setze den Cursor auf das `b` von `bar` und teste c w.
  - Wiederhole mit c e.

  Beobachte, ob und wie sich das Verhalten unterscheidet, insbesondere im Hinblick auf Leerzeichen.
- Wiederhole die Experimente mit aktivierten Counts:
  - 2 c w auf einer Zeile mit mehreren kurzen Wörtern
  - c 2 w an derselben Position

  Prüfe, ob sich dein mentales Modell von „Wort ab Cursor“ gegenüber „bis Wortende“ bestätigt.

4.2.6. Warum c makrosensibel ist

Der Operator c wechselt nach der Motion sofort in den Insert Mode. Das macht ihn im interaktiven Arbeiten extrem komfortabel, in Makros aber sensibel:

- Makros müssen *immer* mit Esc sauber aus dem Insert Mode zurückkehren.
- Jeder Tippfehler während der Makroaufzeichnung wird später exakt reproduziert – inklusive falscher Zeichen.
- Bewegungsbasierte Varianten wie c % oder c 3 w sind empfindlicher gegenüber Layoutänderungen als Textobjekte.
- Textobjekte wie c i w, c i (, c i p bleiben stabil, selbst wenn sich Einrückungen oder Zeilenumbrüche ändern.

Merksatz für Makros: Nutze nach Möglichkeit *Textobjekte* (c i w, c i (, c i p) statt freier Bewegungen mit Pfeiltasten oder reinen h/l-Sequenzen.

4.2.7. Beispiele aus der Praxis

- c i ” ändert den Inhalt eines Anführungsblocks (z. B. in Strings wie `"status=ok"`).
- c % ändert von einer Klammer zur passenden Gegenklammer; ideal für Funktionsaufrufe oder Argumentlisten.
- c 3 j ändert die aktuelle Zeile plus drei weitere Zeilen; geeignet für „Blockersatz“ in Logik- oder Kommentarblöcken.

In allen Fällen gilt: c übernimmt den Löschteil, wechselt in den Insert Mode, und du schreibst den neuen Inhalt direkt an die Stelle der alten Struktur.

4.2.8. Zusammenfassung

Der Operator c gehört zu den flexibelsten Werkzeugen in VIM:

- kombiniert Löschen und Schreiben in einem Schritt,
- arbeitet mit denselben Bewegungen wie d,
- wird durch Textobjekte (c i w, c i (, c i p) besonders präzise und robust,
- ist in Makros mächtig, erfordert aber sauberen Abschluss mit Esc

Wer c gezielt mit Textobjekten und klaren Bewegungen kombiniert, kann komplexe strukturelle Änderungen mit wenigen Tastenanschlägen ausdrücken – interaktiv wie auch in stabilen Makros.

4.3. Kopieren und Einfügen (y, p)

Der Operator y (*yank*) kopiert Text in ein Register, ohne ihn zu löschen. Das Einfügen erfolgt mit p oder P. y folgt dabei exakt dem Operator–Motion-Konzept wie d und c: erst der Operator, dann die Bewegung oder das Textobjekt.

Standardmäßig landen alle Yanks im „unnamed register“, das auch von Löschbefehlen (d, x) verwendet wird. Dadurch kannst du gelöschte oder kopierte Inhalte mit denselben Einfügebefehlen wieder einsetzen.

4.3.1. Grundprinzip

Die Grundform lautet immer: y gefolgt von einer Motion.

Typische Beispiele:

- y w kopiert vom Cursor bis zum Anfang des nächsten `word`
- y e kopiert bis zum Ende des aktuellen bzw. nächsten `word`
- y $ kopiert vom Cursor bis zum Zeilenende
- y 0 kopiert vom Cursor bis zum Zeilenanfang

Der kopierte Text landet im Standardregister und kann mit p (nach dem Cursor) oder P (vor dem Cursor) wieder eingefügt werden.

Übungen

- Lege eine Testzeile an, z. B.:

  ```
  alpha beta gamma
  ```

  Setze den Cursor auf das `b` von `beta` und teste nacheinander: y w, y e, y $. Füge jeweils mit p in einer neuen Zeile ein und überprüfe, welcher Bereich genau kopiert wurde.
- Wiederhole die Übung, indem du den Cursor bewusst *mitten* in ein Wort setzt (z. B. auf das `p` in `alpha`). Beobachte, wie sich y w und y e unterscheiden, je nach Cursorposition.

- Kombiniere Yanks mit dem Wiederholungsoperator [.]: Kopiere mit [y][w] ein Wort und füge es mehrfach mit [p] ein. Verwende anschließend [u], um die Wiederholungen rückgängig zu machen.

4.3.2. Zeilenweises Kopieren

Wird [y] verdoppelt, arbeitet der Operator zeilenweise:

- [y][y] kopiert die aktuelle Zeile
- [2][y][y] kopiert die aktuelle und die nächste Zeile
- [y][2][j] kopiert die aktuelle Zeile und zwei weitere darunter

Zeilenweises Yanking verhält sich analog zu [d][d], nur ohne Löschung: Der Inhalt wird vollständig kopiert, die Originalzeilen bleiben erhalten. Beim Einfügen mit [p] oder [P] werden diese Zeilen als ganze Blöcke ober- oder unterhalb der aktuellen Zeile eingefügt.

Übungen

- Erzeuge drei Zeilen:

  ```
  Zeile 1
  Zeile 2
  Zeile 3
  ```

  Stelle den Cursor auf `Zeile 2` und teste: [y][y], dann [p]. Kontrolliere, wo die kopierte Zeile eingefügt wird.
- Wiederhole mit [2][y][y] und [p]: Überprüfe, ob beide Zeilen in der richtigen Reihenfolge eingefügt werden.
- Teste [y][2][j] auf einem Block aus mehreren Zeilen und füge ihn an anderer Stelle mit [P] ein. Achte darauf, wie sich das Einfügeverhalten von [p] und [P] bei zeilenweisen Yanks unterscheidet.

4.3.3. Kopieren mit Textobjekten

Wie bei [d] und [c] bieten Textobjekte besonders präzise Kopiervorgänge. Anstatt mühsam Zeichen oder reine Wortbewegungen zu kombinieren, beschreibst du direkt die logische Einheit, die kopiert werden soll.

- [y][i][w] kopiert das aktuelle Wort (ohne umgebenden Leerraum)

- y a w kopiert das aktuelle Wort *inklusive* nachfolgendem Leerraum
- y i (kopiert den Inhalt eines runden Klammerpaars (...)
- y a { kopiert einen gesamten Block inklusive geschweifter Klammern
- y i p kopiert den aktuellen Absatz

Textobjekte sind für Makros besonders stabil, weil sie sich an der Struktur orientieren (Wörter, Blöcke, Absätze) und nicht an fixen Zeilenlängen oder Einrückungen.

Übungen

- Lege eine Zeile an wie:

  ```
  int counter = value;
  ```

  Stelle den Cursor auf `counter` und teste nacheinander y i w und y a w. Füge mit p in einer neuen Zeile ein und vergleiche den Unterschied (Leerraum ja/nein).
- Erzeuge eine Zeile mit Klammern, z. B.:

  ```
  log_debug("status=", code);
  ```

  Setze den Cursor irgendwo innerhalb der runden Klammern und teste: y i (und y a (. Achte darauf, ob die Klammern selbst mitkopiert werden.
- Schreibe zwei Absätze mit einer Leerzeile dazwischen. Stelle den Cursor in den ersten Absatz und verwende y i p . Füge den kopierten Absatz unterhalb mit p ein und überprüfe, dass genau der gesamte Absatz (ohne umgebende Leerzeilen) kopiert wurde.

4.3.4. Einfügen (p und P)

Einfügen erfolgt in VIM über zwei Grundbefehle:

- p fügt den zuletzt kopierten oder gelöschten Text *nach* dem Cursor ein
- P fügt den Text *vor* dem Cursor ein

Das Verhalten unterscheidet sich je nach Art des Inhalts:

- **zeichen- oder wortweise Yanks**: p und P fügen direkt an der Cursorposition ein (vor/nach dem aktuellen Zeichen).
- **zeilenweise Yanks (z. B. y y)**:
  - p fügt die Zeile(n) *unterhalb* der aktuellen Zeile ein

  - P fügt die Zeile(n) *oberhalb* der aktuellen Zeile ein

VIM richtet zeilenweise Einfügungen automatisch an vollständigen Zeilen aus, sodass die Struktur (z. B. Codeblöcke) erhalten bleibt.

Übungen

- Lege drei Zeilen an:

  ```
  alpha
  beta
  gamma
  ```

  Stelle den Cursor auf `beta`, führe y y aus und teste nacheinander p und P. Beobachte, wo die kopierte Zeile landet.
- Schreibe eine Zeile wie:

  ```
  foo bar baz
  ```

  Setze den Cursor auf `b` von `bar`, führe y i w aus und füge mit p und P ein. Achte darauf, ob der Text vor oder nach dem aktuellen Zeichen eingefügt wird.

4.3.5. Register

Standardmäßig landet kopierter oder gelöschter Text im *Default-Register* `""`. VIM verwendet jedoch eine ganze Reihe von Registern, die gezielt adressiert werden können.

Wichtige Register im Überblick:

- `""` aktuelles Standardregister (zuletzt *gelöscht oder kopiert*)
- `"0` reines Yank-Register (zuletzt *kopiert*, unabhängig von Löschungen)
- `"1`–`"9` Verlaufsregister für Löschoperationen (d d, d w, ...)
- `"a`–`"z` benutzerdefinierte Register (explizit befüllbar)
- `"+` System-Clipboard (unter macOS/Linux meist „CLIPBOARD“)
- `"*` Auswahl-Zwischenablage (unter X11: „PRIMARY“)

Ein Register wird durch ein vorangestelltes Anführungszeichen adressiert:

- " a y i w kopiert das aktuelle Wort in Register `a`
- " a p fügt den Inhalt von Register `a` ein
- " + y y kopiert die aktuelle Zeile in das System-Clipboard
- " + p fügt aus dem System-Clipboard in VIM ein

Wichtige Feinheit: [y][y] schreibt gleichzeitig in `""` *und* `"0`. Löschbefehle wie [d][d] überschreiben `""`, aber nicht `"0`. Damit bleibt das zuletzt explizit kopierte Fragment im Yank-Register erhalten.

Übungen

- Kopiere ein Wort mit [y][i][w]. Lösche anschließend eine Zeile mit [d][d]. Füge zweimal ein:
  - [p] (verwendet `""`)
  - [”][0][p] (verwendet explizit das Yank-Register)

  Vergleiche, ob sich der eingefügte Inhalt unterscheidet.
- Nutze ein benutzerdefiniertes Register:
  - [”][b][y][a][p] speichere einen ganzen Absatz in Register `b`
  - bewege dich an eine andere Stelle im Dokument und füge mit [”][b][p] ein

  Prüfe, dass der Inhalt von `b` unverändert bleibt, auch wenn du zwischendurch andere Yanks ausführst.
- Kopiere eine Zeile mit [”][+][y][y] und füge sie außerhalb von VIM in eine andere Anwendung ein (z. B. Terminal oder Browser). Kopiere anschließend einen Text aus der GUI in die Zwischenablage und füge ihn in VIM mit [”][+][p] ein.

4.3.6. Suchbasiertes Kopieren

Wie bei [d] kannst du auch mit [y] bis zu einem Suchtreffer kopieren:

- [y][/]`string`[Enter] kopiert vom aktuellen Cursor bis *kurz vor* die nächste Übereinstimmung von `string`.
- [y][?]`string`[Enter] kopiert rückwärts bis *kurz vor* die vorherige Übereinstimmung von `string`.

Der kopierte Bereich wird in ein Register (standardmäßig `""` und `"0`) geschrieben und kann anschließend mit [p] oder [P] eingefügt werden. Suchbasiertes Kopieren ist besonders nützlich in:

- langen Logfiles mit wiederkehrenden Markern (ERROR, WARN, ...)
- strukturierten Texten mit Schlüsselwörtern
- Konfigurationsdateien, in denen bis zu einem bestimmten Token kopiert werden soll

Übungen

- Lege eine Testzeile an, z. B.:

  `INFO start ... DEBUG step ... ERROR stop`

  Setze den Cursor vor `INFO` und teste y / DEBUG Enter. Füge mit p ein und kontrolliere, dass exakt der Bereich bis direkt vor DEBUG kopiert wurde.
- Wiederhole die Übung rückwärts: Setze den Cursor auf ERROR und verwende y ? DEBUG Enter. Füge mit P ein und prüfe, welcher Bereich rückwärts kopiert wurde.

4.3.7. Makrohinweise

Für Makros sind beim Kopieren mit y insbesondere folgende Punkte entscheidend:

- **Textobjekte verwenden**: y i w, y a { oder y i p orientieren sich an Struktur, nicht am Sichtlayout, und sind daher stabiler als Pfeiltasten.
- **Keine Pfeiltasten**: Pfeiltasten erzeugen terminalabhängige Sequenzen und machen Makros fragil. Nutze stattdessen w, $, % und Textobjekte.
- **Register explizit setzen**, wenn mehrere Yanks gemischt werden: " a y i w, " b y i p usw. sorgen dafür, dass ein Makro nicht versehentlich das falsche Register überschreibt.
- **Zeilenweises Yanking** (y y, 2 y y, y 2 j) ist besonders stabil, wenn Makros auf Zeilenebene arbeiten sollen.

Übungen

- Zeichne ein Makro auf, das mit y i w ein Wort kopiert und an einer anderen Stelle mit p einfügt. Spiele das Makro in mehreren Zeilen ab und prüfe, dass es unabhängig von Zeilenlängen stabil bleibt.
- Erzeuge ein Makro, das eine Zeile mit y y kopiert und in ein benanntes Register schreibt, z. B. " a y y. Nutze später " a p, um diesen Inhalt gezielt einzufügen, während andere Yanks weiterlaufen.

4.3.8. Zusammenfassung

Yanking kombiniert das Operator–Motion–Konzept mit der Flexibilität der VIM-Register. Dadurch wird präzises Kopieren und Einfügen möglich – sowohl im manuellen Alltag als auch in komplexen Makros. Besonders stark sind:

- **Textobjekte** (y i w, y a {, y i p) für strukturorientiertes Kopieren,
- **suchbasierte Yanks** (y / `string` Enter, y ? `string` Enter) für Logfiles und Marker,
- **explizite Registersteuerung** (" a, " +, " 0) für reproduzierbare Makros und saubere Workflows.

Wer diese Techniken kombiniert, kann Text in VIM äußerst effizient und stabil manipulieren, ohne auf Maus oder visuelle Selektion angewiesen zu sein.

4.4. Ersetzen (r, R)

VIM bietet zwei grundlegende Mechanismen zum Ersetzen von Zeichen: den einfachen Einzelersetzungsbefehl r und den Überschreibmodus R. Beide unterscheiden sich klar vom Änderungsoperator c, weil sie keinen Operator–Motion–Aufbau haben, sondern direkt am Cursor arbeiten.

4.4.1. Einzelzeichen ersetzen (r)

Der Befehl r ersetzt genau *ein* Zeichen an der aktuellen Cursorposition. Nach r wartet VIM auf das neue Zeichen und trägt es an derselben Stelle ein.

Beispiele:

- r x ersetzt das Zeichen unter dem Cursor durch `x`.
- r (ersetzt das Zeichen unter dem Cursor durch `(`.
- r 0 ersetzt das Zeichen unter dem Cursor durch `0`.

Wichtige Eigenschaften von r:

- VIM bleibt im Normal Mode (kein Wechsel in den Insert Mode).
- Es gibt keine Motion und keine Textobjekte – r wirkt immer nur auf das Zeichen direkt unter dem Cursor.
- Ideal für punktuelle Korrekturen (falsches Zeichen, Vertipper in Zahlen, einzelne Klammerfehler).

- Ein vorangestellter Count (3 r) ist *nicht* als Mehrfachersetzung gedacht, sondern ersetzt nacheinander die folgenden Zeichen mit demselben neuen Zeichen – praktisch selten sinnvoll, aber vorhanden.

Übungen

- Lege eine Testzeile an, z. B.:

  ```
  prinf("Hallo Wunstorf");
  ```

  Korrigiere den Tippfehler `prinf` zu `printf`, indem du den Cursor auf das falsche Zeichen setzt und mit r das korrekte einsetzt.
- Erzeuge eine Zeile mit gemischten Klammern, z. B.:

  ```
  if [x == 1) { do_something(); ]
  ```

  Korrigiere die falschen Klammern ausschließlich mit r (kein Insert Mode, keine Visual-Selektion).
- Probiere den Count-Effekt: Lege z. B. eine Zeile AAAAA an, setze den Cursor auf den ersten Buchstaben und führe 3 r x aus. Beobachte, wie die ersten drei Zeichen durch `x` ersetzt werden, ohne den Normal Mode zu verlassen.

4.4.2. Mehrfaches Ersetzen (r mit Count)

Auch r akzeptiert einen vorangestellten Count: VIM ersetzt dann *mehrere* Zeichen hintereinander durch dasselbe neue Zeichen.

- 3 r x ersetzt die nächsten drei Zeichen (ab Cursor) durch `x`.

Dabei gilt:

- Der Cursor bleibt im Normal Mode.
- VIM nimmt den Count (3 r), wartet dann auf *ein* neues Zeichen (z. B. `x`) und überschreibt damit die nächsten drei Zeichen.
- Zeilenumbrüche werden nicht übertreten – der Effekt endet spätestens am Zeilenende.

Mehrfaches Ersetzen mit r ist ein sehr grobes Werkzeug: Es eignet sich für schnelle Korrekturen in künstlichen Testzeilen (Zeichenfolgen, Dummy-Daten), kann aber leicht Struktur oder Syntax zerstören. In realem Code sind Textobjekte und c-Operationen meist die bessere Wahl.

Übungen

- Lege eine Zeile an:

```
##########
```

Setze den Cursor auf das erste # und führe [4][r][x] aus. Prüfe, dass genau vier Zeichen durch x ersetzt wurden und der Rest unverändert blieb.

- Erzeuge eine Zeile:

  ```
  00000000
  ```

  Ersetze die mittleren vier Ziffern durch 7, indem du den Cursor auf die dritte 0 setzt und [4][r][7] verwendest. Kontrolliere, ob du die gewünschte Stelle getroffen hast – kleine Off-by-one-Fehler werden hier schnell sichtbar.
- Teste bewusst den Unterschied zu [R]: Nimm eine Zeile mit Text, wende einmal [3][r][x] an und einmal [R][x][x][x][Esc] an derselben Stelle. Beobachte, wie sich beide Varianten in Bezug auf Cursorposition und Gefühl beim Editieren unterscheiden.

4.4.3. Überschreibmodus (Replace Mode, R)

Der Befehl [R] aktiviert den Replace Mode. Im Gegensatz zum Insert Mode ersetzt jede Eingabe ein bestehendes Zeichen, statt Text einzuschieben.

Eigenschaften:

- jeder getippte Buchstabe überschreibt das Zeichen unter dem Cursor,
- der Cursor bewegt sich dabei nach rechts,
- [<BS>] geht ein Zeichen zurück und stellt das ursprüngliche Zeichen wieder her,
- [Esc] beendet den Replace Mode.

Beispiel:

```
Ausgang:   fooBar
Cursor:    fooBar
               ^
Aktion:    R -> hallo
Ergebnis:  foohallo
```

Übungen

- Lege eine Zeile an:

  ABCDEFGH

Setze den Cursor auf das C und tippe R gefolgt von `xyz<Esc>`. Kontrolliere, welche Zeichen überschrieben wurden und wo der Cursor am Ende steht.

- Erzeuge eine Zeile:

  ```
  status=OK; mode=auto
  ```

  Setze den Cursor auf das O von OK und ersetze mit R den Text durch `FAIL<Esc>`. Vergleiche das Ergebnis mit einer Änderung per c i w und beachte den Unterschied im Editiergefühl.
- Spiele mit R und <BS>: Schreibe zunächst bewusst ein falsches Wort und korrigiere es dann, indem du im Replace Mode mit <BS> zurückgehst. Beobachte, dass die ursprünglichen Zeichen wieder erscheinen, statt gelöscht zu werden.

4.4.4. Unterschied zwischen c und R

c und R wirken auf den ersten Blick ähnlich, verhalten sich intern aber deutlich unterschiedlich:

- c *löscht* zuerst den betroffenen Bereich (per Motion oder Textobjekt) und wechselt dann in den Insert Mode.
- R bleibt im Replace Mode und *überschreibt* Zeichen direkt, ohne vorab zu löschen.
- c arbeitet *strukturell*: mit Bewegungen wie w, e, $ oder Textobjekten wie i w, i (, i p.
- R arbeitet *flach zeichenweise*: jeder Tastendruck ersetzt genau ein bestehendes Zeichen.

Praktisch bedeutet das:

- c i w ist ideal, um ein ganzes Wort auszutauschen, egal wie lang es ist.
- R ist gut geeignet, um feste Spaltenbreiten (Tabellen, ASCII-Art) zu bearbeiten, bei denen die Position wichtiger ist als die Textstruktur.

Übungen

- Lege eine Zeile an:

  ```
  status=OK; mode=auto
  ```

Setze den Cursor auf O in OK und ersetze einmal mit [c][i][w] durch FAIL<Esc>, danach in einer Kopie der Zeile mit [R]FAIL[Esc]. Vergleiche: Wie unterscheiden sich Ergebnis und Cursorposition?

- Erzeuge eine Zeile:

  ```
  count=0000; limit=9999
  ```

  Setze den Cursor auf die erste 0 und probiere: [c][3][w] und [R]1234[Esc] in jeweils separaten Kopien der Zeile. Achte darauf, wie [c] ganze „Einheiten“ ersetzt, während [R] strikt zeichenweise arbeitet.
- Notiere dir Situationen aus deinem Alltag (Logzeilen, Konfigs, Code), in denen du eher strukturell (c…) oder eher positionsgenau ([R]) arbeiten möchtest. Übertrage ein bis zwei dieser Fälle testweise in eine Datei und spiele beide Varianten durch.

4.4.5. Visueller Replace Mode (gR)

Der Befehl [g][R] aktiviert einen erweiterten Replace Mode, der sich besonders für *visuelle* Strukturen eignet:

- ideal für Tabellen, ASCII-Art, Monospace-Diagramme, Spaltenlayouts,
- jedes Zeichen wird an seiner aktuellen Position überschrieben,
- Cursorbewegungen mit [h], [j], [k], [l] ändern, *wo* weiter überschrieben wird,
- [Esc] beendet den visuellen Replace Mode.

Im Unterschied zu [R] ist [g][R] darauf ausgelegt, ganze „Blöcke“ mit fester Geometrie zu bearbeiten. Das Verhalten ist vorhersehbar, solange Zeilenlängen und Spaltenbreiten stabil bleiben.

Übungen

- Lege einen kleinen Tabellenblock an:

  ```
  +--------+
  | foo    |
  | bar    |
  +--------+
  ```

Setze den Cursor auf das `f` in `foo`, aktiviere [g][R] und schreibe HELLO, bewege dich mit [j] in die nächste Zeile und überschreibe `bar` ebenfalls. Behalte im Auge, dass der Rahmen unverändert bleibt.

- Erzeuge eine einfache ASCII-Zeitleiste, z. B.:

```
[----|----|----|----]
```

  Nutze [g][R], um Markierungen oder Labels exakt an bestimmten Spalten zu platzieren, ohne die Länge der Linie zu verändern.
- Vergleiche [R] und [g][R] in einem mehrzeiligen Block: Überschreibe denselben Bereich einmal mit [R] (Zeile für Zeile) und einmal mit [g][R], während du dich mit [j]/[k] durch den Block bewegst. Notiere dir, welche Variante für deine typischen Aufgaben intuitiver wirkt.

4.4.6. Makrohinweise

Für Makros sollte [r] sehr gezielt und sparsam eingesetzt werden:

- [r] ist *punktuell* und zuverlässig: genau ein Zeichen wird ersetzt.
- [R] ist *riskant*: schon kleine Textlängenänderungen können ein Makro brechen.
- Für strukturelle Änderungen sind [c] + Motion oder Textobjekte ([c][i][w], [c][i][p], [c][i][(]) fast immer die bessere Wahl.

Bewährtes Muster für stabile Makros:

- [c][i][w], [c][w], [c][$] statt langer [R]-Sequenzen,
- [r] nur für einzelne, klar definierte Zeichenkorrekturen (z. B. „typo“ beheben, ein einzelnes Trennzeichen ändern).

Übungen

- Nimm eine Testzeile mit einem offensichtlichen Tippfehler, z. B.:

```
statuz=OK; mode=auto
```

  Nimm ein Makro auf, das den Buchstaben `z` in `s` ändert (mit [r]), und wiederhole es in mehreren Zeilen. Prüfe, wie stabil das Makro bleibt, wenn du dieselbe Struktur in leicht veränderten Zeilen anwendest.
- Erzeuge mehrere ähnliche Zeilen mit unterschiedlichen Werten:

```
status=OK
status=WARN
status=ERROR
```

Nimm ein Makro auf, das mit c i w den Wert hinter dem Gleichheitszeichen durch PENDING ersetzt. Vergleiche die Stabilität dieses Makros mit einer Variante, die versucht, dasselbe mit R zu erreichen.

4.4.7. Zusammenfassung

- r ersetzt genau *ein* Zeichen an der Cursorposition (optional mit Count für mehrere aufeinanderfolgende Zeichen).
- R überschreibt fortlaufend wie ein klassischer Editor und ist vor allem für feste Spaltenlayouts geeignet.
- r ist gut makrotauglich, solange es um punktuelle Korrekturen geht.
- R ist in Makros nur sehr eingeschränkt sinnvoll, weil jede Abweichung in der Textlänge das Makro aus dem Tritt bringen kann.
- Für strukturelle Änderungen sind Kombinationen aus c + Motion bzw. Textobjekten (c i w, c i p, c i (, c i ” usw.) die robustere und flexiblere Lösung.

4.5. Wiederholen (.)

Der Punkt . ist einer der mächtigsten Befehle in VIM. Er wiederholt die *zuletzt ausgeführte Änderung*. Dabei spielt es keine Rolle, ob die Änderung mit d, c, i, r, x oder einem Textobjekt ausgeführt wurde – entscheidend ist, dass tatsächlich Text verändert wurde.

4.5.1. Grundprinzip

Der Befehl . wiederholt exakt das letzte Änderungskommando, inklusive aller Bewegungen und Counts.

Typische Beispiele:

- d w gefolgt von . löscht erneut ein Wort
- c a w gefolgt von . ändert das nächste Wort (inklusive Leerraum)

- x gefolgt von . löscht erneut ein Zeichen
- 3 d w gefolgt von . löscht wieder drei Wörter

Wichtig: . wiederholt immer die letzte *Änderung*, nicht die letzte Taste. Ein reines w oder j (nur Bewegung, keine Textänderung) setzt die Wiederholungsbasis nicht neu.

4.5.2. Was . wiederholt — und was nicht

Wiederholt werden:

- die *Operation* (z. B. d, c, x, r)
- die komplette *Motion* (w, e, $, j, Textobjekte wie i w, i (usw.)
- der *Count* (z. B. 3 d w wird als „lösche drei Wörter“ gemerkt)
- der eingegebene Text bei Änderungen im Insert/Replace Mode (z. B. c i w + `foo` Esc lässt sich mit . erneut ausführen)

Nicht (bzw. nur indirekt) wiederholt werden:

- reine Bewegungen ohne Textänderung (w, j, G, % allein)
- reine Such- und Navigationsbefehle wie / `pattern` Enter oder n, solange ihnen keine Änderung folgt
- exklusive Kommandozeilenbefehle wie `:w`, `:q`, `:set`
- Scroll- und Ansichtsbefehle (Ctrl - f, Ctrl - b, z z, z t usw.), also „nur Bildschirmbewegung“
- visuelle Markierungen (v, V, Ctrl - v) ohne anschließende Änderung

Ein Count vor . skaliert die letzte Änderung:

- d w gefolgt von 5 . führt fünfmal „ein Wort löschen“ aus
- c i w + Text gefolgt von 3 . wendet dieselbe Änderung auf drei Wörter an

Übungen

- Erzeuge mehrere ähnliche Zeilen, z. B.:

```
int value = 1;
int value = 2;
int value = 3;
int value = 4;
```

Setze den Cursor auf die erste Zeile, führe c i w auf `1 ;` aus und ersetze durch `count ;` Esc. Wandere mit j durch die weiteren Zeilen und benutze jeweils ., um die Änderung zu wiederholen. Beobachte, wie exakt die Kombination „c i w + Text“ reproduziert wird.

- Nimm eine Zeile mit mehreren Wörtern, z. B.:

  ```
  foo bar baz qux
  ```

  Lösche das erste Wort mit d w. Nutze dann ., um nacheinander die folgenden Wörter zu löschen. Spiele danach mit u (Undo) und wieder ., um zu sehen, welche Änderung tatsächlich wiederholt wird.
- Kombiniere Count und Punkt: Lösche mit 2 d w zwei Wörter auf einmal und wiederhole dann mit 3 .. Kontrolliere, ob insgesamt sechsmal „Wort löschen“ ausgeführt wurde (dreimal je zwei Wörter).

4.5.3. Wiederholen von Texteinfügungen

Auch Einfügungen im Insert Mode werden als eine einzige Änderung gespeichert. VIM merkt sich dabei:

- den auslösenden Befehl (z. B. c w, c i w, o, O)
- den kompletten eingegebenen Text
- das abschließende Esc (Ende der Änderung)

Dadurch kann . komplexe Texteinfügungen exakt reproduzieren.

Beispielablauf:

1. c w (wechsel in den Insert Mode und lösche das aktuelle Wort)
2. Text eingeben: `Hallo`
3. Esc (Änderung abschließen)
4. . wiederholt exakt: „c w + `Hallo` + Esc“

Das macht . ideal für Serienänderungen, bei denen derselbe Text an mehreren Stellen eingefügt oder ersetzt werden soll.

Übungen

- Erzeuge eine Zeile:

  ```
  foo foo foo foo
  ```

Setze den Cursor auf das erste `foo` und führe [c][i][w] + `bar`[Esc] aus. Wandere mit [w] über die restlichen `foo` und wiederhole die Änderung jedes Mal mit [.]. Kontrolliere, dass überall exakt `bar` eingesetzt wird.

- Schreibe mehrere nahezu identische Logzeilen, z. B.:

```
[INFO] status=ok
[INFO] status=ok
[INFO] status=ok
```

  Ändere in der ersten Zeile mit [c][i][w] den Token `INFO` zu `WARN`[Esc]. Nutze dann [j] und [.], um dieselbe Änderung in den weiteren Zeilen zu wiederholen.
- Teste den Unterschied zwischen einer Änderung mit Motion und ohne: Führe einmal [A] `// TODO`[Esc] am Zeilenende aus und wiederhole mit [.] in weiteren Zeilen. Beobachte, dass VIM den gesamten „Anfügen-und-Schreiben“ Vorgang reproduziert — inkl. Sprung ans Zeilenende.

4.5.4. Sonderfall: x

Der Befehl [x] zählt als Änderung und ist daher direkt mit [.] kombinierbar:

- [x]
- [.]

löscht zwei Zeichen hintereinander. Das ist oft schneller und flüssiger als [2][x] zu tippen und wird in der Praxis gern für kleine Korrekturen genutzt.

Übungen

- Lege eine Zeile an:

```
abcdef
```

  Setze den Cursor auf `c`, führe [x] und dann [.] aus. Kontrolliere, dass exakt zwei Zeichen entfernt wurden (`c` und `d`).
- Wiederhole dieselbe Übung mit [2][x]. Achte darauf, wie sich das Tippgefühl unterscheidet und welche Variante dir in der Praxis besser von der Hand geht.

4.5.5. Wiederholen und Textobjekte

Textobjekt-basierte Änderungen eignen sich ideal für [.], weil sie strukturorientiert arbeiten:

- [c][i][w] + [.] ändert mehrere Wörter nacheinander
- [d][i][(] + [.] löscht mehrere Klammerinhalte
- [c][i][p] + [.] löscht oder ändert mehrere Absätze

Wichtig: [.] wiederholt das Textobjekt selbst, nicht die Cursorbewegung, die dich zum nächsten Bereich bringt. Du musst den Cursor also jeweils vor der Wiederholung sinnvoll positionieren (z. B. mit [w], [j], Absatz- oder Klammerbewegungen).

Übungen

- Lege mehrere ähnlich aufgebaute Funktionsaufrufe an, z. B.:

  ```
  log_debug("status=ok");
  log_debug("status=ok");
  log_debug("status=ok");
  ```

  Setze den Cursor in die erste Klammer, führe [c][i][(]`"status=failed"`[Esc] aus. Wandere mit [j] durch die weiteren Zeilen und verwende jeweils [.], um den Klammerinhalt zu ersetzen.
- Schreibe mehrere Absätze mit identischer Struktur. Ändere einen Absatz mit [c][i][p] + neuem Text und schließe mit [Esc] ab. Positioniere den Cursor jeweils an den Anfang der nächsten Absätze und nutze [.], um die Änderung zu übertragen. Beobachte, dass Layoutänderungen innerhalb des Absatzes keine Rolle spielen – das Textobjekt steuert die Wiederholung.

4.5.6. Counts und Wiederholen

Counts wirken auch in Kombination mit [.]:

- [3][d][w] löscht drei Wortfragmente
- [.] löscht erneut drei Wortfragmente

Ein Count vor [.] skaliert die zuletzt ausgeführte Änderung:

- [d][w] gefolgt von [5][.] führt fünfmal „ein Wort löschen“ aus

- 2 x gefolgt von 3 . löscht insgesamt acht Zeichen (zweimal durch 2 x, sechsmal durch 3 .)

Übungen

- Lege eine Zeile an:

  ```
  eins zwei drei vier fünf sechs sieben
  acht neun zehn
  ```

  Führe 2 d w aus und anschließend 3 .. Prüfe, dass insgesamt viermal „zwei Wörter löschen“ ausgeführt wurde (acht Wörter verschwinden).
- Wiederhole die Übung mit d w und 5 .. Achte darauf, wie viel weniger du tippen musst und wie exakt sich die Änderung wiederholen lässt.

4.5.7. Warum . makrotauglich ist

Der Punkt fasst die gesamte Änderung als eine kompakte Einheit zusammen. Dadurch entstehen sehr robuste Makros:

- **weniger Komplexität** – die eigentliche Änderung liegt in der ursprünglichen Aktion, das Makro triggert nur noch .
- **weniger Cursorbewegungen** – viele Schritte lassen sich durch eine Kombination aus Bewegung + . ausdrücken
- **strukturelle Wiederholung** – in Verbindung mit Textobjekten (c i w, c i p, d i (...) bleibt das Makro stabil, auch wenn sich Zeilenlängen oder Umbrüche ändern

Makros, die gezielt . einsetzen, sind oft stabiler als Makros, die jede Einzelaktion (mehrfaches d, c, y usw.) explizit aufzeichnen. Statt eine lange Sequenz abzufahren, wird nur die bereits getestete Änderung wiederverwendet.

Wichtiger Hinweis: Punkt wiederholt kein Makro

Der Befehl . wiederholt *nie* ein aufgezeichnetes Makro. Er wiederholt ausschließlich die letzte Änderung – selbst wenn diese innerhalb eines Makros entstanden ist.

Makros selbst werden ausgeführt mit:

- @ q Makro im Register q ausführen

- @ @ zuletzt ausgeführtes Makro erneut ausführen

Eine typische Falle: Wer erwartet, dass . das gesamte Makro reproduziert, wiederholt nur das *letzte* Änderungskommando, das während der Makroausführung stattgefunden hat.

Übung

- Zeichne ein Makro in q auf, das in mehreren Schritten einen Textblock verändert (z. B. Bewegung + c i w + Text). Führe danach einmal @ q, dann . und dann @ @ aus. Beobachte genau den Unterschied zwischen:
  - kompletter Makroausführung (@ q, @ @)
  - reiner Wiederholung der letzten Änderung (.)

4.5.8. Zusammenfassung

- . wiederholt die letzte Änderung, nicht die letzte Taste.
- Counts wirken sowohl bei der ursprünglichen Änderung als auch vor ..
- In Kombination mit Textobjekten und sauberen Bewegungen wird . zum zentralen Werkzeug für Serienänderungen.
- In Makros reduziert . die Komplexität und erhöht die Stabilität, weil eine einmal getestete Änderung immer wiederverwendet wird.

5. Textobjekte

5.1. Innere und äußere Objekte (i, a)

Textobjekte gehören zu den mächtigsten Konzepten in VIM. Sie definieren semantische Bereiche wie Wörter, Sätze, Absätze oder Klammerstrukturen und ermöglichen präzise Operationen *unabhängig* von der exakten Cursorposition oder Zeilenstruktur.

Zwei Präfixe steuern, *welcher Teil* eines Objekts erfasst wird:

- **i** (*inner*) – nur der inhaltliche Kern
- **a** (*around* / *outer*) – Kern *inklusive* Rand (Leerraum/Klammern)

Textobjekte werden immer mit einem Operator kombiniert, etwa **d**, **c** oder **y**. Dadurch entstehen Operationen wie **d** **i** **w**, **c** **i** **p** oder **y** **a** **(**, die strukturell arbeiten und sich perfekt für Makros eignen.

5.1.1. Grundprinzip

Ein Textobjekt wird immer nach dem Schema „Operator + **i**/**a** + Objekt“ aufgerufen.

Typische Formen:

- **d** **i** **w** löscht das *innere Wort* an der Cursorposition
- **d** **a** **w** löscht das Wort *inklusive* nachfolgendem Leerraum
- **c** **i** **(** ändert den Inhalt runder Klammern
- **d** **a** **{** löscht einen gesamten Block inklusive geschweifter Klammern
- **v** **i** **p** wählt den *inneren Absatz* visuell aus

Wichtige Eigenschaften:

- Textobjekte funktionieren von *fast jeder* Cursorposition innerhalb des Objekts.
- Sie ignorieren Zeilenumbrüche und Einrückungen – entscheidend ist die Struktur.
- Sie sind wesentlich robuster als reine Zeilen- oder Zeichenbewegungen.

Übungen

- Lege eine kleine C-ähnliche Zeile an:

  ```
  if (status == OK) { do_something(); }
  ```

  Setze den Cursor irgendwo zwischen (und) und teste: d i (, c i (, y i (. Beobachte, dass es >keine Rolle</emph> spielt, ob du am Anfang, in der Mitte oder am Ende des Ausdrucks stehst.
- Wiederhole die Übung mit dem Block:

  ```
  { do_something(); log_debug("status=OK");
  }
  ```

  und probiere d i { und d a { aus. Vergleiche, welcher Teil jeweils verschwindet (nur Inhalt vs. Inhalt + Klammern).
- Wähle einen Absatz mit mehreren Sätzen und probiere nacheinander: v i p, d i p, c i p. Achte darauf, wie konsistent der gesamte Absatz als Einheit behandelt wird.

5.1.2. Innere Objekte (i)

i bezeichnet immer den *inneren* Inhalt eines Objekts – ohne umgebende Leerzeichen, Klammern oder Anführungszeichen. Das ist ideal, wenn du den Kern bearbeiten willst, ohne die Struktur zu zerstören.

Typische innere Textobjekte:

- i w inneres Wort
- i (Inhalt runder Klammern (...)
- i { Inhalt geschweifter Klammern { ...}
- i [Inhalt eckiger Klammern [...]
- i ” Inhalt zwischen Anführungszeichen " ..."
- i p innerer Absatz (ohne nachfolgende Leerzeile)

Kombiniert mit Operatoren entstehen präzise Befehle:

- d i w löscht das *aktuelle Wort*, ohne den umgebenden Leerraum anzutasten
- c i ” ändert nur den Inhalt eines String-Literals
- y i (kopiert nur den Ausdruck in Klammern, nicht die Klammern selbst

- d i p löscht den Inhalt eines Absatzes, lässt die Struktur außen herum stehen

Übungen

- Lege eine Zeile an:

  ```
  user.name = "jens";
  ```

  Setze den Cursor irgendwo innerhalb von `"jens"` und teste: c i ” gefolgt von einem neuen Namen und Esc Wiederhole die Übung mit d i ” und y i ”.
- Erzeuge eine Datenstruktur:

  ```
  { "user": "jens", "role": "admin" }
  ```

  Probiere nacheinander:
  - d i { – Inhalt des Objekts löschen
  - c i { – Inhalt ersetzen
  - y i { – Inhalt in ein Register kopieren

  Achte darauf, dass die äußeren { } jeweils erhalten bleiben.
- Wähle einen Fließtext mit mehreren Absätzen und teste: c i p und d i p an verschiedenen Stellen innerhalb eines Absatzes. Kontrolliere, dass immer der komplette Absatz erwischt wird, egal, wo der Cursor innerhalb des Absatzes steht.

5.1.3. Äußere Objekte (a)

a umfasst den gesamten Bereich *inklusive* der umschließenden Struktur – also z. B. Leerraum oder Klammern. Im Gegensatz zu inneren Objekten (i) wird der Rahmen bewusst mit einbezogen.

Typische Beispiele:

- a w Wort inklusive vor-/nachfolgendem Leerraum
- a (runde Klammern *mit* Inhalt
- a { Block inklusive geschweifter Klammern
- a ” Anführungszeichen *inklusive* Inhalt
- a p Absatz inklusive trennender Leerzeile

Äußere Objekte sind ideal, wenn ein kompletter Block *mit* seinem Umfeld entfernt, verschoben oder kopiert werden soll – etwa beim Umstrukturieren von Code oder beim Anpassen ganzer Textbausteine.

Übungen

- Lege eine Zeile an:

  ```
  foo bar baz
  ```

  Setze den Cursor irgendwo in `bar` und vergleiche **d** **i** **w** mit **d** **a** **w**. Beobachte den Unterschied: **d** **i** **w** lässt den Leerraum stehen, **d** **a** **w** nimmt den umgebenden Leerraum mit.
- Erzeuge eine Zeile:

  ```
  log_debug("status=ok");
  ```

  Setze den Cursor in den Klammerinhalt und teste nacheinander: **d** **i** **(**, **d** **a** **(**. Kontrolliere, ob bei **d** **i** **(** nur der Inhalt verschwindet und bei **d** **a** **(** die Klammern gleich mit gelöscht werden.
- Schreibe zwei Absätze, getrennt durch eine Leerzeile. Setze den Cursor irgendwo im ersten Absatz und probiere: **d** **i** **p** und **d** **a** **p**. Vergleiche, wie sich der Unterschied „innerer Absatz“ vs. „Absatz inklusive Leerzeile“ auf das Layout auswirkt.

5.1.4. Visuelle Auswahl mit Textobjekten

Textobjekte lassen sich auch direkt im Visual Mode nutzen. Statt reiner Cursorbewegungen markierst du damit sofort sinnvolle Einheiten, die du anschließend mit einem Operator bearbeiten kannst.

Typische Kombinationen:

- **v** **i** **w** wählt das aktuelle Wort (innen)
- **v** **a** **w** wählt das Wort inklusive umgebendem Leerraum
- **v** **i** **(** wählt den Inhalt runder Klammern
- **v** **a** **{** wählt den gesamten Block mit geschweiften Klammern
- **v** **i** **p** wählt den inneren Absatz

Im Anschluss kannst du z. B. mit **d**, **c**, **y** oder **=** auf die markierte Struktur operieren, ohne dich mühsam zeichenweise heranzutasten.

Übungen

- Lege eine Zeile an:

  ```
  foo bar baz
  ```

  Setze den Cursor irgendwo in `bar` und vergleiche [v][i][w] mit [v][a][w]. Beobachte, dass [v][i][w] nur das Wort selbst markiert, während [v][a][w] auch angrenzenden Leerraum einbezieht.
- Erzeuge eine Zeile:

  ```
  log_debug("status=ok");
  ```

  Setze den Cursor in den Klammerinhalt und teste nacheinander: [v][i][(] und [v][a][(]. Ergänze dann jeweils [d] oder [c], um den Unterschied zwischen „nur Inhalt“ und „Block inklusive Klammern“ praktisch zu sehen.
- Schreibe zwei Absätze, getrennt durch eine Leerzeile. Setze den Cursor in den ersten Absatz und benutze [v][i][p], anschließend [d][a][p]. Prüfe, wie sich Visual-Selection mit [v][i][p] und eine anschließende Operation ([d]) auf das Layout auswirken.

5.1.5. Makrohinweise

Textobjekte sind das Rückgrat stabiler Makros. Sie sind:

- unabhängig von Zeilenumbrüchen,
- weitgehend unabhängig von der exakten Cursorposition,
- robust gegenüber Einrückungs- und Layoutänderungen.

Statt cursorabhängiger Befehle wie [d][w] ist in Makros fast immer eine textobjektbasierte Variante vorzuziehen, z. B.:

- [d][w] löscht vom Cursor bis Wortgrenze (fehleranfällig),
- [d][i][w] löscht das gesamte Wort als Einheit (strukturell stabil).

Je mehr Makros auf [i][w], [a][w], [i][(], [i][p] usw. setzen, desto weniger brechen sie, wenn sich der Text später verändert.

5.2. Wort-, Satz- und Absatzobjekte

Neben Klammer- und Zeichenobjekten gehören Wort-, Satz- und Absatzobjekte zu den am häufigsten verwendeten Textobjekten in VIM. Sie bilden natürliche

Spracheinheiten ab und eignen sich besonders gut für Operatoren wie **d**, **c**, **y** und **v**.

Der große Vorteil: Statt mühsam Zeichen oder rohe Bewegungen zu kombinieren, arbeitest du direkt auf logischen Einheiten wie „Wort“, „Satz“ oder „Absatz“. Das macht Änderungen stabiler, präziser und deutlich makrotauglicher.

5.2.1. Wortobjekte

VIM unterscheidet zwei Arten von Wörtern:

- **klassisches Wort** (**w**)
  Folge aus Buchstaben, Ziffern und Unterstrich (`a–z`, `A–Z`, `0–9`, `_`); Interpunktion trennt.
- **WORD** (**W**)
  Folge beliebiger Nicht-Leerzeichen; getrennt wird nur durch Whitespace.

Darauf aufbauend gibt es die entsprechenden Textobjekte:

- **i** **w** inneres „word“ (ohne angrenzenden Leerraum)
- **a** **w** „word“ inklusive folgendem Leerraum
- **i** **W** inneres „WORD“
- **a** **W** „WORD“ inklusive Leerraum

Typische Operator-Kombinationen:

- **c** **i** **w** ändert den Wortinhalt an der Cursorposition
- **d** **a** **w** löscht Wort *plus* nachfolgenden Leerraum
- **y** **i** **W** kopiert das gesamte „WORD“

Übungen

- Lege eine Testzeile an:

  ```
  foo_bar, foo.bar; foo-bar
  ```

  Setze den Cursor in `foo_bar` und vergleiche systematisch: **c** **i** **w**, **c** **a** **w**, **d** **i** **w**, **d** **a** **w**. Notiere dir, wann Leerzeichen mit entfernt bzw. ersetzt werden und wie Interpunktion (`,`, `;`, `-`) das „word“ beeinflusst.
- Erzeuge eine Zeile mit Pfaden und Optionen:

  ```
  /usr/local/bin/script.sh
  --option=value,foo=bar;rc=0
  ```

Positioniere den Cursor in der Mitte des Pfads und vergleiche [c][i][w] mit [c][i][W]. Prüfe, wie fein [i][w] und wie grob [i][W] den Bereich wählen. Wiederhole die Übung mit [y][i][w] und [y][i][W], um Yanking-Verhalten zu verstehen.

- Schreibe eine kurze Konfigurationszeile:

  ```
  timeout = 30
  ```

  Setze den Cursor auf `30` und wende nacheinander an: [c][i][w] `60`[Esc], dann auf der nächsten Zeile [.]. Beobachte, wie sich Wortobjekte und der Punkt-Befehl ([.]) perfekt ergänzen, um Serienänderungen vorzunehmen.

5.2.2. Satzobjekte

Ein „Satz“ endet in VIM typischerweise bei:

- `.` (Punkt)
- `!` (Ausrufezeichen)
- `?` (Fragezeichen)

Zusätzlich berücksichtigt VIM beim Erkennen von Satzgrenzen:

- nachfolgende Leerzeichen,
- nachfolgende Anführungs- oder Klammerzeichen,
- bestimmte Einstellungen in `:set cpoptions` (insbesondere das Flag `s`).

Darauf aufbauend gibt es zwei Satzobjekte:

- [i][s] innerer Satz (ohne umgebende Leerzeichen/Abschlusszeichen)
- [a][s] Satz inklusive Abschlusszeichen und angrenzendem Leerraum

Typische Operator-Kombinationen:

- [c][i][s] ändert den Satzinhalt
- [d][a][s] löscht den gesamten Satz (inklusive Begrenzung)
- [v][i][s] markiert den inneren Satz visuell

Übungen

- Lege einen kurzen Absatz an, z. B.:

  ```
  Dies ist ein Test. Hier folgt ein zweiter
  Satz! Und noch einer?
  ```

Setze den Cursor in die Mitte des zweiten Satzes und vergleiche **c** **i** **s** mit **c** **a** **s**. Beobachte genau, ob Leer- und Satzzeichen mit geändert werden oder nicht.

- Wiederhole die Übung mit **d** **i** **s** und **d** **a** **s**. Achte darauf, wie sich die folgenden Sätze verschieben, wenn du den Satz inklusive Leerraum entfernst (**d** **a** **s**) im Vergleich zum inneren Satz (**d** **i** **s**).
- Setze den Cursor in verschiedene Sätze und benutze **v** **i** **s** und **v** **a** **s**, um zu sehen, welche Bereiche als Satz erkannt werden. Prüfe dabei insbesondere, wie VIM mit Anführungszeichen und Klammern am Satzende umgeht.

5.2.3. Absatzobjekte

Ein Absatz wird in VIM definiert als eine zusammenhängende Textpassage, die durch mindestens eine Leerzeile von der nächsten Passage getrennt ist.

Die passenden Textobjekte sind:

- **i** **p** innerer Absatz (ohne trennende Leerzeile)
- **a** **p** Absatz inklusive folgender Leerzeile

Typische Operator-Kombinationen:

- **d** **i** **p** löscht den inneren Absatz
- **d** **a** **p** löscht den Absatz inklusive Leerzeile
- **v** **i** **p** wählt den inneren Absatz visuell aus

Übungen

- Lege zwei Absätze an, getrennt durch eine Leerzeile. Setze den Cursor in den ersten Absatz und vergleiche **d** **i** **p** und **d** **a** **p**. Achte darauf, ob die Leerzeile erhalten bleibt oder mit entfernt wird.
- Öffne ein längeres Logfile oder einen Text mit mehreren Absätzen und markiere mit **v** **i** **p** verschiedene Absätze. Prüfe, wie zuverlässig VIM die Absatzgrenzen erkennt.
- Nimm einen Absatz mit unterschiedlicher Einrückung und veränderten Zeilenlängen. Wende **c** **i** **p** an und ersetze den Inhalt durch einen neuen Textblock. Wiederhole die Änderung mit **.** in weiteren Absätzen, um den Effekt in Serie zu testen.

5.2.4. Praxisempfehlungen und Makrohinweise

Wort-, Satz- und Absatzobjekte gehören zu den wichtigsten Bausteinen für strukturiertes Arbeiten:

- c i w ist eines der am häufigsten verwendeten Änderungsobjekte für präzise Wortänderungen.
- c i p eignet sich hervorragend, um ganze Absätze umzubauen (z. B. bei Refactorings oder Umstrukturierungen in Dokumentation).
- v i W ist ideal für technische Elemente wie Pfade, Parameter oder Kommandozeilen, da ein „WORD“ oft eine komplette Einheit darstellt.
- a s ist hilfreich, wenn ein vollständiger Satz inklusive Abschlusszeichen bearbeitet oder verschoben werden soll.

Für Makros sind diese Objekte besonders geeignet, weil sie:

- robust gegen Zeilenumbrüche sind,
- weitgehend unabhängig von der exakten Cursorposition funktionieren,
- sich ideal für automatisierte Umformatierungen und Serienänderungen eignen.

Beispiel für ein typisches Makromuster:

c i p + neuer Absatzinhalt + Esc $\Longrightarrow$ beliebig oft mit . wiederholbar.

5.3. Klammerobjekte

Klammerobjekte gehören zu den wichtigsten strukturellen Textobjekten in VIM. Mit ihnen lassen sich logisch abgegrenzte Bereiche wie Funktions blöcke, mathematische Ausdrücke oder LaTeX-Umgebungen präzise bearbeiten. Sie sind ein essenzielles Werkzeug sowohl für normales Editieren als auch für Makro-Programmierung.

5.3.1. Unterstützte Klammerarten

VIM kennt mehrere Klammerpaare, die als Textobjekte angesprochen werden können. Diese Klammern bilden die Grundlage für Operationen wie c i (, d i { oder v a [:

- ( ) runde Klammern
- { } geschweifte Klammern

- [ ] eckige Klammern
- < > spitze Klammern (nur in einigen Dateitypen / Modi)

Für jedes dieser Klammerpaare existieren zwei systematische Varianten:

- i (, i {, i [, i < → *innerer* Inhalt ohne Begrenzungszeichen
- a (, a {, a [, a < → Bereich *inklusive* der Klammern

Damit lassen sich Operatoren wie d, c, y und auch visuelle Auswahl (v) strukturell kombinieren, z. B.:

- d i (löscht den Inhalt runder Klammern
- c a { ändert einen gesamten Block inklusive Klammern
- v a [wählt einen eckigen Klammerblock komplett aus

5.3.2. Innere Klammerobjekte (i(, i{, i[, i<)

Innere Klammerobjekte beziehen sich nur auf den logischen Inhalt *zwischen* den Begrenzungszeichen – die Klammern selbst bleiben erhalten. Sie sind für präzise Änderungen ideal, weil die Struktur nicht zerstört wird.

Typische Operator-Kombinationen:

- c i (ändert den Inhalt runder Klammern
- c i { ändert den Inhalt geschweifter Klammern
- d i [löscht den inneren Bereich eckiger Klammern
- y i (kopiert nur den Inhalt runder Klammern

Typische Einsatzszenarien:

- Funktionsargumente anpassen, ohne Funktionsnamen oder Klammern anzufassen
- LaTeX-Befehle wie `\command{...}` umschreiben, ohne die Kommando-Struktur zu beschädigen
- Inhalte in JSON-/Konfigurationsblöcken gezielt ersetzen

Übungen

- Lege eine Zeile an:

  ```
  log_debug("status=OK, code=200");
  ```

  Setze den Cursor irgendwo zwischen den Anführungszeichen und verwende c i ”. Ersetze den Inhalt durch `status=FAIL, code=500` Enter.

Kontrolliere, dass nur der Text zwischen den Anführungszeichen geändert wurde und alles andere unverändert blieb.

- Erzeuge eine Funktionszeile:

  ```
  sum(a, b, c);
  ```

  Setze den Cursor in die runde Klammer und benutze [c][i][(], um nur die Argumentliste zu ersetzen, z. B. durch `x, y`[Enter]. Prüfe, dass Funktionsname und Klammern erhalten bleiben.
- Lege ein JSON-ähnliches Fragment an:

  ```
  {"level": "info", "msg": "start"}
  ```

  Setze den Cursor irgendwo innerhalb der geschweiften Klammern und teste nacheinander:
  - [c][i][{] mit einem neuen Inhalt, z. B. `"level": "error"`
  - [d][i][{] gefolgt von [u] (Undo), um den Effekt zu vergleichen

  Beobachte den Unterschied zwischen „Inhalt ersetzen“ und „Inhalt löschen“, während die äußeren Klammern stehen bleiben.

5.3.3. Äußere Klammerobjekte (a(, a{, a[, a<)

Äußere Klammerobjekte beziehen den kompletten Bereich *inklusive* Begrenzungszeichen ein. Im Unterschied zu den inneren Objekten ([i][(], [i][{] usw.) wird hier stets die gesamte strukturierende Einheit bearbeitet.

Typische Operator-Kombinationen:

- [d][a][(] löscht eine komplette runde Klammergruppe inklusive Klammern
- [c][a][{] ersetzt einen Block inklusive geschweifter Klammern
- [v][a][[] wählt einen eckigen Klammerblock vollständig aus
- [y][a][<] kopiert einen Bereich inklusive spitzer Klammern

Das ist besonders nützlich für:

- Entfernen oder Umstrukturieren ganzer Codeblöcke
- Ersetzen kompletter Funktionsaufrufe (inklusive Argumentklammern)
- Verschieben zusammengehöriger Strukturen in andere Dateien/Module

Übungen

- Lege eine Zeile an:

```
log_debug("start", ctx);
```

Setze den Cursor irgendwo innerhalb der runden Klammern und teste:

- **d** **i** **(** nur der Inhalt der Klammern verschwindet
- **d** **a** **(** komplette Klammergruppe inklusive Klammern wird gelöscht

Vergleiche beide Varianten und mach dir klar, wann du welche Form brauchst.

- Erzeuge einen Block mit geschweiften Klammern:

```
if (status == OK) {
    handle_success();
}
```

Setze den Cursor irgendwo im Block und benutze:

- **d** **a** **{** um den gesamten Block (inklusive { und }) zu löschen
- **c** **a** **{** um den Blockinhalt durch eigenen Code zu ersetzen

Achte darauf, dass die äußere Struktur (z. B. das `if`-Statement) bei **c** **a** **{** erhalten bleibt, während der Block neu gefüllt wird.

- Nimm ein Array-Fragment:

```
values[10 + offset]
```

Setze den Cursor in die eckigen Klammern und probiere:

- **v** **i** **[** gefolgt von **y** (kopieren)
- **v** **a** **[** gefolgt von **d** (löschen des kompletten Index-Ausdrucks)

Beobachte den Unterschied zwischen innerem und äußerem Objekt auch im visuellen Modus.

5.3.4. Verschachtelte Strukturen

VIM berücksichtigt die aktuelle Klammertiefe: Steht der Cursor in einer verschachtelten Struktur, wirken **c** **i** **{** und **d** **i** **{** nur auf die jeweils innerste Ebene. Befindet sich der Cursor hingegen auf einer äußeren Klammer, beziehen **c** **a** **{** oder **d** **a** **{** den gesamten Block mit ein.

Typische Situationen:

- Cursor innerhalb einer tiefen Verschachtelung
  - **c** **i** **{** ändert genau den innersten Block
  - **d** **i** **{** löscht nur den Inhalt dieses innersten Blocks

- Cursor auf oder nahe der äußeren Klammer
  - c a { ersetzt den kompletten äußeren Block
  - d a { entfernt die gesamte umschließende Struktur

Damit werden Klammerobjekte zu einem sehr präzisen Werkzeug für Sprachen mit klaren Blockstrukturen, etwa C, Rust, Go, JavaScript, JSON, Lua oder LaTeX-Umgebungen.

Übungen

- Lege folgenden Code an:

```
if (status == OK) {
    while (retries > 0) {
        handle_job(job);
        retries--;
    }
}
```

  - Setze den Cursor irgendwo innerhalb von `handle_job(job);` und führe c i { aus. → Es wird nur der *innere* {...}-Block der `while`-Schleife geändert.
  - Setze den Cursor auf die äußere { des `if`-Blocks und führe d a { aus. → Der komplette `if`-Block (inklusive geschweifter Klammern) verschwindet.
- Teste ein JSON-Fragment:

```
{
  "user": {
    "name": "Alice",
    "active": true
  }
}
```

  - Cursor in der inneren `"user"`-Struktur → d i { löscht nur den Inhalt des inneren Objekts.
  - Cursor in der äußeren {...}-Struktur → d a { entfernt das komplette JSON-Objekt.

5.3.5. Klammerobjekte im Visual Mode

Alle Klammerobjekte stehen auch im Visual Mode zur Verfügung. Die Kombination aus v und i (/ a (usw. erzeugt extrem präzise Markierungen mit minimaler Tastenfolge.

Typische Befehle:

- v i (wählt nur den inneren Klammerinhalt
- v a { markiert den gesamten Block inklusive { und }
- v i [markiert den Inhalt eckiger Klammern
- v a < wählt einen Bereich inklusive spitzer Klammern

Nach der visuellen Auswahl können alle üblichen Befehle eingesetzt werden, etwa d, c, y, = (Einrücken) oder : für Ex-Kommandos auf dem markierten Bereich.

Übungen

- Lege eine Zeile an:

  ```
  result = func(a, b, c[0]);
  ```

  - Cursor irgendwo zwischen den runden Klammern → v i (und anschließend y zum Kopieren des Argumentbereichs.
  - Undo (u), dann v a (und d, um den gesamten Funktionsaufruf inklusive Klammern zu entfernen.
- Verwende ein Array-Beispiel:

  ```
  buffer[start + offset * 2]
  ```

  - Cursor im Ausdruck innerhalb der eckigen Klammern → v i [und c, um den Index-Ausdruck strukturiert zu ersetzen.
  - Wiederhole die Operation mit v a [, um zu sehen, wie sich das Einbeziehen der Klammern auswirkt.

5.3.6. Makrohinweise

Klammerobjekte sind für Makros besonders stabil, weil sie sich an der Struktur orientieren – nicht an sichtbaren Zeilenumbrüchen oder Einrückungen. Entscheidend ist, dass i (, a { usw. immer denselben logischen Bereich treffen, egal wie der Code formatiert ist.

Eigenschaften:

- unabhängig von Zeilenumbrüchen
- weitgehend unabhängig von Whitespace und Einrückung
- funktionieren zuverlässig in tief verschachtelten Strukturen

Typische Makro-Bausteine:

- c i (Argumentliste oder Ausdruck in runden Klammern ersetzen, egal wie lang oder umgebrochen
- d a { kompletten Codeblock löschen, auch bei variabler Einrückung oder zusätzlichen Leerzeilen
- c i [Index-/Array-Ausdrücke austauschen, ohne die Klammern selbst anzufassen

Grundregel für Makros: **Wenn ein Bereich durch Klammern strukturell begrenzt ist, nutze Textobjekte (i (, a { etc.) statt Cursorbewegungen mit Pfeiltasten oder f/t.**

5.3.7. Zusammenfassung

Klammerobjekte fassen das Strukturprinzip von VIM kompakt zusammen:

- i X = nur der *Inhalt* einer Klammer (z. B. i (, i { , i [, i <)
- a X = Klammer *plus* Inhalt (z. B. a (, a { , a [, a <)
- funktioniert ideal in Code, LaTeX, JSON, Markdown und jeder Form verschachtelter Struktur
- sehr gut für Makros geeignet, weil sie unabhängig von Formatierung und Zeilenumbrüchen arbeiten

Merksatz: i X **für „nur den Inhalt“,** a X **für „alles drumherum“.**

6. Suche und Ersetzen

6.1. Einfache Suche

Die grundlegende Suche in VIM erfolgt mit dem Schrägstrich:

- [/] `muster` sucht vorwärts nach `muster`
- [?] `muster` sucht rückwärts nach `muster`

Nach einer Suche kannst du mit [n] und [N] durch die Treffer navigieren:

- [n] nächster Treffer in *Suchrichtung*
- [N] vorheriger Treffer (entgegengesetzte Richtung)

VIM übernimmt das zuletzt verwendete Suchmuster automatisch für Operatoren und Bewegungen:

- [d] [/] `muster` [Enter] löscht bis zum nächsten Treffer von `muster`
- [d] [?] `muster` [Enter] ändert rückwärts bis zum vorherigen Treffer

6.1.1. Besonderheiten der einfachen Suche

- Die Suche ist standardmäßig *inkrementell*: Bereits beim Tippen des Musters zeigt VIM Zwischenergebnisse an (abhängig von `'incsearch'`).
- Groß-/Kleinschreibung ist relevant, solange `'ignorecase'` nicht gesetzt ist bzw. `'smartcase'` aktiv wird.
- Mit ^ im Suchmuster werden nur Treffer am Zeilenanfang gefunden.
- Mit $ im Suchmuster werden nur Treffer am Zeilenende gefunden.

6.1.2. Such-Historie

VIM speichert alle Suchmuster in einer History. Du kannst frühere Einträge bequem wiederverwenden:

- [/] + [↑] frühere Suchmuster durchlaufen
- `:history /` komplette Suchhistorie anzeigen

Damit lassen sich komplexe Muster schnell erneut verwenden, ohne sie noch einmal eintippen zu müssen.

6.1.3. Schnelle Wiederholung

- n und N funktionieren auch *nach* Operatoren wie d / muster Enter oder d ? muster Enter.
- . wiederholt Such-basierte Änderungen inklusive der zugrunde liegenden Suchbewegung.

Die einfache Suche bildet damit die Grundlage aller späteren Suchmechanismen: Bewegungen, Ersetzungen und Makros bauen direkt auf diesem Verhalten auf.

6.2. Rückwärts- und Vorwärtssuche

Die Suche in VIM ist immer richtungsgebunden. Es wird klar unterschieden zwischen Vorwärts- und Rückwärtssuche. Beide Varianten verwenden dieselben Navigationsbefehle, aber das *Startsignal* legt die Richtung fest:

- / muster Enter startet eine *Vorwärtssuche*
- ? muster Enter startet eine *Rückwärtssuche*

Die aktuelle Suchrichtung bleibt so lange erhalten, bis eine neue Suche gestartet wird.

6.2.1. Treffer-Navigation

Nach einer Suche sind die folgenden Befehle aktiv:

- n springt zum nächsten Treffer in *derselben Richtung*
- N springt zum nächsten Treffer in der *entgegengesetzten Richtung*

Wichtig: Die Richtung von n und N hängt *immer* von der zuletzt ausgeführten Suchart ab — nicht vom Inhalt des Musters. Wurde zuletzt mit / gesucht, bewegt sich n vorwärts und N rückwärts; bei ? ist es genau umgekehrt.

Übungen

- Suche in einer Beispieldatei nach ERROR mit [/]ERROR[Enter]. Nutze [n] und [N], um gezielt vor und zurück zu springen. Beobachte, wie sich die Richtung ändert, wenn du zwischendurch [?]ERROR[Enter]verwendest.
- Suche nach einem Wort mit [/]`foo`[Enter], bewege den Cursor zwischendurch per [j] / [k], ohne neu zu suchen, und nutze nur [n] / [N]. Kontrolliere, dass die Suchrichtung trotzdem konsistent bleibt.

6.2.2. Operatoren mit richtungsgebundener Suche

Suchbefehle können direkt als Bewegung für Operatoren verwendet werden. Die Kombination lautet immer:
Operator gefolgt von [/]`muster`[Enter] oder [?]`muster`[Enter]Typische Beispiele:

- [d][/]`muster`[Enter] löscht bis *exklusive* des nächsten Vorwärtstreffers
- [d][?]`muster`[Enter] löscht bis *exklusive* des vorherigen Rückwärtstreffers
- [c][/]`muster`[Enter] ändert den Bereich bis zum nächsten Vorwärtstreffer
- [y][?]`muster`[Enter] kopiert den Bereich bis zum vorherigen Rückwärtstreffer

Die Suche fungiert dabei als Motion: VIM bewegt den Cursor deterministisch bis zum nächsten Treffer in der jeweiligen Richtung und wendet den Operator genau auf diesen Bereich an. Es werden keine Zeichen „blind“ gezählt, sondern immer bis zur strukturell definierten Marke gearbeitet.

Übungen

- Lege ein kleines Logfragment an, z. B.:

```
INFO start
DEBUG check
WARN retry
ERROR stop
```

  Setze den Cursor an den Anfang von `INFO` und teste [d][/]DEBUG[Enter]. Kontrolliere, dass genau der Bereich `INFO start` gelöscht wird.

- Wiederhole die Übung rückwärts: Setze den Cursor auf ERROR und verwende d ? WARN Enter. Beobachte, welcher Bereich entfernt wird und wo der Cursor danach steht.
- Kombiniere Suche und Punkt-Operator: Führe d / ERROR Enter in einem längeren Log aus und wiederhole die Löschung weiterer Blöcke mit . . Achte darauf, dass die Kombination „Operator + Suche“ als eine einzige Änderung behandelt wird.

6.2.3. Verhalten bei Fehltreffern

Falls kein Treffer gefunden wird:

- gibt VIM eine Fehlermeldung aus (beispielsweise `E486: Pattern not found`)
- bleiben n und N wirkungslos, bis erneut ein Suchbefehl ausgeführt wurde

Ein Fehltreffer ändert jedoch *nicht* die Suchrichtung: Wurde zuletzt / `muster` verwendet, bleibt die Richtung vorwärts; wurde zuletzt ? `muster` verwendet, bleibt die Richtung rückwärts.

6.2.4. Historie und Wiederholung

Die Suchhistorie ist für Vorwärts- und Rückwärtssuche identisch:

- / oder ? plus ↑ erlaubt das Wiederverwenden alter Suchmuster
- `:history /` zeigt alle bisherigen Suchmuster — unabhängig von der Richtung

Auch Operatoren profitieren davon:

- . wiederholt Such-basierte Änderungen korrekt
- n und N funktionieren auch direkt nach Operatoren

Vorwärts- und Rückwärtssuche bilden zusammen die Grundlage für viele strukturorientierte Bewegungen — insbesondere in Kombination mit Operatoren, Makros, Textobjekten und regulären Ausdrücken.

6.3. Globale Ersetzungen

Globale Ersetzungen gehören zu den mächtigsten Funktionen in VIM. Sie basieren auf der Kommandozeile und folgen der Grundform

```
:s/alt/neu/
```

Ohne weitere Optionen wird dabei nur der *erste Treffer der aktuellen Zeile* ersetzt.

6.3.1. Mehrere Ersetzungen pro Zeile

Um *alle* Vorkommen in der aktuellen Zeile zu ersetzen, wird der globale Schalter `g` ergänzt:

```
:s/alt/neu/g
```

`g` bedeutet hier: alle Treffer in der Zeile werden ersetzt, nicht nur der erste.

6.3.2. Ersetzungen im gesamten Dokument

Mit einem Adressbereich kann der Geltungsbereich erweitert werden. Die gesamte Datei wird mit `%` adressiert:

```
:\%s/alt/neu/g
```

`%` entspricht dem Bereich `1 , $` (von der ersten bis zur letzten Zeile). Damit werden *alle* Vorkommen von `alt` in der gesamten Datei durch `neu` ersetzt.

6.3.3. Interaktive Ersetzungen

Ersetzungen lassen sich interaktiv bestätigen. Dazu wird der Schalter `c` („confirm“) ergänzt:

```
:\%s/alt/neu/gc
```

Bei jedem Treffer fragt VIM nach:

- `y` diesen Treffer ersetzen
- `n` diesen Treffer überspringen
- `a` alle weiteren Treffer ersetzen (ohne weitere Nachfrage)
- `q` gesamte Ersetzung abbrechen
- `l` diesen Treffer ersetzen und dann abbrechen („last“)

Übungen

- Lege eine Datei mit mehreren Zeilen an, die das Wort `foo` enthalten. Ersetze in einer einzelnen Zeile nur das erste `foo` durch `bar` mit `:s/foo/bar/` und anschließend alle Vorkommen in der Zeile mit `:s/foo/bar/g`. Beobachte den Unterschied.
- Erzeuge einen kurzen Text mit mehrfach vorkommendem TODO. Ersetze alle TODO der gesamten Datei durch DONE mit `:\%s/TODO/DONE/g`.
- Wiederhole das vorherige Beispiel mit `:\%s/TODO/DONE/gc` und entscheide bei jedem Treffer, ob ersetzt werden soll. Teste bewusst die Optionen `y`, `n`, `a`, `q` und `l`.

6.3.4. Bereichsbezogene Ersetzungen

Ersetzungen lassen sich auf beliebige Zeilenbereiche einschränken. Typische Adressierungen sind:

- `:10,20s/alt/neu/g`
  ersetzt alle `alt` durch `neu` in den Zeilen 10 bis 20.
- `:.,\$s/alt/neu/g`
  ersetzt alle `alt` von der aktuellen Zeile (`.`) bis zur letzten Zeile ($).
- `:.,+5s/alt/neu/g`
  ersetzt alle `alt` von der aktuellen Zeile bis fünf Zeilen darunter.

Adressbereiche können beliebig mit `s`-Kommandos kombiniert werden und sind ein zentrales Werkzeug für kontrollierte Massenänderungen.

Übungen

- Erzeuge eine Datei mit mindestens 30 Zeilen und mehreren Vorkommen von DEBUG. Ersetze DEBUG nur in den Zeilen 5 bis 15 durch `INFO` mit `:5,15s/DEBUG/INFO/g`.
- Ersetze ein Muster nur vom aktuellen Cursor bis zum Dateiende mit `:.,\$s/foo/bar/g` und beobachte, welche Bereiche unverändert bleiben.

6.3.5. Ersetzungen mit Backreferences

Backreferences erlauben es, Teile eines Treffers im Ersatzmuster wieder zu verwenden. Dadurch lassen sich z. B. Wörter vertauschen.

Beispiel: Zwei Wörter vertauschen (vereinfachter Fall: zwei „Wörter“ getrennt durch Leerraum):

```
:\%s/(\textbackslash w+)\textbackslash
s\textbackslash +(\textbackslash
w+)/\textbackslash 2, \textbackslash 1/
```

Bedeutung:

- `(\w+)` erste Wortgruppe (Gruppe 1)
- `\s\+` ein oder mehrere Leerraumzeichen
- `(\w+)` zweite Wortgruppe (Gruppe 2)
- `\1`, `\2` Backreferences auf die erste bzw. zweite Gruppe

Der Ausdruck macht aus `foo bar` die Folge `bar, foo`.

Übungen

- Lege eine Zeile wie `Vorname Nachname` an und wende obigen Befehl an. Kontrolliere, dass die Reihenfolge zu `Nachname, Vorname` wird.
- Variiere das Muster auf `:\%s/\textbackslash w+,\textbackslash s\textbackslash +\textbackslash w+/\textbackslash 2 \textbackslash 1/` und teste es auf Zeilen der Form `Nachname, Vorname`.
- Beobachte mit **u** und **.**, wie VIM die Ersetzung als Änderung speichert und wiederholen kann.

6.3.6. Ersetzungen in Makros

Globale Ersetzungen sind makrotauglich, wenn:

- das gesamte Kommando deterministisch ist,
- keine Rückfragen auftreten (also ohne `c`),
- das Suchmuster statisch bleibt.

Empfohlen für Makros sind vollständig spezifizierte Ersetzungen, z. B.:

```
:\%s/alt/neu/g
```

oder für einen definierten Bereich:

```
:10,20s/alt/neu/g
```

Solche Kommandos verhalten sich auf jeder Datei mit derselben Struktur gleich und sind damit gut in Makros einsetzbar.

6.4. Ersetzungs-Syntax

Die Ersetzungsbefehle in VIM folgen einer festen Grundstruktur:

```
:Adresse s/Muster/Ersatz/Flags
```

Die Bestandteile im Einzelnen:

- **Adresse** Bereich, auf den sich die Ersetzung bezieht (z. B. `:\%`, `:10,20`, `:,\$`)
- **Muster** Suchausdruck (normaler String oder Regex)
- **Ersatz** Zielstring, optional mit Backreferences (z. B. `:\textbackslash 1`, `:\textbackslash 2`)
- **Flags** Steuerung des Verhaltens (z. B. `:g`, `:c`, `:i`)

Die meisten Praxisbeispiele lassen sich auf diese Grundform zurückführen.

6.4.1. Adresse

Die Adresse bestimmt, auf welchen Bereich der Befehl wirkt. Sie steht immer vor dem `s`:

- `:s/alt/neu/` nur aktuelle Zeile
- `:1,10s/alt/neu/` Zeile 1 bis 10
- `:.,\$s/alt/neu/` von aktueller Zeile (`:.`) bis Dateiende (`:\$`)
- `:\%s/alt/neu/` gesamte Datei (`:\%` = 1,$)
- `:/Muster/s/alt/neu/` erste Zeile ab Cursor, die „Muster“ enthält

Adressierung ist ein eigenes Mini-Sprachsystem: Zeilennummern, Bereiche (`:1,10`, `:.,\$`) und Suchmuster (`:/Muster/`) lassen sich frei kombinieren und sind damit auch für Makros sehr gut geeignet.

6.4.2. Suchmuster

Das Suchmuster folgt direkt nach dem ersten Schrägstrich und endet am nächsten Schrägstrich. Es kann als einfaches Literal oder als regulärer Ausdruck interpretiert werden:

- `foo` findet das Literal *foo*

- `\w+` ein oder mehrere Wortzeichen (Buchstaben, Ziffern, Unterstrich)
- `^bar` Treffer nur am Zeilenanfang
- `baz$` Treffer nur am Zeilenende
- `colou\r?` `colour` oder `color`

VIM verwendet standardmäßig eine Mischung aus Literal- und Regex-Suche: normale Buchstaben werden wörtlich genommen, typische Regex-Zeichen wie `.`, `*`, `[ ]`, `^`, `$` oder `\w` haben eine Sonderbedeutung. Erweiterte reguläre Ausdrücke und Feinheiten der „magic"-Einstellungen werden in einem späteren Abschnitt behandelt.

6.4.3. Ersatztext

Der Ersatzteil steht zwischen dem zweiten und dritten Trennzeichen und bestimmt, wodurch der gefundene Text ersetzt wird:

- `neu` ersetzt den Treffer wörtlich durch *neu*
- `\1` Backreference auf die erste Klammergruppe aus dem Suchmuster
- `\2`, `\3` weitere Gruppen (zweite, dritte, ...)
- `\0` der komplette Treffer (inklusive aller Gruppen)
- `&` ebenfalls der komplette Treffer

Wichtige Punkte:

- Backreferences (`\1`, `\2`, ...) funktionieren nur im *Ersatzteil*, nicht im Suchmuster.
- Soll im Ersatzteil ein wörtliches `&` erscheinen, muss es mit `\&` maskiert werden.
- Enthält der Ersatz selbst einen Schrägstrich `/`, muss dieser als `\/` geschrieben werden.

6.4.4. Flags

Flags steuern das Verhalten des Ersatzvorgangs:

- `g` global: alle Treffer in einer Zeile ersetzen
- `c` confirm: Rückfrage vor jedem Treffer
- `i` „ignorecase": Groß-/Kleinschreibung ignorieren
- `I` Groß-/Kleinschreibung erzwingen (überschreibt `i`)
- `n` nur zählen, nicht ersetzen

Beispiel:

Ex%s/foo/bar/gi

Dieser Befehl setzt alle `foo` im gesamten Dokument durch `bar` – ohne auf Groß-/Kleinschreibung zu achten (`i`) und mit globaler Ersetzung pro Zeile (g).

6.4.5. Alternative Trennzeichen

Da der Schrägstrich / häufig selbst im Muster vorkommt (z. B. in Pfaden oder URLs), erlaubt VIM alternative Separatoren. Anstelle von / kann fast jedes andere Zeichen verwendet werden:

- Exs#pfad/alt#pfad/neu#g
- Exs@/home/user@/mnt/disk@g

Vorteil: Innerhalb von Muster und Ersatz müssen Schrägstriche nicht mehr mühselig mit `\/` maskiert werden. Nur das jeweils gewählte Trennzeichen (#, @, ...) muss weiterhin maskiert werden, falls es selbst im Muster oder im Ersatz vorkommt.

6.4.6. Ersetzungs-Besonderheiten

- Ein leeres Suchmuster wiederholt das letzte verwendete Muster. Beispiel:

  ```
  :s//neu/g
  ```

  Hier wird das zuletzt gesuchte Muster (z. B. aus `:/foo`) durch `neu` ersetzt.
- Ein leerer Ersatzteil löscht die Treffer. Beispiel:

  ```
  :\%s/foo//g
  ```

  entfernt alle Vorkommen von `foo` im gesamten Dokument.
- & im Ersatzteil steht für den kompletten Treffer. Beispiel:

  ```
  :\%s/foo/\&\_alt/g
  ```

  macht aus `foo` den Text `foo_alt`.
- `\r` erzeugt einen echten Zeilenumbruch im Ersatz. Beispiel:

  ```
  :\%s/;/\textbackslash r/g
  ```

  ersetzt jedes Semikolon durch einen Zeilenumbruch.

Die Ersetzungs-Syntax in VIM bildet damit ein vollwertiges Mini-Sprachsystem: flexibel, präzise und sehr gut für Makros geeignet.

7. Visual Mode

7.1. Zeichenweise Auswahl

Die zeichenweise Auswahl ist der grundlegendste Visual-Mode in VIM. Sie markiert einzelne Zeichen oder lineare Textbereiche zwischen zwei Cursorpositionen. Dieser Modus ist ideal für feingranulare Bearbeitungen, bei denen exakt definierte Zeichenbereiche mit Operatoren wie **d**, **c**, **y** oder **>** bearbeitet werden sollen.

Typische Anwendungsfälle:

- einzelne Wörter oder Wortteile markieren und löschen/ändern
- kurze Codefragmente selektiv einrücken oder verschieben
- gezielte Kopien kleiner Textbereiche erstellen

7.1.1. Aktivieren des Visual Mode

Der zeichenweise Visual Mode wird mit **v** aktiviert.

Ab dem Druck auf **v** befindet sich VIM im Visual Mode. Die Auswahl reicht immer vom *Ankerpunkt* (der ursprünglichen Cursorposition) zur aktuellen Cursorposition. Jede weitere Bewegung des Cursors mit **h**, **j**, **k**, **l**, Wortbewegungen oder Sprüngen vergrößert oder verkleinert die Markierung entsprechend.

Wichtige Tastenkombinationen:

- **v** zeichenweise Visual Mode aktivieren / umschalten
- **o** Anfang und Ende der Markierung vertauschen
- **Esc** Visual Mode beenden, Auswahl verwerfen

Die Auswahl kann anschließend mit einem Operator kombiniert werden, etwa:

- **d** markierten Bereich löschen
- **c** markierten Bereich ändern (Visual-Change)
- **y** markierten Bereich kopieren
- **>** / **<** markierten Bereich ein- oder ausrücken

Übungen

- Lege eine Zeile an, z. B.:

  ```
  foo_bar_baz_qux
  ```

  Setze den Cursor auf das erste `f`, drücke **v** und bewege dich mit **l** schrittweise nach rechts. Beobachte, wie sich die Auswahl zeichenweise erweitert. Beende mit **Esc**.
- Wiederhole die Übung und drücke nach **v** direkt **w** oder **e**, um die Auswahl wortweise zu vergrößern. Lösche die Auswahl anschließend mit **d**.
- Teste **o**: Aktiviere den Visual Mode mit **v**, erweitere die Auswahl mit **j** über mehrere Zeilen und drücke dann **o**, um zu sehen, wie der Cursor zwischen Anfang und Ende der Markierung springt.

7.1.2. Bewegen der Auswahl

Die Auswahl wird mit den normalen Bewegungsbefehlen verändert. Jede Cursorbewegung verschiebt das *Ende* der Markierung:

- **h** Auswahlzeichen nach links verschieben/verkürzen
- **l** Auswahlzeichen nach rechts verschieben/verlängern
- **j** Auswahl um eine Zeile nach unten erweitern
- **k** Auswahl um eine Zeile nach oben erweitern

Auch komplexere Bewegungen funktionieren im Visual Mode und passen die Markierung entsprechend an:

- **w**, **e**, **b** wortweise Bewegung
- **(**, **)** satzweise Bewegung
- **{**, **}** absatzweise Bewegung
- **f** **X**, **t** **X**, **F** **X**, **T** **X** bis/zum Zeichen X

Damit lassen sich ganze strukturelle Bereiche präzise erfassen, ohne die Maus zu benutzen. Besonders in Kombination mit Textobjekten (**i** **w**, **i** **(**, **i** **p** usw.) wird der Visual Mode zu einem sehr flexiblen Werkzeug.

Übungen

- Lege eine Zeile an, z. B.:

  ```
  alpha beta gamma delta
  ```

Setze den Cursor auf das `b` von `beta`, drücke **v** und bewege dich mit **w**, **b**, **e**. Beobachte, wie sich die Auswahl wortweise verändert.

- Erzeuge zwei Zeilen Text untereinander und aktiviere den Visual Mode auf der ersten Zeile mit **v**. Bewege den Cursor mit **j** und **k** zwischen den Zeilen hin und her. Kontrolliere, wie sich die Auswahl vertikal verhält.
- Nimm eine längere Zeile, z. B.:

  ```
  foo = bar + baz * qux;
  ```

  Setze den Cursor auf das Anfangs-`f`, drücke **v** und verwende **f** **;** bzw. **t** **;** im Visual Mode. Prüfe, wie sich die Auswahl bis/zum Semikolon erweitert und lösche sie mit **d**.

7.1.3. Anwenden von Operatoren

Jeder Operator, der im Normal Mode funktioniert, kann auch im Visual Mode auf die aktuelle Auswahl angewendet werden. Anstatt „Operator + Motion“ gibst du im Visual Mode einfach nur den Operator ein – die bestehende Markierung ersetzt die Motion.

Typische Operatoren im Visual Mode:

- **d** markierten Bereich löschen
- **c** Bereich ändern (löschen + Insert Mode)
- **y** Bereich kopieren (yanken)
- **>** Bereich einrücken
- **<** Einrückung rückgängig machen

Wichtig dabei:

- Die Auswahl definiert die *Zielregion*, nicht mehr die Motion.
- Counts (**3** **d** usw.) sind im Visual Mode unüblich – die Größe der Auswahl bestimmt den Umfang.
- Nach **c** wechselt VIM in den Insert Mode; **Esc** kehrt in den Normal Mode zurück und beendet die visuelle Auswahl.

Übungen

- Erzeuge drei Zeilen Text, z. B.:

```
alpha
beta
gamma
```

Setze den Cursor auf `beta`, drücke [v], gehe mit [j] auf `gamma` und lösche die Auswahl mit [d]. Prüfe, dass `beta` und `gamma` komplett entfernt wurden.

- Lege einen eingerückten Block an (z. B. vier Leerzeichen vor jeder Zeile). Markiere den Block mit [V] (zeilenweiser Visual Mode) und drücke [<]. Kontrolliere, dass die Einrückung um genau eine Stufe reduziert wurde. Wiederhole mit [.], um den Effekt zu verinnerlichen.
- Erzeuge eine Zeile wie:

  ```
  foo = bar + baz;
  ```

  Markiere `bar + baz` mit dem Visual Mode ([v] und Bewegungen). Drücke [c] und ersetze den Ausdruck durch `qux`[Esc]. Wiederhole die Übung an anderen Stellen, bis klar ist, wie sich [c] im Visual Mode von [c][w]/[c][i][w] im Normal Mode unterscheidet.

7.1.4. Spezialfall: Wiederholbarkeit

Die zeichenweise Auswahl ist nur eingeschränkt mit [.] wiederholbar. VIM merkt sich zwar die *Änderung*, aber nicht den exakten Verlauf der visuellen Auswahl. Bei einer Wiederholung mit [.] wird daher oft nur die Operation nachvollzogen, während sich die Auswahlform anders verhält als erwartet.

Praktisch bedeutet das:

- Änderungen aus dem Visual Mode lassen sich mit [.] nur bedingt reproduzierbar wiederholen.
- Bereits kleine Layout- oder Cursorverschiebungen führen zu anderem Ergebnis.

Makro-Empfehlung:

- Visual Mode in Makros so sparsam wie möglich einsetzen.
- Stattdessen strukturierte Bewegungen oder Textobjekte nutzen ([d][i][w], [d][i][p], [c][i][(], [d][a][{] usw.).
- Wann immer möglich: „Operator + Motion/Textobjekt“ im Normal Mode verwenden, nicht visuelle Auswahl.

Übungen

- Erzeuge drei ähnliche Zeilen:

  ```
  foo(alpha);
  foo(beta);
  foo(gamma);
  ```

  Markiere in der ersten Zeile mit [v] und Bewegungen den Inhalt der Klammer und lösche ihn mit [d]. Versuche nun, mit [j] und [.] die Änderung auf die nächsten Zeilen zu übertragen. Beobachte, dass das Verhalten weniger vorhersehbar ist als bei [c][i][(] + [.].
- Wiederhole die Übung mit [c][i][(] statt Visual Mode:
  1. Cursor irgendwo zwischen den Klammern in der ersten Zeile.
  2. [c][i][(] `delta`[Esc]
  3. Mit [j] in die nächste Zeile springen und [.] drücken.

  Vergleiche die Stabilität der beiden Ansätze und notiere dir, warum Textobjekte in Makros vorzuziehen sind.

7.1.5. Praktische Anwendungsfälle

- einzelne Zeichen oder kurze Sequenzen ersetzen (z. B. [v] + Bewegung + [r][X] für ein einheitliches Ersatzzeichen)
- „fragile“ Bereiche markieren, wenn kein passendes Textobjekt existiert (z. B. mitten in einer komplexen Regex oder URL)
- komplexe Muster erfassen, die nicht sauber strukturell definiert sind (gemischte Kommentare, Prosa mit Sonderzeichen, Logzeilen mit vielen Spalten)

Übungen

- Lege eine Zeile an:

  ```
  foo-123-bar-456-baz
  ```

  Markiere mit [v] und Bewegungen genau `123-bar-456` und ersetze die Markierung mit [r][X]. Überprüfe, dass die Struktur `foo-XXX-baz` erhalten bleibt und kein Zeichen zu wenig/zu viel ersetzt wurde.
- Erzeuge eine unübersichtliche Logzeile:

  ```
  2025-11-18 12:34:56 INFO [user=alice]
  status=OK path=/api/v1/test
  ```

Markiere mit v ab `[user=alice]` bis vor `path=` und kopiere den Bereich mit y. Füge ihn darunter mit p ein und ändere nur den Usernamen. Notiere dir, warum hier eine zeichenweise Auswahl sinnvoller ist als ein grobes Textobjekt.

7.1.6. Zusammenfassung

- v aktiviert die zeichenweise Auswahl.
- Normale Bewegungen (h, j, k, l, Wort- und Klammerbewegungen) erweitern oder verkleinern die Markierung.
- Operatoren wie d, c, y, <, > wirken auf den gesamten markierten Bereich.
- Im Makro-Kontext ist der Visual Mode nur eingeschränkt empfehlenswert; strukturelle Bewegungen und Textobjekte sind in der Regel robuster.

7.2. Zeilenweise Auswahl

Die zeilenweise Auswahl (Visual Line Mode) markiert immer ganze Zeilen. Sie eignet sich besonders für strukturiertes Arbeiten an Absätzen, Codeblöcken, Listen oder Konfigurationsdateien. Im Gegensatz zur zeichenweisen Auswahl musst du dich nicht um exakte Spaltenpositionen kümmern – jede Operation wirkt immer auf komplette Zeilen.

Gerade für wiederholbare Aktionen und Makros ist der Visual Line Mode robuster als die zeichenweise Auswahl, weil sich kleine Verschiebungen in der Horizontalen nicht auf das Ergebnis auswirken.

7.2.1. Aktivieren des Visual Line Mode

Der zeilenweise Visual Mode wird mit V aktiviert.

Sofort wird die gesamte aktuelle Zeile markiert, unabhängig von der Spaltenposition des Cursors. Mit weiteren Bewegungen erweiterst oder verkleinerst du die Auswahl immer in ganzen Zeilen. Ein erneutes V oder Esc beendet den Visual Line Mode und bringt dich zurück in den Normal Mode.

7.2.2. Erweitern der Auswahl

Die zeilenweise Markierung wird mit den normalen Bewegungsbefehlen vergrößert oder verkleinert. Entscheidend: Die Auswahl bleibt immer zeilenweise, egal in welcher Spalte der Cursor steht.

Typische Bewegungen im Visual Line Mode:

- **j** Markierung um eine Zeile nach unten erweitern
- **k** Markierung um eine Zeile nach oben erweitern
- **G** von der Startzeile bis zur letzten Zeile markieren
- **g** **g** von der aktuellen Zeile bis zum Dateianfang markieren
- **/** `muster` **Enter** Markierung bis zur nächsten Zeile mit „muster“ erweitern

Auch Textobjekte lassen sich im Visual-Kontext nutzen:

- **v** **i** **p** inneren Absatz markieren (ohne zusätzliche Leerzeilen)
- **v** **a** **p** gesamten Absatz inklusive Leerzeile markieren

Gerade **/** `muster` **Enter** in Kombination mit **V** ist sehr mächtig: Du kannst schnell einen ganzen Block von einer Startzeile bis zu einem logischen Marker (z. B. „ERROR“, „TODO“, „END“) einrahmen.

Übungen

- Lege mehrere Zeilen an, z. B. ein paar Code- oder Textblöcke. Aktiviere mit **V** die zeilenweise Auswahl und erweitere die Markierung mit **j**, **k**, **G** und **g** **g**. Beobachte, dass stets ganze Zeilen markiert werden, unabhängig von der Spaltenposition.
- Schreibe einen kleinen Text mit mehreren Absätzen (durch Leerzeilen getrennt). Setze den Cursor in einen Absatz und vergleiche **v** **i** **p** und **v** **a** **p** im Visual Mode: Achte darauf, wann die Leerzeile mit markiert wird und wann nicht.
- Erzeuge ein Log-Snippet mit Zeilen, die „INFO“, „WARN“ und „ERROR“ enthalten. Setze den Cursor auf die erste „INFO“-Zeile, drücke **V** und erweitere dann mit **/** ERROR **Enter** die Auswahl. Prüfe, ob der gesamte Block von „INFO“ bis „ERROR“ korrekt markiert wurde.

7.2.3. Operatoren im Visual Line Mode

Im Visual Line Mode wirken alle Operatoren immer auf ganze Zeilen – unabhängig davon, in welcher Spalte der Cursor steht. Sobald die Auswahl mit **V** begonnen wurde, werden alle markierten Zeilen als Einheit behandelt.

Typische Operatoren im Visual Line Mode:

- **d** markierte Zeilen löschen
- **c** markierte Zeilen ändern (löschen + Insert Mode)
- **y** markierte Zeilen kopieren
- **>** markierte Zeilen (gemäß `:shiftwidth`) einrücken
- **<** Einrückung der markierten Zeilen verringern
- **=** Zeilen automatisch neu einrücken (Auto-Indent)

Gerade in Codeblöcken (C, Python, Lua, LaTeX, JSON ...) ist der Visual Line Mode damit ideal, um ganze Blöcke zu verschieben, neu einzurücken oder in ein anderes Register zu kopieren.

Übungen

- Lege einen kleinen Codeblock an (3–5 Zeilen), setze den Cursor in die mittlere Zeile und aktiviere mit **V** den Visual Line Mode. Erweitere die Auswahl mit **j** nach unten und teste nacheinander **>**, **<** und **=**. Beobachte, wie sich die Einrückung des gesamten Blocks verhält.
- Schreibe mehrere Konfigurationszeilen (z. B. `key=value`) und markiere sie mit **V** + **j**. Kopiere sie mit **y** und füge sie mit **p** an anderer Stelle ein. Achte darauf, dass immer ganze Zeilen kopiert und eingefügt werden.
- Erzeuge einen gemischten Text (Kommentarzeilen, Code, Leerzeilen). Markiere einen größeren Block mit **V** und lösche ihn mit **d**. Nutze anschließend **u** (Undo), um den Block zurückzuholen, und probiere dasselbe mit **c**, um den Block direkt neu zu schreiben.

7.2.4. Beispielszenarien

- schnell mehrere Zeilen löschen oder verschieben
- Absätze komplett selektieren und umformatieren
- Codeblöcke sauber einrücken oder ausrichten
- große Textteile zur Weiterverarbeitung kopieren

7.2.5. Wiederholbarkeit

Im Gegensatz zum zeichenweisen Visual Mode sind viele zeilenweise Operationen mit `.` gut wiederholbar. Wiederholt wird die letzte Änderung inklusive verwendeter Bewegung und Count.

Beispiel:

- `V` `j` `j` `>` drei Zeilen einrücken
- `.` wiederholt die Einrückung auf die nächsten drei Zeilen

7.2.6. Makrohinweise

Für Makros ist die zeilenweise Auswahl oft die bevorzugte Form des Visual Mode:

- stabil unabhängig von der Spaltenposition
- robust gegen gemischte Einrückung
- ideal für Blockoperationen (ganze Zeilen)

Wo möglich, sollte in Makros `V` statt `v` verwendet werden, um unklare, zeichenbasierte Selektionen zu vermeiden.

7.2.7. Zusammenfassung

- `V` aktiviert die zeilenweise Auswahl
- die Auswahl wird über normale Bewegungen oder Suchmuster erweitert
- Operatoren wirken immer auf vollständige Zeilen
- deutlich makrostabiler als die zeichenweise Auswahl mit `v`

7.3. Blockweise Auswahl

Die blockweise Auswahl (Visual Block Mode) ermöglicht rechteckige Markierungen über mehrere Zeilen und Spalten. Dieser Modus ist besonders nützlich für tabellarische Daten, Spaltenoperationen, Alignment-Aufgaben oder das gleichzeitige Einfügen/Löschen in mehreren Zeilen.

7.3.1. Aktivieren des Blockmodus

Der Blockmodus wird mit `Ctrl`-`v` aktiviert.

Ab diesem Moment spannt VIM ein rechteckiges Auswahlgebiet zwischen der ursprünglichen Cursorposition und der aktuellen Cursorposition auf. Jede weitere Bewegung verschiebt eine Ecke des Rechtecks; die andere bleibt an der Startposition verankert.

7.3.2. Erweitern der Auswahl

Die Blockauswahl wird mit den normalen Bewegungsbefehlen verändert:

- **h**, **l** Block horizontal erweitern oder verkleinern
- **j**, **k** Block vertikal über mehrere Zeilen ziehen
- **w**, **b**, **e** wortweise Blockerweiterung
- **0**, **$** Block bis zum Zeilenanfang bzw. Zeilenende ziehen
- **f** **X**, **t** **X** bis/zum nächsten Auftreten von **X** erweitern

So lassen sich präzise rechteckige Markierungen in jede Richtung aufspannen – unabhängig von Einrückungen oder Zeilenlängen.

Übungen

- Lege eine kleine Tabelle an, z. B.:

  ```
  foo 10
  bar 20
  baz 30
  ```

  Setze den Cursor auf das erste 1 und aktiviere mit **Ctrl**-**v** den Blockmodus. Ziehe den Block mit **j** und **l** so, dass alle Zahlen markiert sind, und ersetze sie mit **c** 0 **Esc**. Prüfe, ob alle Werte auf 0 gesetzt wurden.
- Erzeuge mehrere Zeilen mit Kommentarzeichen, z. B.:

  ```
  // TODO: eins
  // TODO: zwei
  // TODO: drei
  ```

  Markiere die führenden `//` mit **Ctrl**-**v**, **j**, **j** und lösche sie mit **d**. Danach stelle den ursprünglichen Zustand mit **u** wieder her und füge stattdessen vor diesen Zeilen ein anderes Präfix ein, indem du erneut blockweise auswählst und **I** verwendest.
- Nimm einen Textblock mit ungleich langen Bezeichnern in der ersten Spalte. Nutze **Ctrl**-**v**, **j**/**k** und **$**, um eine Spalte zum Ausrichten von

Gleichheitszeichen oder Pfeilen zu markieren, und rücke sie mit [>] bzw. [<] gemeinsam ein bzw. aus.

7.3.3. Operatoren im Blockmodus

Blockselektionen verhalten sich anders als normale Visual-Selektionen. Viele Operatoren wirken *spaltenbezogen* auf alle Zeilen des Blocks:

- [d] rechteckigen Bereich löschen
- [c] rechteckigen Bereich ändern
- [y] Block rechteckig kopieren
- [r][X] alle markierten Zeichen durch [X] ersetzen
- [>] / [<] Block spaltenweise ein- bzw. ausrücken

Im Unterschied zu zeilenweisen Operationen bleibt die Spaltenlage der nicht markierten Teile jeder Zeile unverändert; nur der ausgewählte rechteckige Bereich wird verändert.

Übungen

- Lege einen Block an wie:

  ```
  user1: active
  user2: active
  user3: active
  ```

  Markiere mit [Ctrl]-[v] den Spaltenbereich `active` über alle drei Zeilen und ersetze ihn mit [c]`disabled`[Esc]. Prüfe, dass nur der rechteckige Bereich geändert wurde.
- Erzeuge eine einfache ASCII-Tabelle:

  ```
  foo = 1
  bar = 2
  baz = 3
  ```

  Markiere mit [Ctrl]-[v] die Spalte der Ziffern und ersetze sie mit [r][0]. Beobachte, wie [r][0] in allen markierten Zeilen gleichzeitig wirkt.
- Erzeuge mehrere eingerückte Codezeilen, markiere mit [Ctrl]-[v] eine Spalte am Zeilenanfang und rücke sie mit [>] und anschließend [<] wieder aus. Achte darauf, dass nur der Block eingerückt wird, nicht der übrige Text.

7.3.4. Einfügen in mehreren Zeilen

Eine Besonderheit des Blockmodus:

- **I** Text am Anfang jeder markierten Zeile einfügen
- **A** Text am Ende jeder markierten Zeile einfügen

Ablauf:

1. **Ctrl**-**v** Block markieren
2. **I** oder **A** drücken
3. gewünschten Text schreiben
4. mit **Esc** abschließen

VIM wendet die Änderung anschließend automatisch auf alle Zeilen im Block an – die Eingabe erfolgt nur einmal, wird aber auf jede Zeile des Blocks repliziert.

Übungen

- Lege folgenden Text an:

  ```
  host1.example.com
  host2.example.com
  host3.example.com
  ```

  Markiere mit **Ctrl**-**v** die erste Spalte der drei Zeilen, drücke **I** und füge `ssh` ein, danach **Esc**. Prüfe, dass alle Zeilen nun mit `ssh` beginnen.
- Erzeuge eine einfache Liste:

  ```
  item1
  item2
  item3
  ```

  Markiere mit **Ctrl**-**v** die Spalte nach dem letzten Zeichen der Einträge, drücke **A** und füge `, enabled=true` ein, dann **Esc**. Kontrolliere, dass alle Zeilen das Suffix erhalten haben.
- Nimm mehrere Kommentarzeilen, z. B.:

  ```
  // todo 1
  // todo 2
  // todo 3
  ```

  Markiere mit **Ctrl**-**v** die ersten beiden Spalten `//`, drücke **d**, um die Kommentarzeichen in allen Zeilen gleichzeitig zu entfernen. Wiederhole die

Übung in umgekehrter Richtung, indem du Ctrl - v eine Spalte am Anfang markierst und mit I // Esc Kommentare hinzufügst.

7.3.5. Typische Anwendungsfälle

- gleichzeitig in mehreren Zeilen Präfixe oder Suffixe einfügen
- Spalten in Konfigurationsdateien oder Tabellen bearbeiten
- mehrere Kommentarzeichen synchron entfernen oder hinzufügen
- Codeblöcke spaltenweise ausrichten
- JSON- oder CSV-Strukturen spaltenweise verändern

7.3.6. Wiederholbarkeit und Makrohinweise

Der Blockmodus ist nur eingeschränkt mit . wiederholbar. VIM wiederholt zwar die *Operation*, aber nicht die Blockauswahl – der rechteckige Bereich muss also jedes Mal neu markiert werden.

Für Makros gilt daher:

- Blockmodus nur verwenden, wenn es wirklich keinen strukturellen Ersatz gibt
- feste Spaltenpositionen nach Möglichkeit vermeiden
- Textobjekte (i w, i p, i (...) sind in der Regel stabiler als Blockoperationen
- wenn Blockmodus im Makro unvermeidlich ist: so kleine Blöcke wie möglich wählen und vorher mit robusten Bewegungen (0, $, w, / pattern Enter) positionieren

7.3.7. Zusammenfassung

- Ctrl - v aktiviert die blockweise Auswahl
- rechteckige Bereiche können quer über Spalten und Zeilen markiert werden
- besonders geeignet für Tabellen, Spaltenoperationen und Mehrzeilen-Einfügungen
- I und A gehören zu den wichtigsten Befehlen im Blockmodus
- für Makros nur begrenzt empfehlenswert – wo möglich durch Textobjekte und zeilenweise Operationen ersetzen

7.4. Operatoren im Visual Mode

Im Visual Mode wird zuerst eine Auswahl markiert und *danach* ein Operator ausgeführt. Die Reihenfolge ist immer:

Reihenfolge: zuerst Auswahl ⇒ dann Operator. Der Visual Mode kennt drei Selektionsarten:

- **v** zeichenweise Auswahl
- **V** zeilenweise Auswahl
- **Ctrl**-**v** blockweise (rechteckige) Auswahl

Sobald die Auswahl steht, wird ein Operator ausgeführt. Zu den wichtigsten Operatoren im Visual Mode gehören:

- **d** markierten Bereich löschen
- **c** Bereich ändern (löschen und in den Insert Mode wechseln)
- **y** Bereich kopieren (yanken)
- **g** **q** Text neu formatieren (Reflow)
- **=** automatisch einrücken (Auto-Indent)
- **>** einrücken
- **<** Einrückung verringern

Navigation innerhalb der bestehenden Auswahl ist weiterhin möglich, ohne dass der Visual Mode verlassen wird:

- **o** Anfang und Ende der Auswahl vertauschen
- **h**, **j**, **k**, **l** Auswahl vergrößern oder verkleinern
- **w**, **b**, **e** wortweise bewegen
- **f**X, **t**X, **F**X, **T**X bis/zum Zeichen X springen

Besonderheit bei blockweiser Auswahl (Ctrl-v)

- Operatoren wirken spaltenweise auf alle markierten Zeilen
- Einfügen mit **I** schreibt Text an den Anfang jeder Zeile des Blocks
- Einfügen mit **A** schreibt Text an das Ende jeder Zeile des Blocks
- **r**X ersetzt in allen markierten Spalten das Zeichen durch X

7.4.1. Wiederholen und Makros

Der Punkt-Operator [.] wiederholt im Visual-Kontext in der Regel nur die *Operation*, nicht die exakte Auswahl. Für Makros bedeutet das:

- Visual-Selektoren sind nur eingeschränkt reproduzierbar
- strukturelle Bewegungen ([w], [f]X, Textobjekte wie [i][w], [i][p]) sind stabiler
- blockweise Operationen sollten im Makro mit Vorsicht eingesetzt werden

Übungen

- Markiere eine Zeile mit [V] und rücke sie mit [>] ein. Wiederhole die Operation mit [.] auf den folgenden Zeilen und beobachte, was reproduziert wird (Einrückung ja, Auswahl nein).
- Markiere einen Textbereich mit [v] und formatiere ihn mit [g][q]. Versuche anschließend, mit [.] auf einem anderen Absatz denselben Reflow-Effekt zu erzielen.
- Erzeuge einen Block mit [Ctrl]-[v] über mehrere Zeilen und ersetze die markierte Spalte mit [r]#. Wiederhole [.] an einer anderen Stelle und prüfe, wie sich das Verhalten von Auswahl und Operation unterscheidet.

7.4.2. Zusammenfassung

- Visual Mode: zuerst Auswahl, dann Operator
- drei Modi: [v], [V], [Ctrl]-[v]
- Operatoren wirken auf den *gesamten* markierten Bereich
- Blockmodus bringt spaltenweise Speziallogik mit [I], [A], [r]X
- für Makros sind Bewegungen und Textobjekte robuster als Visual-Auswahlen

8. Bereiche und Ex-Befehle

8.1. Range-Syntax

Viele Ex-Befehle in VIM operieren nicht nur auf der aktuellen Zeile, sondern auf frei definierbaren Bereichen. Diese Bereiche heißen *Ranges*. Eine Range besteht aus einer oder zwei Adressen, optional kombiniert mit einem Bereichsoperator.

Adressen können z. B. konkrete Zeilennummern, Suchmuster oder Marken sein. Die Details dazu werden in den folgenden Abschnitten Stück für Stück aufgebaut – hier geht es zunächst um den grundsätzlichen Aufbau.

8.1.1. Grundaufbau

Eine klassische Range folgt diesem Muster:

```
:[Adresse][,[Adresse]] Befehl
```

- Die Doppelpunkteingabe `:...` erfolgt immer im *Normal Mode*.
- Fehlt die erste Adresse, bezieht sich der Befehl auf die aktuelle Zeile.
- Fehlt die zweite Adresse, wird nur die erste Adresse verwendet.

Beispiele:

- `:5d` – löscht Zeile 5.
- `:3,9y` – yankt (kopiert) Zeilen 3 bis 9 in den Standardregisterpuffer.
- `:1,\$s/foo/bar/g` – ersetzt im Bereich Zeile 1 bis Dateiende alle Vorkommen von `foo` durch `bar`.
- `:\%s/foo/bar/g` – Kurzform für `1,$s/foo/bar/g`, d. h. „ganze Datei“ als Range.

Wichtig ist, dass du die Trennung zwischen *Range* und *Befehl* verstanden hast: `:1,\$` ist nur die Adressangabe, `:s/foo/bar/g` ist der eigentliche Befehl. Beides zusammen ergibt eine vollständige Ex-Kommandokette.

Übungen

1. Öffne eine Datei mit mindestens zehn Zeilen Text. Lösche Zeile 5 mit `:5d` und mach die Änderung mit **u** rückgängig. Wiederhole den Vorgang ein paar Mal, bis sich `:5d` „im Muskelgedächtnis“ verankert hat.
2. Kopiere die Zeilen 3 bis 9 mit `:3,9y` und füge sie unterhalb von Zeile 15 wieder ein:
   a) Springe zu Zeile 15, z. B. mit `:15` oder `:15G`.
   b) Füge den kopierten Block mit **p** unterhalb ein.

   Spiele mit **u** und **Ctrl**-**r**, um das Einfügen rückgängig zu machen und wiederherzustellen.
3. Ersetze in der gesamten Datei alle Vorkommen von `foo` durch `bar` mit `:\%s/foo/bar/g`. Prüfe anschließend mit **u** und **Ctrl**-**r**, ob wirklich alle Stellen erwischt wurden und ob du die Änderung sicher vor- und zurückspulen kannst.

8.1.2. Wichtige Adressen

Die folgenden Adressen werden am häufigsten verwendet und bilden das Fundament der Range-Syntax. Sie können einzeln als Zieladresse verwendet oder in Ranges kombiniert werden.

`.` aktuelle Zeile (Cursorposition), also die Zeile, in der du dich beim Eingeben des Ex-Befehls befindest. Beispiel: `:.p` gibt die aktuelle Zeile noch einmal aus.

`$` letzte Zeile der Datei. Beispiel: `:\$d` löscht die letzte Zeile.

`%` gesamte Datei (Kurzform für `1,$`). Typischer Anwendungsfall: `:\%s/alt/neu/g` für eine globale Ersetzung über alle Zeilen.

`'x` Zeile der Marke *x*. Setzen der Marke im Normal Mode mit **m** **x**, anspringen mit **'** **x** oder als Adresse in Ranges, z. B. `:'a,'bd` zum Löschen eines markierten Blocks.

`/Pattern/` nächste Zeile, deren Inhalt auf `Pattern` matcht (Suche *vorwärts* ab der aktuellen Cursorposition). Beispiel: `:/TODO/d` löscht die erste Zeile mit TODO.

`?Pattern?` vorherige Zeile, deren Inhalt auf `Pattern` matcht (Suche *rückwärts*). Beispiel: `:?Fehler?y` kopiert die vorherige Zeile, die `Fehler` enthält.

Diese Adressen lassen sich in Ranges kombinieren, etwa `:.,\$`, `:'a,'b` oder `:/BEGIN/,/END/`. In den nächsten Abschnitten kommt dann die „Feinsteuerung“ über relative Offsets dazu.

8.1.3. Relative Adressen

Adressen lassen sich relativ verschieben, indem man `+n` oder `-n` anhängt (`n` = Anzahl der Zeilen). Das funktioniert sowohl mit einfachen Adressen als auch mit Suchmustern oder Marken.

- `:.+1` – eine Zeile unterhalb der aktuellen Zeile.
- `:.-2` – zwei Zeilen oberhalb der aktuellen Zeile.
- `:\$-1` – vorletzte Zeile der Datei.
- `:/foo/+1` – die Zeile direkt nach der nächsten Zeile, die `foo` enthält.
- `:'a+3` – dritte Zeile *unterhalb* der Marke `a`.

Die eigentliche Stärke zeigt sich in Kombination mit Operatoren oder Ex-Befehlen:

- `:.,.+20d` – löscht von der aktuellen Zeile bis 20 Zeilen tiefer.
- `:/BEGIN/+1,/END/-1y` – yankt alles *zwischen* den Zeilen mit `BEGIN` und `END`, ohne die Markerzeilen.

Übungen

1. Setze in einer längeren Datei zwei Marken:
   - Springe an den Anfang des interessanten Blocks und setze mit **m** **a** die Marke `a`.
   - Springe an das Ende des Blocks und setze mit **m** **b** die Marke `b`.

   Lösche anschließend den Bereich mit `:'a,'b d` und mache die Änderung mit **u** wieder rückgängig.
2. Navigiere mit relativen Adressen:
   - Gib `:.+5p` ein und beobachte, welche Zeile ausgegeben wird.
   - Probiere `:\$-3p` und kontrolliere, ob es wirklich die drittletzte Zeile ist.
3. Erzeuge einen Block, der durch Marken und relative Offsets definiert ist:
   a) Markiere den Blockanfang mit **m** **a**.
   b) Suche weiter unten nach einem Stichwort, z. B. `:/ERROR`.
   c) Lösche von Marke `a` bis zwei Zeilen über dem Treffer mit `:'a,/ERROR/-2 d`.

Spiele mit u und Ctrl-r, bis dir klar ist, wie sich die Offsets `+n`/`-n` auf das Ergebnis auswirken.

8.1.4. Adress-Kombinationen

Adressen werden in der Praxis fast immer in Kombination verwendet. Ein Range besteht dann aus zwei Adressen, getrennt durch ein Komma:

```
:[Adresse],[Adresse] Befehl
```

Typische Beispiele:

- `:.,\$y` yankt alle Zeilen von der aktuellen Zeile (`.`) bis zum Dateiende (`$`). Praktisch, um „ab hier bis unten“ in ein Register zu kopieren.
- `:/BEGIN/,/END/d` löscht den gesamten Block von der nächsten Zeile, die `BEGIN` enthält, bis zur nächsten Zeile mit `END` – inklusive der `BEGIN`- und `END`-Zeilen selbst.
- `:'a,'by` yankt alle Zeilen von Marke `a` bis Marke `b`. (Marken zuvor im Normal Mode mit m a bzw. m b setzen.)
- `:1,10s/foo/bar/g` ersetzt `foo` nur in den Zeilen 1 bis 10 durch `bar`.
- `:.,.+20d` löscht die aktuelle Zeile und die folgenden 20 Zeilen (insgesamt 21 Zeilen) – eine typische „ab hier den Block weg“-Operation.

Wichtig: Beide Adressen sind vollwertige Range-Adressen – alle in den vorherigen Abschnitten beschriebenen Formen (. , $, / `Pattern`, ' a , relative Offsets usw.) lassen sich hier frei kombinieren.

8.1.5. Komma vs. Semikolon

Wenn zwei Adressen kombiniert werden, spielt es eine Rolle, ob sie mit Komma `,` oder mit Semikolon `;` verbunden sind:

- **Komma** `,` beide Adressen werden relativ zur *ursprünglichen* Cursorposition ausgewertet.
- **Semikolon** `;` VIM wertet zuerst die *erste* Adresse aus, setzt den Cursor dorthin und interpretiert die *zweite* Adresse dann ausgehend von dieser neuen Position.

Beispiel:

```
:1,/foo/d
```

- springt logischerweise zur Zeile 1 und
- löscht von Zeile 1 bis zur nächsten Zeile, die `foo` enthält.

Mit Semikolon sieht das formal ähnlich aus, verhält sich aber anders:

```
:1;/foo/d
```

- die erste Adresse `1` setzt den Cursor zunächst in Zeile 1,
- das Semikolon `;` bewirkt, dass die nächste Adresse von dieser neuen Position aus ausgewertet wird,
- die zweite Adresse `/foo/` sucht von dort aus die nächste Zeile mit `foo`,
- der Operator `d` löscht den Bereich von Zeile 1 bis zu dieser Trefferzeile.

Der Unterschied wird in Kombination mit relativen Adressen deutlich:

- `:.,/foo/d`
  die zweite Adresse `/foo/` wird von der ursprünglichen Cursorposition aus interpretiert.
- `:.,;/foo/d`
  ist (wenn `.` schon die aktuelle Position ist) meist identisch – aber in komplexeren Konstrukten mit Marken oder `$` kann das Semikolon dafür sorgen, dass die zweite Adresse von einer „neuen“ Position aus gerechnet wird.

Man kann sich merken:

Komma = beide Adressen von der alten Position,
Semikolon = zweite Adresse von der neuen Position.

Übungen

1. Erzeuge eine Datei mit mehreren BEGIN/END-Blöcken. Probiere:
   - `:/BEGIN/,/END/d`
   - `:/BEGIN/;/END/d`

   Beobachte, wie sich das Verhalten ändert, je nachdem, ob der Cursor vor oder hinter dem ersten BEGIN steht.
2. Setze zwei Marken `a` und `b` in einer Datei:
   - **m** **a** am Blockanfang (a),
   - **m** **b** am Blockende (b).

   Lösche den Block zunächst mit `:'a,'bd` (von `a` bis `b`) und stelle ihn mit **u** wieder her. Variiere dann mit relativen Offsets, z. B. `:'a+1,'b-1d`, und überprüfe das Ergebnis visuell.

3. Stelle den Cursor in die Mitte einer Datei und experimentiere mit:
   - `:.,\$y`
   - `:.,/foo/y`
   - `:.,;/foo/y`

   Notiere dir, welche Zeilen tatsächlich in das Register übernommen werden und wie sich das Semikolon auf die zweite Adresse auswirkt.

8.1.6. Sonderadresse %

Die Sonderadresse % ist eine Abkürzung für den gesamten Puffer:

- % alleine entspricht der Range `1,$`, also „von der ersten bis zur letzten Zeile der Datei".
- `:\%s/alt/neu/g` führt eine globale Ersetzung in der *gesamten* Datei aus.
- `:\%y` yankt alle Zeilen der Datei in das Standardregister.
- `:\%g/TODO/d` löscht alle Zeilen, die TODO enthalten, im gesamten Puffer.
- `:\%normal! I\#` fügt in jeder Zeile am Zeilenanfang ein # ein – praktisch zum schnellen Kommentieren.

Man kann sich merken: % bedeutet immer „die ganze Datei", egal ob mit `:s`, `:g`, `:d`, `:y` oder `:normal` kombiniert.

Übungen

1. Erstelle eine Datei mit gemischten Zeilen, z. B. mehrere Vorkommen von `foo` und `bar`. Probiere nacheinander:
   - `:\%s/foo/bar/g`
   - `:\%s/bar/BAZ/g`

   Kontrolliere das Ergebnis jeweils mit `:\%s/.../.../gn`, um nur die Anzahl der Treffer anzeigen zu lassen.
2. Füge an jede Zeile ein Semikolon am Ende an, indem du `:\%normal! A;` ausführst. Mache die Änderung anschließend mit **u** rückgängig.
3. Erzeuge eine Datei mit Kommentarzeilen, z. B. beginnend mit #. Entferne alle Kommentarzeilen mit `:\%g/\^\#/d` und stelle sie danach mit **u** wieder her.

8.1.7. Range ohne Befehl

Eine Range ohne nachfolgenden Befehl dient als „Vorschau“: VIM zeigt die betreffenden Zeilen im Command-Line-Fenster an, ohne etwas zu ändern.

```
:3,9
```

gibt die Zeilen 3 bis 9 im Command-Line-Fenster aus. Das ist funktional identisch zu:

```
:3,9p
```

Nur dass das `:p` hier implizit ist.
Typische Anwendungsfälle:

- Einen Bereich vor einer Operation ansehen: Erst `:10,20`, dann z. B. `:10,20d`, wenn der Bereich passt.
- Prüfen, ob eine Adresskombination das trifft, was man erwartet (z. B. `:.,.+10`, `:/BEGIN/,/END/`).
- Schnell einen Ausschnitt aus einer großen Datei anzeigen, ohne scrollen zu müssen.

Übungen

1. Öffne eine größere Datei (mindestens 50 Zeilen) und probiere:
   - `:1,10`
   - `:20,30`
   - `:\$-9,\$`

   Beobachte jeweils, welche Zeilen angezeigt werden und ob die Range zu deiner Erwartung passt.
2. Markiere einen logischen Block in der Datei (z. B. eine Funktionsdefinition) und finde seine Zeilennummern über `:set number`. Gib anschließend genau diesen Block mit einer Range ohne Befehl aus (z. B. `:42,71`) und vergleiche die Ausgabe mit dem sichtbaren Bildschirmbereich.
3. Nutze eine Muster-Range zur Kontrolle:
   - `:/BEGIN/,/END/`

   Wenn der Block stimmt, wiederhole denselben Range-Befehl, diesmal „scharf“ mit Operator: `:/BEGIN/,/END/d`. Ziel: Range erst *sichtbar machen*, dann erst mit einem Befehl kombinieren, der wirklich etwas verändert.

8.1.8. Fehlertoleranz

Ranges sind bewusst streng: VIM versucht nicht zu „raten“, was gemeint sein könnte, sondern bricht mit einer klaren Meldung ab. Das schützt dich davor, versehentlich falsche Bereiche zu verändern.

- Ungültige oder leere Ranges führen zu einer klaren Fehlermeldung (z. B. `E16: Invalid range`). Beispiele: `:10,5d` in einer Datei, in der Zeile `10` vor 5 liegt, oder `:200,300d` in einer Datei mit nur 100 Zeilen.
- Sucht ein Muster nichts, wird der zugehörige Ex-Befehl nicht ausgeführt und VIM meldet den Fehler (`E486: Pattern not found`). Beispiel: `:/XYZ/`, wenn XYZ im Puffer nicht vorkommt.
- Gültige Kombinationen wie `:.,.+20d` sind erlaubt: VIM wertet zuerst die Adressen aus und führt danach den Befehl auf dem so entstandenen Bereich aus. Wichtig ist: Erst wird die Range berechnet, *dann* wird der Operator angewendet – nicht Zeile für Zeile „live“.

Übungen

1. Öffne eine Datei mit etwa 30 Zeilen und probiere nacheinander:
   - `:10,5p`
   - `:40,50p`

   Beobachte die Fehlermeldungen (`E16`) und überlege, warum VIM die Ranges ablehnt.
2. Erzeuge eine Testdatei mit einigen Zeilen Text, in denen `foo` vorkommt, aber XYZ *nicht*. Führe aus:
   - `:g/foo/p`
   - `:g/XYZ/p`

   Achte auf den Unterschied: einmal Ausgabe, einmal `Pattern not found` (`E486`).
3. Stelle sicher, dass dein Cursor *nicht* in der ersten Zeile steht, und probiere:
   - `:.,.+5p`
   - `:.,.-1p`

   Interpretiere die Ausgabe: Welche Zeilen hat VIM tatsächlich angezeigt? Wiederhole anschließend mit einem bewusst zu großen Range wie `:.,.+999p` und beobachte die Fehlermeldung.

8.2. Globale Ex-Kommandos (:g)

Der Befehl :g gehört zu den mächtigsten Ex-Kommandos in VIM. Er ermöglicht es, einen Ex-Befehl auf allen Zeilen auszuführen, deren Inhalt zu einem bestimmten Muster passt. Anders als bei *Ranges* über Zeilennummern wählst du hier Zeilen über Suchmuster aus.

8.2.1. Grundform

Eine typische globale Anweisung hat diese Form:

```
:g/<Muster>/ <Ex-Befehl>
```

VIM sucht dabei alle Zeilen, die auf <Muster> matchen, und führt für jede gefundene Zeile den angegebenen Ex-Befehl aus (z. B. :p zum Anzeigen oder :d zum Löschen).

8.2.2. Einfache Beispiele

- :g/foo/p zeigt alle Zeilen, die foo enthalten.
- :g/foo/d löscht alle Zeilen, die foo enthalten.
- :g/\^\#/d löscht alle Zeilen, die mit # beginnen (typische Kommentarzeilen in vielen Konfigurationsdateien).
- :g/\^\$/d löscht Leerzeilen, deren Inhalt genau aus einem Zeilenende besteht.

Wichtig ist die Lücke zwischen Muster und Befehl: :g/foo/p ist gültig, :g/foo p *nicht* – dort würde VIM versuchen, foo p als Teil des Musters zu interpretieren.

Übungen

1. Erzeuge eine Datei mit gemischten Zeilen, z. B. einige mit foo, andere ohne:
   - Zeige alle Zeilen mit :g/foo/p.
   - Lösche alle Zeilen mit :g/foo/d und mache die Änderung mit **u** rückgängig.
2. Erzeuge eine kleine Konfigurationsdatei mit Kommentarzeilen, die mit # beginnen:
   - Entferne alle Kommentarzeilen mit :g/\^\#/d.

    - Stelle sie anschließend mit **u** wieder her.
3. Baue eine Datei mit Leerzeilen zwischen Absätzen:
    - Entferne alle Leerzeilen mit `:g/\^\$/d`.
    - Füge neue Leerzeilen gezielt mit **o** im Normal Mode ein.

8.2.3. Negativauswahl

Neben `:g` gibt es die Negativvariante `:v` (von „*inverse global*“). Statt „alle Zeilen, die zum Muster passen“ wählt `:v` alle Zeilen, die *nicht* zum Muster passen.

- `:v/foo/p` zeigt alle Zeilen, die `foo` *nicht* enthalten.
- `:v/\^\$/d` löscht alle Zeilen, die *nicht* leer sind (denn nur leere Zeilen matchen `$^` – `:v` dreht die Auswahl um).

Man kann sich merken:

`:g` = „alle Zeilen, die passen“
`:v` = „alle Zeilen, die *nicht* passen“

Übungen

1. Erzeuge eine Datei mit gemischten Zeilen, z. B. einige mit `foo` und andere ohne:
    - Zeige alle Zeilen *ohne* `foo` mit `:v/foo/p`.
    - Zeige alle Zeilen *mit* `foo` anschließend zum Vergleich mit `:g/foo/p`.
2. Erzeuge eine Datei mit mehreren nicht-leeren Zeilen und einigen wirklich leeren Zeilen:
    - Lösche alle *nicht-leeren* Zeilen mit `:v/\^\$/d`.
    - Mache die Änderung mit **u** wieder rückgängig.
3. Kombiniere beide Varianten:
    - Zeige erst alle Zeilen mit `:g/bar/p`.
    - Zeige dann alle Zeilen ohne `bar` mit `:v/bar/p`.

    Ziel: Das Zusammenspiel von `:g` und `:v` bewusst verstehen.

8.2.4. Kombination mit Bereichen

Globale Befehle `:g` und `:v` lassen sich mit klassischen Ranges kombinieren. Damit kannst du den Wirkungsbereich zusätzlich *räumlich* einschränken (z. B. auf Zeilen 10–50 oder zwischen zwei Marken):

- `:10,50g/error/d` durchsucht nur die Zeilen 10 bis 50 und löscht dort alle Zeilen, die `error` enthalten.
- `:'a,'bg/TODO/p` wirkt nur im Bereich von Marke `a` bis Marke `b` und zeigt alle Zeilen mit TODO in diesem Abschnitt.

Der Ablauf ist immer gleich:

1. VIM schränkt die Zeilenmenge durch die Range ein (z. B. `:10,50` oder `:'a,'b`).
2. Innerhalb dieser Menge werden nur die Zeilen ausgewählt, die zum Muster von `:g` oder `:v` passen (bzw. nicht passen).
3. Auf diese *Schnittmenge* wird der Ex-Befehl angewendet (z. B. `:d`, `:p`, `:s/.../.../`).

Übungen

1. Öffne eine Logdatei mit vielen Zeilen:
   - Zeige alle Fehlerzeilen im gesamten File mit `:g/error/p`.
   - Zeige dann nur Fehlerzeilen im Bereich 50–120 mit `:50,120g/error/p`.

   Beobachte den Unterschied im Umfang der Ausgabe.
2. Setze zwei Marken in einer Datei:
   - Positioniere den Cursor auf den Blockanfang und setze eine Marke mit **m** **a**.
   - Gehe zum Blockende und setze eine Marke mit **m** **b**.
   - Zeige alle TODO-Zeilen nur in diesem Block mit `:'a,'bg/TODO/p`.
3. Kombiniere Range + Negativauswahl:
   - Zeige in den Zeilen 20–60 alle Zeilen, die *kein* DEBUG enthalten, mit `:20,60v/DEBUG/p`.
   - Überlege, wie du dieselbe Auswahl mit `:g` und einem anderen Muster ausdrücken könntest.

8.2.5. Kettenbefehle

Globale Kommandos werden erst richtig stark, wenn du mehrere Ex-Befehle in einer einzigen `:g`-Zeile verkettest. Die einzelnen Befehle werden mit dem Pipe-Symbol **|** getrennt und nacheinander auf jede Trefferzeile angewendet.

```
:g/foo/ s/bar/baz/g | normal! A;
```

Was passiert hier?

- `:g/foo/ \dots` wählt alle Zeilen, die `foo` enthalten.
- `:s/bar/baz/g` ersetzt `bar` durch `baz` (nur innerhalb dieser Zeilen).
- `:normal! A;` führt im Normal Mode auf jeder Trefferzeile den Befehl [A][;] aus:
  - [A] – ans Zeilenende springen und in den Insert-Mode wechseln,
  - [;] – ein Semikolon anhängen.

Übungen

1. Erzeuge eine Datei mit mehreren Zeilen, die DEBUG enthalten. Füge an jede dieser Zeilen am Ende `// debug` an, z. B. mit:
   - `:g/DEBUG/ normal! A // debug`

   Mache die Änderung mit [u] wieder rückgängig.
2. Erzeuge eine Liste von Funktionsköpfen (z. B. in C oder Python) und lasse dir alle Zeilen mit TODO anzeigen, in denen zusätzlich `FIXME` vorkommt:
   - `:g/TODO/ g/FIXME/p`

   Beobachte, wie `:g` verschachtelt arbeitet.
3. Baue eine Kette, die Zeilen mit ERROR zuerst markiert und dann löscht:
   - `:g/ERROR/ normal! I#  :d`

   Überlege dir, warum die Reihenfolge der Befehle in der Kette relevant ist.

8.2.6. Sondermuster

In Kombination mit `:g` sind ein paar RegEx-Muster besonders praxisrelevant:

- `:g/^\s*$/d` löscht alle Zeilen, die nur aus Leerraum bestehen (Leerzeilen inklusive Spaces/Tabs).
- `:g/\<main\>/p` findet nur das Wort `main` als eigenes Wort, nicht `domain` oder `mainloop`.
- `:g!/regex/p` historische Negativform von `:g`, entspricht inhaltlich der Verwendung von `:v/regex/p`. Heute ist `:v` die bevorzugte Schreibweise.

Übungen

1. Erzeuge eine Datei mit „leer“ wirkenden Zeilen, die aber teils Spaces oder Tabs enthalten. Entferne alle optisch leeren Zeilen mit:
   - `:g/^\s*$/d`

Kontrolliere das Ergebnis mit **u** und wiederhole den Befehl, bis du das Muster im Blick hast.

2. Erzeuge eine Datei mit Funktionsnamen wie `main`, `domain`, `maintain`. Lasse dir nur die reine `main`-Definition anzeigen mit:
   - `:g/\<main\>/p`

   Überlege dir, wie du alternativ mit einem „rohen“ Pattern wie `/\<main\>/` im Suchmodus arbeiten würdest.
3. Nimm eine Datei mit gemischten Zeilen (einige passen auf `regex`, andere nicht). Vergleiche:
   - `:g/regex/p`
   - `:v/regex/p`
   - `:g!/regex/p`

   Ziel: Den Unterschied zwischen `:g`, `:v` und der älteren `:g!`-Schreibweise bewusst wahrnehmen.

8.2.7. Fehlerquellen

Auch `:g`-Kommandos haben ein paar typische Fallstricke, die du kennen solltest:

- Richtig: `:g/foo/p`
  Falsch: `:g/foo p`
  Zwischen Muster und Befehl muss immer ein Leerzeichen stehen *nach* dem schließenden Slash. Alles nach dem Muster interpretiert VIM als Ex-Befehl.
- Wenn das Muster selbst `/` enthält, solltest du einen anderen Trenner verwenden, z. B. `@`:
  `:g@/tmp@ s@/tmp@/var/tmp@g`
  So bleibt das Pattern `/tmp` lesbar, ohne dass du alle Slashes escapen musst.
- `:normal` immer mit `:normal!` verwenden, damit keine Remaps greifen und das Verhalten reproduzierbar bleibt:
  `:g/foo/ normal! dd`
  `:normal dd` würde ggf. gemappte **d**-Befehle ausführen und Makros unvorhersehbar machen.

Übungen

1. Erzeuge eine Datei mit Zeilen, die `foo`, `foobar` und `barfoo` enthalten. Probiere:
   - `:g/foo/p`

   - `:g/\^{}foo/p`
   - `:g/foo\$/p`

   Beobachte, welche Zeilen jeweils getroffen werden und wie sich das Pattern auf das Ergebnis auswirkt.
2. Erzeuge eine Datei, in der Pfade wie `/tmp` und `/var/tmp` vorkommen. Versuche zunächst:
   - `:g//tmp/ s//tmp/\/var\/tmp/g`

   und ersetze das Kommando danach durch die Variante mit alternativen Trennern:
   - `:g@/tmp@ s@/tmp@/var/tmp@g`

   Vergleiche, welche Variante du schneller lesen und warten kannst.

8.2.8. Makrotauglichkeit

Globale Kommandos sind besonders interessant, wenn sie in Makros oder Scripts verwendet werden. Einige Eigenschaften machen `:g` dafür geeignet:

- **Keine Cursorabhängigkeit:** `:g`-Kommandos arbeiten immer über den gesamten (oder explizit eingeschränkten) Bereich und sind nicht davon abhängig, wo der Cursor gerade steht.
- **Stabil im Batch-Betrieb:** `:g`-Kommandos können in Makros, Mappings und Start-Skripten verwendet werden, ohne dass der Benutzer jedes Mal interaktiv eingreifen muss.
- **Deterministisches Verhalten:** Bei gleichem Inhalt und gleichem Pattern liefern `:g`-Kommandos immer dasselbe Ergebnis – ideal für wiederholbare Refactorings oder Log-Auswertungen.

Übungen

1. Nimm ein Makro in Register `"q` auf, das alle Zeilen mit TODO löscht, indem es `:g/TODO/d` ausführt. Spiele das Makro mehrfach auf derselben Datei ab und beobachte, dass das Ergebnis immer gleich bleibt, egal wo der Cursor steht.
2. Baue dir ein einfaches Refactoring-Makro, das in einer Logdatei alle Zeilen mit DEBUG entfernt und `INFO` in `WARN` umbenennt. Nutze dazu z. B.:
   - `:g/DEBUG/d`
   - `:\%s/INFO/WARN/g`

Wiederhole das Makro auf verschiedenen Dateien und prüfe, ob es sich überall gleich verhält.

8.3. Filterbefehle (!)

Filterbefehle erlauben es, einen Textbereich durch ein externes Programm zu schicken und dessen Ausgabe wieder in den Puffer zurückzuschreiben. Das ist ideal für Sortierung, Analyse, Formatierung oder statische Checks, ohne VIM selbst verlassen zu müssen.

8.3.1. Grundprinzip

Die allgemeine Form eines Filterbefehls lautet:

```
:[Range]!Kommando
```

Dabei bedeutet:

- `[Range]` – der betroffene Zeilenbereich (z. B. `%`, `1,$`, `10,20`)
- `!` – „schicke diesen Bereich durch ein externes Programm“
- `Kommando` – ein Shell-Kommando, das auf der Range arbeitet

Wichtige Beispiele:

- `:\%!sort`
  sortiert die gesamte Datei lexikographisch.
- `:10,20!sort`
  sortiert nur die Zeilen 10 bis 20, alle anderen Zeilen bleiben unverändert.
- `:\%!uniq`
  fasst direkt aufeinanderfolgende Dubletten zu einer Zeile zusammen (typisch nach einem vorherigen `:\%!sort`).
- `:\%!sed 's/foo/bar/g'`
  ersetzt `foo` durch `bar` im gesamten Puffer, aber diesmal durch das externe Kommando `sed` statt durch VIMs eigenes `:s`-Kommando – praktisch, wenn du bestehende Shell-Snippets nutzen willst.

Wichtig: Das externe Programm sieht nur den Text der Range auf `stdin` und schreibt seine Ausgabe auf `stdout`. VIM ersetzt den Bereich dann komplett durch diese Ausgabe. Wenn das Kommando nichts ausgibt, verschwindet der Bereich.

Übungen

1. Erstelle eine Datei mit unsortierten Wörtern, jeweils eines pro Zeile. Probiere:
   - `:\%!sort`
   - `:\%!sort -r`

   Beobachte, wie sich die Sortierreihenfolge verändert.
2. Erzeuge eine Datei mit vielen doppelten Zeilen, zum Beispiel mehrere identische Log-Einträge hintereinander. Wende nacheinander an:
   - `:\%!sort`
   - `:\%!uniq`

   Ziel: alle Duplikate sollen zu Einzelzeilen zusammenfallen.
3. Schreibe in eine Testdatei mehrere Zeilen mit dem Wort `foo` und führe aus:
   - `:\%!sed 's/foo/bar/g'`

   Vergleiche das Ergebnis mit VIMs eigener Variante:
   - `:\%s/foo/bar/g`

   Beobachte, dass beide Wege funktional dasselbe Ergebnis liefern.

8.3.2. Filter im Visual Mode

Statt eine Range explizit zu tippen, kannst du auch einen Bereich im Visual Mode markieren und ihn mit einem Filter bearbeiten:

1. Markiere die gewünschten Zeilen mit [V] (linewise Visual) und Bewegungen wie [j], [k].
2. Drücke [!]. VIM zeigt unten `::'<,'>` als Range an.
3. Tippe das gewünschte Kommando (z. B. `:sort`) und bestätige mit [Enter].

Beispiele:

- Markiere einen Block und führe [!]`sort`[Enter] aus, um nur diesen Block zu sortieren.
- Markiere eine Spalte in einem Log-Ausschnitt und filtere sie durch ein eigenes Script, z. B. [!]`python3 mein_filter.py`[Enter].

Der Visual-Filter ist oft schneller im Alltag, weil du die Range nicht tippen musst und sofort siehst, was betroffen ist.

Übungen

1. Öffne eine Datei mit mehreren Blöcken (z. B. verschiedene Funktionsdefinitionen). Sortiere nur einen Block, indem du:

   a) den Block mit **V** und Bewegungen markierst,
   b) **!** `sort` **Enter** ausführst.

   Vergleiche den sortierten Block mit den übrigen Teilen der Datei.
2. Markiere im Visual Mode eine Liste mit Zahlen oder Versionen (z. B. `v1.2.3`, `v0.9.0`, `v2.0.0`) und sortiere sie nur innerhalb dieses Bereichs. Überlege, ob du `:sort` mit Optionen (z. B. `-n`) kombinieren möchtest.
3. Erzeuge eine Testdatei, die unschöne Leerzeilen und überflüssige Leerzeichen am Zeilenende enthält. Schreibe ein kleines Shell-Pipeline-Kommando, das:
   - Leerzeilen entfernt und
   - `trailing whitespace` abschneidet.

   Teste es zunächst mit `:\%!...`, und überlege dann, ob du das Kommando in deiner `.vimrc` als Mapping oder Kommando hinterlegen möchtest.

8.3.3. Filter auf Bereichen

Statt immer die gesamte Datei mit `%` zu filtern, kannst du jeden beliebigen Bereich an ein externes Kommando übergeben. Die Syntax bleibt gleich, nur die Range ändert sich:

```
:[Range]!Kommando
```

Typische Beispiele:

- `:10,20!sort`
  sortiert nur die Zeilen 10–20, alle anderen Zeilen bleiben unverändert.
- `:.,.+5!sort`
  sortiert sechs Zeilen ab der aktuellen Cursorzeile (aktuelle Zeile plus fünf weitere).
- `'<,'>! sed 's/^/# /'`
  wendet im Visual-Bereich `sed` an und fügt in jeder markierten Zeile am Anfang ein # als Kommentarpräfix ein.

Merke: Alles, was du als Range verwenden kannst (Zeilennummern, `.`, `$`, Marken, Muster-Ranges wie `:/BEGIN/,/END/`), kannst du auch vor **!** setzen. VIM schickt dann genau diesen Ausschnitt durch das externe Programm und ersetzt den Bereich durch dessen Ausgabe.

Übungen

1. Erzeuge eine Datei mit unsortierten Zeilen und sortiere nacheinander:

   - nur die Zeilen 5–15 mit `:5,15!sort`
   - die gesamte Datei mit `:\%!sort`

   Vergleiche die Unterschiede.
2. Markiere mit [V] einen Block von etwa zehn Zeilen und führe:
   - [!]`sort`[Enter]

   aus. Beobachte, dass nur der markierte Block sortiert wird.
3. Erstelle eine Liste, in der einige Zeilen mit # beginnen. Markiere diese Zeilen im Visual Mode und füge mit:

   ```
   '<,'>! sed 's/^/# /'
   ```

   ein zweites # hinzu (doppelt kommentiert). Stelle den ursprünglichen Zustand mit [u] wieder her.

8.3.4. Typische Anwendungsfälle

Filterbefehle lohnen sich immer dann, wenn VIM-internes `:sort`, `:g`, `:s` oder Makros zu umständlich werden – oder wenn es bereits bewährte Unix-Werkzeuge gibt, die genau das tun, was du brauchst.

Typische Szenarien:

- Alphabetisches Sortieren größerer Listen (z. B. Konfig-Einträge, Funktionslisten, Logzeilen).
- Entfernen doppelter Zeilen in Log- oder Konfigdateien.
- Umformatieren von Code oder Daten mit bestehenden Skripten (z. B. `sed`, `perl`, `jq`).
- Anwenden externer Linter oder Formatter (z. B. `black`, `gofmt`, `shfmt`).
- Textstatistiken und Auswertungen mit `wc`, `awk`, `cut`, `sort` etc.

Beispiele:

```
:\%! awk '\{print NR, \$0\}'
```

Nummeriert jede Zeile durch, indem vorne die Zeilennummer (NR) ausgegeben und dahinter die ursprüngliche Zeile ($0) angehängt wird.

```
:\%! tr '[:upper:]' '[:lower:]'
```

Wandelt alle Buchstaben im Puffer in Kleinbuchstaben um – praktisch für ad-hoc-Normalisierung.

```
:\%! rev
```

Spiegelt jede Zeile Zeichen für Zeichen. Das ist selten fachlich nötig, aber ein gutes Beispiel, um das Prinzip „Bereich → Filter → Ergebnis“ zu verstehen.

Übungen

1. Erzeuge eine Datei mit beliebigem Text und nummeriere alle Zeilen mit:
   - `:\%! awk '\{print NR, \$0\}'`

   Mache die Änderung mit **u** rückgängig und wiederhole den Befehl nur für einen Bereich, z. B. `:10,30! awk '\{print NR, \$0\}'`.
2. Schreibe einige Zeilen mit gemischter Groß-/Kleinschreibung. Normalisiere den Text mit:
   - `:\%! tr '[:upper:]' '[:lower:]'`

   Überlege anschließend, wie du nur einen Visual-Block (**V** oder **Ctrl**-**v**) durch **!** `tr` schickst.
3. Erzeuge eine Liste mit mehrfach vorkommenden Zeilen. Entferne Duplikate mit:
   - `:\%! sort | uniq`

   und beobachte den Unterschied zu einem reinen `:\%! sort`. Stelle den ursprünglichen Zustand jeweils mit **u** wieder her.

8.3.5. Komplexe Workflows

Mehrere Filterkommandos lassen sich wie auf der Shell mit Pipes verketten. Damit kannst du in einem Schritt filtern, sortieren, verdichten oder umformatieren.

Ein typisches Beispiel:

```
:\%!\,grep -v TODO \textbar {} sort \textbar
{} uniq -c
```

Workflow im Detail:

1. `:\%` – wendet den Filter auf die *gesamte Datei* an.
2. `grep -v TODO` – entfernt alle Zeilen, die TODO enthalten.
3. `sort` – sortiert die verbleibenden Zeilen.
4. `uniq -c` – fasst identische Zeilen zusammen und zählt sie.

Das Endergebnis ersetzt den ursprünglichen Inhalt im angegebenen Bereich. Mit **u** kannst du den gesamten Filtervorgang wie jede andere Änderung rückgängig machen.

Übungen

1. Erzeuge eine Datei mit gemischten Statuszeilen, z. B.:

```
INFO:  started worker A
WARN:  low disk space
INFO:  started worker B
ERROR: connection lost
INFO:  started worker C
ERROR: timeout
```

   Probiere nacheinander:
   - `:\%!\,grep ERROR`
   - `:\%!\,grep -v INFO`
   - `:\%!\,grep ERROR \textbar {} sort \textbar {} uniq -c`

   Mache die Änderungen jeweils mit **u** rückgängig und beobachte, wie sich der Buffer verändert.
2. Erzeuge eine Datei mit duplizierten Zeilen (z. B. Log-Auszüge mit mehrfach auftretenden Einträgen). Wende an:
   - `:\%!\,sort`
   - `:\%!\,uniq`
   - `:\%!\,sort \textbar {} uniq -c`

   Ziel: Verstehe, wie sich `sort` und `uniq` ergänzen und warum `uniq -c` nur auf sortierten Daten wirklich aussagekräftig ist.
3. Nutze `:\%!\,awk '\{print NR, \$0\}'` auf einer Datei mit mehreren Zeilen. Beobachte, wie `awk` jede Zeile mit ihrer Zeilennummer versieht und der gesamte Buffer durch die Filterausgabe ersetzt wird.

8.3.6. Wichtige Hinweise

- **Filter laufen über die Shell.** VIM übergibt den ausgewählten Bereich an deine Login-Shell ($SHELL) und setzt die Filterausgabe wieder in den Buffer

ein. Unterschiede zwischen `bash`, `zsh` oder `fish` können sich direkt auf das Verhalten deiner Filter auswirken.

- **Bereich bewusst wählen.** `:\%!\,kommando` überschreibt *immer die gesamte Datei.* Wenn du nur einen Ausschnitt filtern willst, nutze Ranges wie `:10,50!\,kommando` oder einen Visual-Bereich mit `:'<,'>!\,kommando`.
- **Undo als Sicherheitsnetz.** Jeder Filtervorgang ist eine normale Änderung am Buffer und lässt sich mit **u** (oder `:undo`) vollständig zurücknehmen. Trotzdem: Komplexe Filter vorher ruhig einmal auf einem kleineren Bereich testen.
- **Portabilität beachten.** Kommandos wie `grep`, `sed` oder `awk` verhalten sich je nach System leicht unterschiedlich (GNU vs. BSD). Für wiederverwendbare Workflows möglichst einfache, POSIX-nahe Varianten wählen.
- **Filter nicht in „flüchtige“ Register packen.** Lange Filterzeilen oder Filterketten kannst du in einem benannten Register (z. B. `"q`) ablegen und mit `:@q` ausführen. Vermeide numerische Register (`0--9`), da diese von VIM für Yanks und Löschungen automatisch überschrieben werden.

8.4. Ex-Befehlsüberblick

Die Ex-Befehle bilden die Kommandoebene von VIM und ermöglichen Operationen, die über den Normal-Mode hinausgehen: Speichern, Laden, Suchen und Ersetzen, Filtern, Sortieren, Bearbeiten von Bereichen, Arbeiten mit Fenstern und sogar das Ausführen externer Programme.

Viele Ex-Befehle folgen einem gemeinsamen Schema:

```
[Range] :Command [Argumente]
```

- `[Range]` – optionaler Zeilenbereich (z. B. `%`, `1`, `$`, `.`, `.+5`)
- `:Command` – eigentlicher Ex-Befehl (z. B. `w`, `s`, `g`, `!`)
- `[Argumente]` – zusätzliche Parameter (z. B. Dateinamen, Muster, Optionen)

Im Alltag werden viele Ex-Befehle ohne Range benutzt. Sobald du Ranges und globale Kommandos verstanden hast, kannst du dieselben Befehle gezielt auf Ausschnitte des Buffers anwenden.

8.4.1. Dateioperationen

Die grundlegendsten Ex-Befehle steuern das Speichern und Beenden von VIM. Sie sind die „Notausgänge“, die immer funktionieren – unabhängig vom aktuellen Modus.

- `:w` Speichert den aktuellen Buffer in die zuletzt geöffnete Datei.
- `:w datei.txt` Speichert den aktuellen Buffer unter einem neuen Namen `datei.txt` (der Buffer bleibt aber an den ursprünglichen Dateinamen gebunden).
- `:wa` Speichert *alle* geänderten Buffer.
- `:q` Beendet das aktuelle Fenster, scheitert aber, wenn ungespeicherte Änderungen vorhanden sind.
- `:q!` Beendet das aktuelle Fenster und verwirft alle ungesicherten Änderungen (harte Exit-Variante).
- `:wq` Speichert den aktuellen Buffer und beendet anschließend.
- `:x` Wie `:wq`, speichert aber nur, wenn der Buffer tatsächlich geändert wurde.

Typische Tastenkombinationen aus dem Normal Mode (**Z** **Z**, **Z** **Q**) sind nur Abkürzungen für diese Ex-Befehle:

- **Z** **Z** entspricht `:x`
- **Z** **Q** entspricht `:q!`

Merke: Wenn du „feststeckst“, kommst du mit **Esc** in den Normal Mode und anschließend mit einem passenden Ex-Befehl immer wieder sauber aus VIM heraus.

Übungen

1. Öffne eine neue Datei mit `:e test.txt`, schreibe ein paar Zeilen Text im Insert Mode, und probiere anschließend:
   - `:w`
   - `:w andere.txt`
   - `:wa`

   Prüfe im Dateisystem, welche Dateien angelegt bzw. verändert wurden.
2. Öffne eine Datei, ändere eine Zeile und versuche dann:
   - `:q` (VIM sollte wegen ungespeicherter Änderungen meckern)
   - `:q!` (diesmal sollte VIM kommentarlos beenden)

   Wiederhole den Versuch, speichere diesmal vorher mit `:w`, und beobachte den Unterschied.
3. Starte VIM mit einer Testdatei und probiere nacheinander:

- Z Z
- Z Q

Vergleiche das Verhalten mit `:x` bzw. `:q!` und präge dir ein, dass es sich nur um komfortable Normal-Mode-Abkürzungen für die entsprechenden Ex-Befehle handelt.

8.4.2. Navigation im Ex-Modus

Ex-Befehle können nicht nur Bereiche bearbeiten, sondern auch als reine Navigationsbefehle dienen. Das ist besonders praktisch, wenn du eine konkrete Zeilennummer oder ein Suchmuster im Kopf hast.

- `:1` springt zur ersten Zeile der Datei.
- `:\$` springt zur letzten Zeile der Datei.
- `:42` springt zur Zeile `42`.
- `:/Muster/` springt zur nächsten Zeile, deren Inhalt zu `Muster` passt.
- `:?Muster?` springt zur vorherigen Zeile, deren Inhalt zu `Muster` passt.

Im Gegensatz zur Navigation im Normal Mode (g g, G, Ctrl-f, ...) sind diese Befehle explizit und wiederholbar:, ...) sind diese Befehle explizit und wiederholbar: Sie lassen sich in Makros, Skripten und Befehlsfolgen einbauen, ohne vom aktuellen Cursorverlauf abzuhängen.

Übungen

1. Öffne eine Datei mit mindestens 100 Zeilen. Probiere nacheinander:
   - `:1`
   - `:50`
   - `:\$`

   Vergleiche das Verhalten mit den Normal-Mode-Befehlen g g, 5 0 G und G.
2. Suche einen Begriff, der mehrfach vorkommt, z. B. TODO:
   - `:/TODO/`
   - `:/TODO/` (mehrfach wiederholen)
   - `:?TODO?`

   Beobachte, wie du mit Ex-Navigation gezielt zur nächsten bzw. vorherigen Fundstelle springen kannst.
3. Kombiniere Ex-Navigation mit einer Range:

- Springe mit `:42` zu einer Zeile deiner Wahl.
- Lösche von dort bis zum Dateiende mit `:.,\$d`.

Mache die Änderung mit **u** wieder rückgängig und versuche anschließend, denselben Effekt nur mit Normal-Mode-Befehlen nachzubauen.

8.4.3. Bearbeitung durch Bereiche

Viele Ex-Befehle kombinieren eine Range mit einem eigentlichen Kommando. Damit kannst du gezielt nur Teile der Datei verändern, statt immer den gesamten Puffer zu bearbeiten.

Typische Muster:

- `:g/<Muster>/d` löscht alle Zeilen, deren Inhalt zu `<Muster>` passt.
- `:v/<Muster>/d` löscht alle Zeilen, deren Inhalt *nicht* zu `<Muster>` passt.
- `:442,480s/foo/bar/g` ersetzt `foo` durch `bar` nur in den Zeilen 442 bis 480.
- `:'<,'>s/.../.../` führt eine Ersetzung nur im aktuell markierten Visual-Bereich aus.

Damit hast du drei Freiheitsgrade:

1. *Wo?* – über die Range (`:\%`, `:10,20`, `:'<,'>` usw.)
2. *Was?* – über das Suchmuster (`<Muster>`)
3. *Wie?* – über den eigentlichen Ex-Befehl (`:s`, `:d`, `:y`, `:normal!` ...)

Übungen

1. Erzeuge eine Datei mit gemischten Zeilen, z. B. Log-Ausgaben mit unterschiedlichen Leveln (`INFO`, `WARN`, `ERROR`). Entferne nacheinander:
   - alle `INFO`-Zeilen mit `:g/INFO/d`
   - alle *nicht*-ERROR-Zeilen mit `:v/ERROR/d`

   Mache die Änderungen jeweils mit **u** rückgängig.
2. Nimm eine Datei mit mindestens 100 Zeilen und markiere im Visual Line Mode (**V**) einen Block von etwa 10 Zeilen. Führe dann aus:
   - `:'<,'>s/foo/bar/g`

   Beobachte, dass nur der markierte Bereich verändert wird.
3. Wähle einen numerischen Bereich, z. B. Zeilen 50–80, und probiere:
   - `:50,80s/alt/neu/g`
   - `:50,80g/TODO/d`

Vergleiche den Effekt mit einer globalen Variante über `:\%`.

8.4.4. Filter und externe Programme

Ex-Befehle können direkt mit externen Programmen der Shell kombiniert werden. Damit wird VIM zu einem Frontend für klassische Unix-Tools wie `sort`, `uniq`, `fmt` oder eigene Skripte.

Typische Formen:

- `:r !ls` liest die Ausgabe von `ls` ein und fügt sie *unterhalb* der aktuellen Zeile in den Puffer ein.
- `:\%!sort` schickt die *gesamte* Datei durch `sort` und ersetzt den Inhalt des Puffers durch die sortierte Ausgabe.
- `:'<,'>!fmt` filtert nur den aktuell im Visual Line Mode markierten Bereich durch `fmt` und ersetzt diesen Block durch den neu formatierten Text.

Wichtig: In allen Fällen gilt das Schema *„Range ! externes Kommando“*. VIM übergibt den Bereich an das Programm auf `stdin` und ersetzt den Bereich anschließend durch `stdout`. Ein **u** macht auch diese Änderungen komplett rückgängig.

Übungen

1. Erzeuge eine Datei mit unsortierten Zeilen (z. B. eine Liste von Dateinamen oder Wörtern). Sortiere zunächst nur den Bereich Zeilen 10–30 mit `:10,30!sort` und danach die *ganze* Datei mit `:\%!sort`. Beobachte den Unterschied.
2. Wähle im Visual Line Mode (**V**) einen Absatz aus und formatiere ihn mit `:'<,'>!fmt`. Probiere verschiedene Textbreiten, indem du `fmt` mit Optionen aufrufst (z. B. `:'<,'>!fmt -w 60`).

8.4.5. Fenster und Buffer

VIM trennt streng zwischen *Buffer* und *Fenstern*:

- Ein *Buffer* ist der im Speicher gehaltene Inhalt einer Datei.
- Ein *Fenster* ist nur eine Sicht auf einen Buffer.

Eine Datei kann in mehreren Fenstern sichtbar sein, aber nur einmal als Buffer existieren. Umgekehrt können mehrere Buffer geöffnet sein, ohne dass für alle ein Fenster existiert.

Buffer verwalten

- `:e \emph {datei}` öffnet *datei* im aktuellen Fenster. Existiert bereits ein Buffer mit dieser Datei, wird er wiederverwendet.
- `:bnext` / `:bprev` wechselt zum nächsten / vorherigen Buffer in der internen Bufferliste.
- `:b \emph {nr}` springt direkt zu Buffer *nr*. Die aktuelle Bufferliste bekommst du mit `:ls` oder `:buffers`.

Buffer können „versteckt“ sein: Auch wenn kein Fenster sie gerade anzeigt, bleiben Änderungen im Speicher, solange der Buffer nicht geschlossen oder VIM beendet wird.

Fenster aufteilen

- `:split \emph {datei}` teilt das aktuelle Fenster horizontal und zeigt oben den bisherigen Buffer, unten *datei*.
- `:vsplit \emph {datei}` teilt vertikal: links der aktuelle Buffer, rechts *datei*.
- `:close` schließt das aktuelle Fenster. Der zugehörige Buffer bleibt geöffnet, solange er noch in einem anderen Fenster angezeigt wird oder in der Bufferliste geführt wird.
- `:only` lässt nur das aktuelle Fenster stehen und schließt alle anderen.

Für das Navigieren zwischen Fenstern nutzt du typischerweise **Ctrl**-**w** gefolgt von **h**, **j**, **k** oder **l**:

- **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l** bewegen den Fokus zwischen Fenstern nach links, unten, oben, rechts.
- **Ctrl**-**w** **q** schließt das aktuelle Fenster (wie `:close`).

Übungen

1. Öffne eine Datei und teile das Fenster mit `:vsplit`. Öffne rechts eine zweite Datei mit `:e \emph {datei2}`. Wechsle mit **Ctrl**-**w** **h** und **Ctrl**-**w** **l** zwischen den Fenstern hin und her.
2. Öffne drei verschiedene Dateien nacheinander mit `:e`. Lass dir dann mit `:ls` die Bufferliste anzeigen und wechsle mit `:bnext` und `:bprev` durch die Buffer.

3. Erzeuge zwei horizontale Splits mit `:split`. Schließe das mittlere Fenster mit **Ctrl**-**w** **q** und beobachte, dass der Buffer weiterhin in der Bufferliste vorhanden ist.

8.4.6. Optionen und Status

VIM steuert sein Verhalten über Optionen, die mit `:set` ein- oder ausgeschaltet bzw. auf bestimmte Werte gesetzt werden können. Viele Optionen beeinflussen direkt, wie der Text angezeigt oder interpretiert wird (z. B. Zeilennummern, Tabbreite, Suchverhalten).

Grundbefehle

- `:set` zeigt alle aktuell von ihrem Standard abweichenden Optionen mit ihrem Wert an.
- `:set all` listet *alle* Optionen mit ihren aktuellen Werten.
- `:set number` / `:set nonumber` aktiviert bzw. deaktiviert Zeilennummern in der linken Spalte.
- `:set number?` zeigt den aktuellen Status der Option `number`.
- `:set tabstop=4` setzt die Tabbreite auf 4 Spalten (Beispiel für eine Wert-Option).
- `:set option!` schaltet Bool-Optionen um (z. B. `:set number!`).
- `:set option&` setzt eine Option auf ihren Standardwert zurück (z. B. `:set number\&`).

Viele Optionen gibt es als globale und lokale Variante, etwa `:setlocal number` (nur für das aktuelle Fenster) und `:setglobal number` (Standard für neue Buffer/Fenster).

Status und Hilfe

Das eingebaute Hilfesystem ist eng mit den Optionen verknüpft:

- `:help` öffnet die Startseite des Hilfesystems.
- `:help user-manual` springt ins Benutzerhandbuch (empfohlener Einstieg).
- `:help :set` zeigt die Dokumentation zum Befehl `:set`.
- `:help 'number'` schlägt die Option `number` nach (Optionen immer in einfachen Hochkommas angeben).

- `:help option-list` Übersicht über die wichtigsten Optionen und ihre Kategorien.

In der Statuszeile und im Kommandobereich siehst du direkt, welche Ex-Befehle du eintippst und wie VIM darauf reagiert. Über `:set ruler` und `:set showcmd` lässt sich diese Rückmeldung zusätzlich verbessern.

Übungen

1. Öffne eine Datei und aktiviere Zeilennummern mit `:set number`. Deaktiviere sie anschließend wieder mit `:set nonumber`. Prüfe den Status mit `:set number?`.
2. Spiele mit Suchoptionen:
   - Aktiviere `:set ignorecase` und `:set smartcase`.
   - Suche mit [/] nach einem Wort in verschiedener Groß-/Kleinschreibung.
   - Deaktiviere die Optionen wieder mit `:set noignorecase` und `:set nosmartcase`.
3. Setze `:set tabstop=4` und `:set shiftwidth=4` und beobachte, wie sich Einrückung und Tabs verhalten. Setze beide Optionen danach mit `:set tabstop\&` und `:set shiftwidth\&` auf ihre Standardwerte zurück.
4. Rufe `:help 'number'` und `:help 'expandtab'` auf und lies jeweils die *Beschreibung* und den Abschnitt „Beispiel“. Übertrage mindestens ein Beispiel auf deine aktuelle Datei und prüfe das Verhalten praktisch.

8.4.7. Meta-Kommandos

Meta-Kommandos bilden die Brücke zwischen VIM und der Außenwelt: Sie erlauben den Aufruf von Shell-Befehlen, das Laden von Konfigurationsdateien und das Sichern kompletter Arbeitsumgebungen als Session.

- `:!ls` führt den Shell-Befehl `ls` aus und zeigt das Ergebnis im Kommandoausgabebereich. Beliebige andere Shell-Kommandos sind möglich, z. B. `:!git status` oder `:!make`.
- `:shell` startet eine interaktive Shell innerhalb von VIM. Mit `exit` oder [**Ctrl**]-[**d**] kehrst du zu VIM zurück. (Nicht in jeder Umgebung sinnvoll, z. B. in strikt eingebetteten Terminals.)

- `:source ~/.vimrc` liest eine VIM-Skriptdatei ein (z. B. `.vimrc` oder ein separates Plugin). Praktisch, um geänderte Konfiguration zu testen, ohne VIM neu zu starten.
- `:mksession session.vim` speichert die aktuelle Arbeitsumgebung (geöffnete Dateien, Fensteraufteilung, Cursorpositionen) in eine Session-Datei. Mit `vim -S session.vim` oder `:source session.vim` kannst du diese Session später wiederherstellen.

Meta-Kommandos sind besonders nützlich, wenn du VIM als zentrale „Steuerkonsole“ für Build-Prozesse, Tests oder Auswertungen verwendest, ohne den Editor verlassen zu müssen.

Übungen

1. Öffne ein Projektverzeichnis in VIM und führe nacheinander aus:
   - `:!pwd`
   - `:!ls`

   Vergleiche die Ausgabe mit dem, was dein normales Terminal anzeigen würde.
2. Erzeuge eine kleine Testkonfiguration, z. B. in einer Datei `test.vim` mit folgendem Inhalt:

   ```
   set number
   set cursorline
   ```

   Öffne eine Datei in VIM und lade diese Konfiguration mit `:source test.vim`. Beobachte, welche Änderungen sofort sichtbar werden.
3. Öffne mehrere Dateien in geteilten Fenstern (vertikal oder horizontal) und speichere den Zustand mit `:mksession vim-session.vim`. Beende VIM, starte ihn neu und lade die Session mit `vim -S vim-session.vim`. Ziel: Du solltest exakt in der gleichen Fensteraufteilung und an den gleichen Positionen weiterarbeiten können.
4. Wenn deine Umgebung es zulässt: Starte mit `:shell` eine interaktive Shell aus VIM heraus, führe dort ein oder zwei Befehle aus (z. B. `ls`, `uname -a`) und kehre mit `exit` zurück. Achte darauf, wie VIM nach dem Verlassen der Shell den Bildschirm neu zeichnet.

8.4.8. Quickfix-Workflow

Der Quickfix-Mechanismus verbindet externe Prüfwerkzeuge (Compiler, Linter, Tests) mit VIM. Typischer Ablauf:

1. Externes Kommando ausführen (z. B. `:make` oder `:grep`).
2. Die Ausgabe wird in eine Quickfix-Liste geparst.
3. Mit Quickfix-Kommandos durch die Fundstellen springen.

Die wichtigsten Befehle dafür:

- `:make` führt das im Editor gesetzte `makeprg` aus (standardmäßig `make`) und schreibt Fehler/Warnungen in die Quickfix-Liste.
- `:copen` öffnet das Quickfix-Fenster am unteren Rand. Mit **Enter** auf einem Eintrag springst du zur entsprechenden Zeile im Quelltext.
- `:cclose` schließt das Quickfix-Fenster wieder, ohne die Liste zu löschen.
- `:cnext` / `:cprev` springt zum nächsten bzw. vorherigen Eintrag in der Quickfix-Liste. Praktisch als „Fehler vor / zurück“.
- `:cfirst` / `:clast` springt direkt zum ersten bzw. letzten Eintrag in der Liste.
- `:cc` zeigt den aktuellen Quickfix-Eintrag noch einmal an (z. B. nach einer Änderung im Code).

Typischer Minimal-Workflow beim Kompilieren eines Projekts:

1. `:make` ausführen.
2. `:copen`, um die Fehlerliste zu sehen.
3. Mit `:cnext` / `:cprev` und **Enter** alle Fundstellen abarbeiten.
4. Zum Schluss `:cclose`, wenn du die Liste nicht mehr brauchst.

Übungen

1. Erstelle ein kleines C- oder Python-Projekt mit absichtlich eingebauten Fehlern (z. B. Tippfehler im Funktionsnamen, fehlendes Semikolon, Syntaxfehler). Führe dann in VIM nacheinander aus:
   - `:make`
   - `:copen`
   - `:cnext`, `:cprev`, `:cc`

   Ziel: Alle Fehler einmal gezielt anspringen und korrigieren.
2. Erzeuge eine Quickfix-Liste durch ein anderes Kommando, z. B. mit einem `grep`-Wrapper (je nach Konfiguration):
   - `:grep TODO \*.c`
   - `:copen`

   Navigiere mit `:cnext` und `:cprev` durch alle TODO-Stellen im Code.
3. Passe in deiner VIM-Konfiguration `makeprg` vorübergehend an, z. B.:

```
:set makeprg=pytest
```

Führe `:make` aus und beobachte, wie Testfehler in der Quickfix-Liste landen. Ziel: Verstehen, dass Quickfix nicht nur für Compiler, sondern für beliebige Werkzeuge nutzbar ist.

Dieser Überblick dient als zentrale Referenz: Alle späteren Kapitel bauen auf den hier gezeigten Mustern auf (insbesondere Bereiche, Filter, globale Kommandos, Quickfix-Mechanismen und komplexe Ersetzungen).

9. Makros und Register

9.1. Makros aufzeichnen

Makros gehören zu den zentralen Produktivitätswerkzeugen in VIM. Sie erlauben es, eine beliebige Folge von Befehlen *aufzuzeichnen* und anschließend beliebig oft wieder abzuspielen — von einmaliger Wiederholung bis hin zu hunderten Zeilen in einem Rutsch.

9.1.1. Grundprinzip

Jedes Makro wird in einem Register gespeichert. Die Aufzeichnung beginnt mit:
[q] `<reg>` (Aufzeichnung starten)
und endet mit:
[q] (Aufzeichnung beenden)

Dabei ist `<reg>` ein normales Register: [a]–[z] oder [0]–[9]. Häufig wird [q] als Makro-Register verwendet, weil es gut erreichbar ist.

Typischer Ablauf:

[q][q] … gewünschte Befehle ausführen … [q]

Dies speichert die komplette Folge der eingegebenen Befehle im Register [q]. Abspielen erfolgt später mit [@][q].

Übungen

- Erzeuge einige Testzeilen:

  ```
  foo
  foo
  foo
  ```

  Nimm ein Makro in Register [q] auf, das aus `foo` den Text `bar` macht. Das entspricht der Tastensequenz: [q][q][c][i][w]`bar`[Esc][q].

Spiele das Makro anschließend mit @ q und @ @ auf den weiteren Zeilen ab.

- Zeichne ein Makro q a auf, das am Zeilenende ein Semikolon anhängt.
  Die zugehörige Sequenz lautet: q a A ; Esc q.
  Teste anschließend, wie sich @ a auf unterschiedlich lange Zeilen verhält.

9.1.2. Makro ausführen

Ein gespeichertes Makro wird mit:
@ q
ausgeführt. Dabei steht q für das Register, in dem das Makro aufgezeichnet wurde. Entsprechend funktionieren auch:

- @ a Makro aus Register a
- @ b Makro aus Register b

9.1.3. Letztes Makro wiederholen

Das zuletzt *ausgeführte* Makro (nicht zwingend das zuletzt *aufgezeichnete*) wird mit:
@ @
erneut gestartet. Das ist besonders praktisch, wenn dasselbe Makro Schritt für Schritt auf viele Zeilen angewendet werden soll.

9.1.4. Makros und Wiederholung mit .

Wichtig ist die Abgrenzung zwischen Makro-Wiederholung und Änderungs-Wiederholung:

- @ @ wiederholt *das gesamte Makro*
- . wiederholt nur *die letzte Änderung*

Führt ein Makro mehrere Änderungen aus, dann referenziert . nur die *letzte* dieser Änderungen, nicht das komplette Makro. Das führt leicht zu Missverständnissen: Wer glaubt, mit . das Makro zu wiederholen, bekommt in Wahrheit nur den letzten Änderungsschritt daraus.

Praxis-Merksatz

Ein sauber strukturiertes Makro, das du mit @ q und @ @ steuerst, ist verlässlicher als ein „ . “, das nur zufällig das Richtige wiederholt.

Übungen

- Zeichne ein Makro in Register q auf, das am Zeilenende ein Semikolon anhängt und in die nächste Zeile springt (z. B. q A ; Enter q). Spiele es auf mehreren Zeilen mit @ q und anschließend mit @ @ durch.
- Füge in ein bestehendes Makro gezielt zwei Änderungen ein (z. B. erst c i w, dann A ; in derselben Zeile). Teste anschließend, was . nach dem Makro noch wiederholt, und stelle fest, dass nur der letzte Schritt reproduziert wird.

9.1.5. Makrotaugliche Regeln

Für zuverlässige Makros gelten einige robuste Stilregeln:

- keine Pfeiltasten verwenden (Terminal- und Mapping-abhängig)
- wiederholbare Bewegungen einsetzen (w, f X, / pattern Enter)
- Textobjekte bevorzugen (c i w, d a (, y i ”)
- Undo-Blöcke klein halten (möglichst wenige, klar abgegrenzte Änderungen)
- keine Interaktion erzwingen (keine Bestätigungsdialoge wie Ex%s/.../.../gc)

Makros werden stabiler, wenn sie nur aus deterministischen Bewegungen und Änderungen bestehen, die unabhängig von Fensterbreite, Zeilenlänge oder Cursor-Feinposition funktionieren.

9.1.6. Makros auf mehrere Zeilen anwenden

Ein Makro in Register q kann mit einem Count mehrfach ausgeführt werden:

1 0 0 @ q

führt das Makro @ q hundertmal hintereinander aus. Das ist besonders nützlich für Transformationen auf:

- Listen (z. B. Checklisten, Bullet-Points, To-do-Listen)
- Tabellen (Spalten ergänzen, Trennzeichen ändern)
- Logdateien (Zeilen umformatieren, Präfixe/Suffixe hinzufügen)

Übungen

- Zeichne ein Makro in Register **q** auf, das am Zeilenende ein Semikolon ergänzt und zur nächsten Zeile springt (z. B. **q** **A** **;** **Enter** **q**). Wende es mit **5** **@** **q** auf fünf Zeilen an und prüfe, ob jede Zeile korrekt abgeschlossen wurde.
- Erstelle ein Makro, das einen Logeintrag von:

  ```
  INFO start
  ```

  nach:

  ```
  [INFO] start
  ```

  umformatiert (z. B. mit **0** **c** **i** **w** `[INFO]` **Esc**). Wende es mit **2** **0** **@** **q** auf eine längere Logliste an und beobachte, ob es auch bei leicht unterschiedlich langen Zeilen stabil bleibt.

9.1.7. Makro prüfen

Nach der Aufzeichnung kann ein Makro mit einem Blick in die Register überprüft werden. Typischerweise nutzt man dazu:

Exreg q

Damit zeigt VIM den Inhalt des Registers **q** an – inklusive aller aufgezeichneten Tastendrücke. So lassen sich z.B. gut erkennen:

- versehentlich verwendete Pfeiltasten
- unnötig lange Cursorbewegungen
- überflüssige **Esc**-Sequenzen
- problematische Suchmuster oder Ersetzungen

Weitere nützliche Varianten:

- Exreg zeigt alle Register
- Exreg a zeigt nur Register **a**

Das Prüfen eines Makros über Exreg ist der wichtigste Schritt, um Makros gezielt zu optimieren, zu verkürzen und robuster zu machen.

Übungen

- Nimm ein einfaches Makro auf:

q "q A // TODO Esc j q

Lass dir den Inhalt mit Exreg q anzeigen. Identifiziere genau, welche Zeichen für A, // TODO, Esc und j gespeichert wurden.

- Zeichne absichtlich ein „schlechtes“ Makro mit Pfeiltasten auf und inspiziere es mit Exreg. Überlege, wie du es mit w, 0, $ oder Textobjekten stabiler formulieren könntest.
- Nimm ein reales Arbeitsmakro auf, das du tatsächlich benutzen willst, und kontrolliere es konsequent mit Exreg. Entferne alles, was nicht deterministisch ist (z. B. <Up> / <Down>).

9.2. Makros ausführen

Sobald ein Makro in einem Register gespeichert ist, kann es jederzeit ausgeführt werden. Die Ausführung erfolgt immer mit dem @-Operator und ist selbst wieder per Count skalierbar.

9.2.1. Makro in bestimmtem Register ausführen

Ein Makro im Register q wird ausgeführt mit:

@ q

Damit startet VIM die komplette im Register gespeicherte Sequenz – exakt so, wie sie aufgezeichnet wurde.

Mit einem Count davor lässt sich das Makro mehrfach hintereinander ausführen:

1 0 @ q

führt das Makro aus Register q zehnmal nacheinander aus, z.B. für Listen, Tabellen oder Logzeilen.

Bereits bekannt, aber im Kontext wichtig:

- @ @ führt das zuletzt ausgeführte Makro erneut aus
- Counts funktionieren auch hier: 5 @ @ wiederholt das letzte Makro fünfmal

Übungen

- Erzeuge zehn identische Zeilen mit einer Zahl am Anfang, z.B.:

  ```
  1 foo
  1 foo
  1 foo
  ...
  ```

  Nimm ein Makro in [q] auf, das die führende 1 in 2 ändert und zur nächsten Zeile springt. Führe es mit [@][q] aus und wiederhole es mit [@][@], bis alle Zeilen angepasst sind.
- Zeichne ein Makro, das am Zeilenende ein Semikolon anhängt ([A][;][Esc][j]). Nutze [1][0][@][q], um zehn Zeilen hintereinander zu bearbeiten. Kontrolliere mit [u], ob die Änderung überall reproduzierbar war.
- Nimm zwei verschiedene Makros in [q] und [w] auf. Führe sie abwechselnd mit [@][q], [@][w] und [@][@] aus und beobachte, welches Makro bei [@][@] jeweils wiederholt wird.

9.2.2. Letztes ausgeführtes Makro wiederholen

Die Wiederholung des zuletzt *ausgeführten* Makros erfolgt mit:

[@][@]

Wichtig ist die Unterscheidung:

- [@][@] bezieht sich immer auf das *zuletzt ausgeführte* Makro, nicht zwingend auf das zuletzt aufgezeichnete.
- Wird z.B. zuerst [@][q] und danach [@][a] ausgeführt, wiederholt [@][@] das Makro aus Register [a].
- [@][@] ist ideal, wenn dasselbe Makro schrittweise auf nachfolgende Zeilen angewendet werden soll (z.B. jeweils nach einem [j]).

Wie bei anderen Befehlen kann auch hier ein Count verwendet werden:

[5][@][@]

führt das zuletzt ausgeführte Makro fünfmal hintereinander aus.

Übungen

- Nimm ein Makro in Register q auf, das am Zeilenende ein Semikolon anhängt und zur nächsten Zeile springt (z.B. q A ; Esc j q). Führe es einmal mit @ q aus und dann mehrmals mit @ @. Beobachte, dass du nach dem ersten Aufruf nur noch @ @ brauchst.
- Zeichne zwei Makros in q und w auf und führe sie abwechselnd mit @ q und @ w aus. Prüfe jeweils mit @ @, welches Makro gerade als „letztes“ gespeichert ist.

9.2.3. Makro mehrfach hintereinander ausführen

Ein Makro lässt sich mit einem Count vervielfachen, indem man vor @ `<reg>` eine Zahl setzt:

1 0 @ q

führt das Makro aus Register q zehnmal hintereinander aus – ohne dass man @ q manuell wiederholt.

Typische Anwendungsfälle:

- Listen transformieren (z.B. jedes Listenelement umformatieren)
- Logzeilen umformatieren (Zeitstempel, Präfixe, Suffixe)
- Nummerierungen oder Zähler inkrementieren
- Tabellenspalten oder CSV-Spalten bearbeiten

In Kombination mit @ @ ergibt sich ein flexibles Muster:

- @ q einmalig testen, ob das Makro tut, was es soll
- 1 0 @ q große Bereiche automatisiert bearbeiten
- @ @ einzelne Nachkorrekturen dort, wo nötig

Übungen

- Erzeuge 15 identische Zeilen, z.B. TODO `item`. Nimm ein Makro in q auf, das am Zeilenanfang eine laufende Nummer ergänzt (z.B. I 1 . Esc j). Führe es mit
- 1 5 @ q aus und kontrolliere, ob alle Zeilen korrekt nummeriert wurden.

- Erstelle eine einfache Logliste und zeichne ein Makro auf, das am Zeilenende einen Zeitstempel (oder Platzhalter wie <TS>) anhängt und zur nächsten Zeile springt. Wende es mit
- 2 0 @ q auf einen ganzen Logblock an und prüfe anschließend mit u, wie viele Änderungen als ein Undo-Block zusammengefasst wurden.

9.2.4. Auf andere Register zugreifen

Makros können nicht nur in q, sondern in jedes benannte Register a–z aufgezeichnet werden, zum Beispiel:

q a, q b, q z

Vorteile dieser Aufteilung:

- mehrere Makros lassen sich parallel vorhalten und gezielt aufrufen
- komplexe Bearbeitungsabläufe können sauber auf verschiedene Register verteilt werden (z. B. ein Makro für Nummerierung, eines für Formatierung, eines für Kommentare)

Es ist sinnvoll, für thematisch zusammengehörende Makros feste Register zu reservieren (z. B. q für Logbearbeitung, w für Tabellen).

9.2.5. Makro im Register einsehen

Mit dem Befehl

```
Exreg
```

werden alle Register angezeigt, inklusive deren Inhalt. Um ein bestimmtes Register zu prüfen, genügt:

```
Exreg q
```

Dies zeigt den Inhalt des Registers q — inklusive aller Steuerzeichen.

Wichtige Prüfpunkte:

- sind Pfeiltasten enthalten, die das Makro instabil machen?
- wurde versehentlich *Text* aufgezeichnet, der dort nicht hingehört?
- befinden sich unerwünschte Newlines oder leere Befehle im Makro?

Auffälligkeiten lassen sich gezielt bereinigen, indem das Makro neu aufgezeichnet oder mit einem besseren Bewegungsmuster ersetzt wird.

9.2.6. Makro abbrechen

Wenn ein Makro „aus dem Ruder läuft“ (Endlosschleife, falscher Bereich, unerwartete Eingaben), kann die Ausführung sofort abgebrochen werden:

Ctrl - **c**

oder — je nach Terminal/Umgebung — in vielen Fällen auch mit:

Ctrl - **Break**

Damit wird die Makroausführung hart gestoppt, ohne VIM komplett zu beenden.

Übungen

- Zeichne zwei unterschiedliche Makros in **a** und **b** auf (z.B. eines für das Anhängen eines Semikolons, eines für das Einfügen eines Kommentars). Prüfe anschließend mit Exreg a und Exreg b, ob die Sequenzen wie erwartet gespeichert wurden.
- Zeichne absichtlich ein Makro mit Pfeiltasten auf und sieh dir mit Exreg q an, wie diese im Register dargestellt werden. Zeichne das Makro danach noch einmal „sauber“ mit **w**, **f** **X** oder Textobjekten neu auf und vergleiche beide Varianten.
- Erzeuge ein Makro, das mehrfach nach unten springt (z.B. **q** **j** **j** **j** **j** **q**) und starte es in einem Bereich, in dem es „wegzulaufen“ droht. Brich die Ausführung mit **Ctrl** - **c** ab und beobachte, an welcher Stelle VIM stoppt.

9.3. Registerübersicht

Register sind zentrale Bausteine der VIM-Arbeitsweise. Sie dienen als temporäre oder permanente Speicher für:

- Makros
- gelöschte Texte
- kopierte Texte (Yank)
- Systemzwischenablage (Clipboard)
- Dateinamen, Suchmuster und Ex-Befehle

Jedes Register wird durch ein einzelnes Zeichen identifiziert und kann über das Präfix **”** angesprochen werden (z. B. **”** **a**, **”** **0**, **”** **+**).

9.3.1. Typen von Registern

VIM kennt mehrere feste Kategorien von Registern:

- **Benannte Register** (" a – " z)
  werden explizit beschrieben, etwa durch Yanks, Deletes oder Makroaufzeichnung. Typische Nutzung:
  - " a y y Zeile in Register a kopieren
  - q a … q Makro in Register a aufzeichnen
- **Großbuchstaben** (" A – " Z)
  hängen an bestehende Register an (*Append* statt Überschreiben):
  - " A y y aktuelle Zeile an Register a anhängen
  - " B y y Inhalt an Register b anhängen
- **Numerische Register** (" 0 – " 9)
  werden automatisch gefüllt:
  - " 0 enthält den letzten expliziten Yank (y)
  - " 1 – " 9 enthalten ältere Lösch- und Änderungsoperationen

  Sie eignen sich, um frühere Stände wieder einzufügen, ohne neu kopieren zu müssen.
- **Zwischenablage-Register** (" *, " +)
  stellen die Verbindung zum System her:
  - " + y y Zeile ins System-Clipboard kopieren
  - " * p Inhalt der X11-PRIMARY-Selektion einfügen (unter Linux/X11)

  Unter macOS und vielen Linux-Setups ist " + das wichtigste Clipboard-Register.
- **Spezialregister**

  Diese werden von VIM intern geführt:
  - " _ Black-Hole-Register (schluckt alles, ohne etwas zu speichern)
    z. B. " _ d w löscht ein Wort, ohne die Register zu verändern
  - " % aktueller Dateiname
  - " # alternativer Dateiname (zuletzt geöffnete andere Datei)
  - " / letztes Suchmuster (nutzt / –Eingabe)
  - " : letzter Ex-Command

– [”][@] zuletzt ausgeführtes Makro

Übungen

- Erzeuge zwei verschiedene Textzeilen und kopiere sie nacheinander mit [”][a][y][y] und [”][b][y][y]. Füge anschließend mit [”][a][p] und [”][b][p] ein und kontrolliere, dass beide benannten Register unabhängig voneinander arbeiten.
- Kopiere eine Zeile mit [y][y], lösche anschließend mehrere andere Zeilen mit [d][d]. Nutze `:reg 0` und `:reg 1-:reg 3`, um zu sehen, wie sich Yank- und Delete-Operationen in den numerischen Registern verteilen.
- Lösche ein störendes Wort mit [”][_][d][w] und überprüfe danach mit Exreg, dass weder [”][0] noch die numerischen Delete-Register verändert wurden. Diese Technik ist wichtig, um bestehende Registerinhalte zu schützen.
- Kopiere eine Zeile mit [”][+][y][y], wechsle in ein anderes Programm (Terminal/GUI) und füge dort ein. Danach füge denselben Inhalt in VIM mit [”][+][p] ein, um das Zusammenspiel mit der Systemzwischenablage zu verstehen.

9.3.2. Benannte Register (”a–”z)

Benannte Register werden explizit verwendet, um Text oder Makros gezielt abzulegen. Sie eignen sich für wiederkehrende Operationen, die nicht von den automatischen Registern überschrieben werden sollen.

Typische Nutzung:

- [”][a][y] Text in Register [”][a] kopieren
- [”][a][p] Inhalt von Register [”][a] einfügen
- [q][a] Makro in Register [a] aufzeichnen
- [@][a] Makro aus Register [a] ausführen

Großbuchstaben ([”][A]–[”][Z]) hängen an bestehende Register an (*Append* statt Überschreiben):

- [”][A][y] an das bestehende Register [”][a] anhängen
- [”][A][d] gelöschten Text an Register [”][a] anhängen

So lassen sich komplexe Textbausteine oder längere Makros schrittweise aufbauen, ohne Zwischenschritte zu verlieren.

9.3.3. Numerische Register ("0–"9)

Numerische Register werden von VIM automatisch gepflegt und protokollieren Yanks und Löschvorgänge:

- "0 enthält den letzten expliziten Yank (y-Operation)
- "1–"9 enthalten gelöschte Texte (Delete-Historie)
- "1 ist dabei immer der zuletzt gelöschte Text

Wichtige Konsequenzen:

- Ein yy oder yw schreibt in "0, ohne die Delete-Historie "1–"9 zu verändern.
- Ein dd oder dw schiebt den bisherigen Inhalt von "1 nach "2, "2 nach "3 usw. und legt den neuen Löschtext in "1 ab.
- Mit "1p, "2p usw. können ältere Löschstände gezielt wieder eingefügt werden.

Übungen

- Kopiere eine Zeile mit "ayy in Register "a und füge sie darunter mit "ap wieder ein. Wiederhole das mit einem anderen Text und vergleiche das Verhalten mit einem normalen yy / p.
- Lösche nacheinander mehrere Zeilen mit dd und inspiziere dann die Register mit Exreg 0 1 2. Prüfe, wo Yank- und Delete-Inhalte tatsächlich gelandet sind.
- Erzeuge einen Textblock in Register "a mit "ay und ergänze ihn anschließend mit "Ay. Füge danach mit "ap ein und kontrolliere, dass beide Teile korrekt hintereinander stehen.
- Kopiere einen Text mit yy, lösche danach mehrere andere Zeilen mit dd und füge dann explizit aus "0 mit "0p ein. Überprüfe, dass der ursprüngliche Yank weiterhin verfügbar ist.

9.3.4. Zwischenablage-Register ("+, "*)

Über die Zwischenablage-Register wird direkt auf das System-Clipboard zugegriffen:

- "+y kopiert in das macOS-/Linux-Clipboard

- [”][+][p] fügt aus dem Clipboard ein
- [”][*][y] kopiert in die X11-Auswahl (PRIMARY-Selection unter Linux)

Auf macOS sind [”][+] und [”][*] in der Regel identisch, unter Linux können sie sich unterscheiden:

- [”][+] meist das „normale“ Clipboard (Copy/Paste mit GUI)
- [”][*] oft die Auswahl (Markieren mit der Maus, Einfügen mit mittlerer Maustaste)

Typische Anwendungsfälle:

- Text aus VIM in andere Programme kopieren: [”][+][y][y], dann in der GUI „Einfügen“
- Text aus dem Browser in VIM einfügen: im Browser kopieren, in VIM [”][+][p]

Übungen

- Kopiere eine Zeile in VIM mit [”][+][y][y] und füge sie testweise in ein anderes Programm (z. B. Terminal, Editor, Browser) ein.
- Kopiere einen Text aus einem externen Programm und füge ihn in VIM mit [”][+][p] ein. Prüfe, ob Umlaute und Sonderzeichen korrekt übernommen werden.
- Unter Linux (mit X11): teste den Unterschied zwischen [”][+][y] und [”][*][y], indem du einmal mit der Zwischenablage und einmal mit der PRIMARY-Selection arbeitest.

9.3.5. Black-Hole-Register (”_)

Das Black-Hole-Register [”][_] nimmt Text an, speichert ihn aber nicht. Der Inhalt landet *nirgendwo* und beeinflusst weder das Yank- noch das Delete-Register.

Beispiele:

- [”][_][d] löscht Text, ohne die Delete-Historie ([”][1]–[”][9]) zu verändern
- [”][_][d][d] löscht eine Zeile, ohne den Inhalt in ein Register zu schreiben
- [”][_][c][a][w] ändert ein Wort, ohne dass der alte Inhalt in einem Register landet

Typischer Einsatz:

- „Müll löschen“, ohne wichtige Yanks oder Deletes zu überschreiben
- beim Refactoring alte Zwischenstände nicht versehentlich in [”][0]–[”][9] schieben
- in Makros sicherstellen, dass nur *gewollte* Inhalte in Registern landen

Übungen

- Kopiere zunächst eine wichtige Zeile mit [y][y]. Lösche anschließend eine andere Zeile einmal mit [d][d] und einmal mit [”][_][d][d]. Prüfe mit [”][0][p] und [”][1][p], welcher Text tatsächlich im Register gelandet ist.
- Ersetze testweise ein Wort mit [c][a][w] und danach mit [”][_][c][a][w]. Untersuche mit Exreg 0 1, ob der alte Wortinhalt in einem Register abgelegt wurde.
- Baue ein kleines Makro, das mit [”][_][d][d] arbeitet, und vergleiche das Verhalten zu einem Makro mit normalem [d][d]. Ziel: Deine wichtigen Yanks sollen im Register [”][0] erhalten bleiben.

9.3.6. Register anzeigen

Mit: Exreg
werden alle Register angezeigt.
Mit: Exreg a
wird nur Register a angezeigt — ideal zur Makrokontrolle.

9.4. Register gezielt nutzen

Register sind nicht nur Speicher, sondern ein strategisches Werkzeug für Makros, komplexe Bearbeitungen und saubere Arbeitsabläufe. Dieser Abschnitt zeigt praxisnahe Muster für effiziente Nutzung.

9.4.1. Gezielt in ein Register schreiben

Alle Operatoren (y, d, c) können mit einem Register kombiniert werden:
[”][a][y] kopiert in Register "a
[”][b][y][i][w] kopiert das innere Wort in Register "b

[”][c][d][a][s] löscht einen ganzen Satz in Register `"c`
[”][a][d][a][p] löscht einen Absatz und speichert ihn in Register `"a`

9.4.2. Register für Makros verwenden

Makros liegen ebenfalls in Registern:
[q][a] Makro in Register `a` aufzeichnen
[@][a] Makro ausführen
[@][@] zuletzt ausgeführtes Makro erneut ausführen

Tipp: Makros niemals in `0`–`9` speichern, da diese Register durch Lösch- und Yank-Operationen automatisch überschrieben werden.

9.4.3. Anhängen an Register

VIM erlaubt das Erweitern eines Registers:
[”][A][y] hängt Text an Register `a` an
[”][A][d] hängt Delete-Ergebnisse an Register `a` an

Perfekt, um Textfragmente über mehrere Aktionen hinweg gezielt in einem Register zu sammeln.

9.4.4. Delete-History sauber halten

Das Black-Hole-Register verhindert, dass unerwünschte Inhalte in der Historie landen:
[”][_][d][a][w] löscht das Wort, ohne Register *oder* Delete-History zu verändern

9.4.5. Clipboard präzise steuern

Auf macOS:
[”][+][y] in das System-Clipboard kopieren
[”][+][p] aus dem System-Clipboard einfügen

9.4.6. Register kombinieren

Register können flexibel im laufenden Prozess gewechselt werden:

- Text einfügen mit [”][a][p]

- anderes Register beschreiben mit [”] [b] [y] [i] [w]
- ein Makro ausführen mit [@] [c]

So entstehen hochflexible, aber deterministische Arbeitsabläufe.

9.4.7. Register inspizieren

Mit:

```
Exreg
```

wird der gesamte Registerinhalt angezeigt.
Mit:

```
Exreg a
```

nur der Inhalt eines bestimmten Registers.

9.5. Makro-Optimierung

Makros gehören zu den mächtigsten Werkzeugen in VIM. Einfache Makros automatisieren zwar bereits Routineabläufe, ihr volles Potenzial entfalten sie aber erst, wenn sie gezielt *optimiert* werden. Das Ziel lautet immer:

stabil – deterministisch – reproduzierbar

9.5.1. Grundprinzipien stabiler Makros

Ein gutes Makro erfüllt drei zentrale Eigenschaften:

- Es hängt nicht von der aktuellen Zeilenlänge oder Spaltenposition ab.
- Es vermeidet Seitenscrollen, Fensterwechsel, Mausbewegungen und ähnliche „UI-Effekte“.
- Es nutzt Operatoren, Bewegungen und Textobjekte statt Pfeiltasten.

Stabile Makros sollen auf möglichst beliebigen Texten funktionieren, ohne dass der Cursor an einer exakt definierten Spalte stehen muss. Operator–Motion-Kombinationen ([d] [a] [w], [c] [i] [w], [c] [i] [p] usw.) übernehmen dabei den größten Teil der „Intelligenz“ und sorgen dafür, dass das Makro sich an die Struktur des Textes anpasst.

Übungen

- Nimm ein bestehendes Makro, das mit Pfeiltasten arbeitet, und zeichne es neu auf, indem du ausschließlich w, b, 0, $ und Textobjekte wie i w, i p verwendest. Prüfe anschließend mit Exreg q, wie sich die Kommandosequenz vereinfacht hat.
- Erzeuge eine kleine Liste mit unterschiedlich langen Zeilen (z. B. Logzeilen mit Zeitstempel und Nachrichtentext). Entwirf ein Makro, das jeweils den Nachrichtentext umklammert, ohne sich auf eine fixe Spaltenposition zu verlassen. Nutze dazu f X, Textobjekte oder Suche (/ `pattern` Enter).
- Nimm ein fragiles Makro, das nur in einer bestimmten Situation funktioniert, und identifiziere systematisch alle Stellen, an denen es von Zeilenlänge, Cursorposition oder zufälligem Scrollen abhängt. Formuliere für jede dieser Stellen eine Alternative mit Textobjekten oder strukturellen Bewegungen.

9.5.2. Vermeiden instabiler Bewegungsarten

Instabil sind insbesondere:

- Pfeiltasten (<Up>, <Down>)
- horizontales Navigieren mit h, l
- Bildschirmbewegungen wie Ctrl-f, Ctrl-b

Diese Aktionen sind abhängig von Fenstergröße, Umbrüchen, Schriftarten und Scrollposition. Schon kleine Formatänderungen können ein Makro unbrauchbar machen.

Stattdessen sollten genutzt werden:

- **Textobjekte**: i w, a w, i s, a s
- **Bewegungen**: f X, t X, F X, T X
- **Strukturbewegungen**: %, [[,]]

9.5.3. Fehlervermeidung durch deterministische Schritte

Eine robuste Makro-Strategie besteht darin, *immer zuerst zu stabilisieren.* Das bedeutet:

- an den Zeilenanfang springen (0)
- oder an das erste Nicht-Whitespace-Zeichen gehen (^)

- oder gezielt ein Textobjekt bzw. eine Strukturposition ansteuern (f X, i w, i p ...)

Damit beginnt das Makro jedes Mal aus einem definierten Zustand und reagiert deutlich weniger empfindlich auf Layout- oder Inhaltsänderungen.

Übungen

- Nimm ein vorhandenes Makro, das noch mit Pfeiltasten oder h / l arbeitet. Zeichne es neu auf, indem du stattdessen 0, ^, w, f X und Textobjekte wie i w verwendest. Prüfe das Verhalten in mehreren leicht veränderten Zeilen.
- Erzeuge drei Varianten derselben Zeile (unterschiedliche Einrückung, zusätzliche Leerzeichen, leicht veränderte Wörter). Baue ein Makro, das jeweils das erste Wort nach der Einrückung ersetzt — einmal mit h / l, einmal mit ^ + w und c i w. Vergleiche, welche Variante stabil bleibt.

9.5.4. Makros modularisieren

Anstatt ein langes, schwer nachvollziehbares Makro zu bauen, ist es oft besser, mehrere kleine Makros zu kombinieren, die jeweils eine klar abgrenzbare Aufgabe haben:

- Makro a: Text vorbereiten oder „aufräumen“
- Makro b: Struktur ändern (z. B. Felder umsortieren)
- Makro c: Ausgabe oder Formatierung erzeugen

Vorteile der Modularisierung:

- einzelne Schritte sind leichter zu testen und zu debuggen
- Teile lassen sich in anderen Kontexten wiederverwenden
- Fehlerquellen sind klarer zu isolieren

9.5.5. Register für Makrotests nutzen

Für Experimente und Tests sollten eigene Register verwendet werden:

- Makros zunächst in q oder z entwickeln
- Testläufe auf Kopien des Textes durchführen
- erst bei Bewährung in „produktive“ Register wie a oder b übernehmen

So werden bestehende, bewährte Makros nicht versehentlich überschrieben oder zerstört.

9.5.6. Makros mit Textobjekten kombinieren

Die Kombination aus Operatoren und Textobjekten sorgt für präzise, layoutunabhängige Makros:

d a w löscht ein Wort inklusive Leerraum zuverlässig

c i (ändert den Inhalt einer runden Klammer

y a s kopiert einen gesamten Satz zur Weiterverarbeitung

Makros, die auf Textobjekten und strukturellen Bewegungen basieren, sind deutlich stabiler als solche, die mit „so vielen Zeichen nach rechts“ oder „zweimal runter, dreimal hoch“ arbeiten.

Übungen

- Zeichne ein langes Makro in **q** auf, das mehrere Aufgaben erledigt (z. B. Leerraum bereinigen, Felder umsortieren, Kommentar anhängen). Analysiere danach mit Exreg q, welche Teile sich sinnvoll in einzelne Makros **a**, **b**, **c** zerlegen lassen.
- Erzeuge zwei Versionen desselben Makros: einmal mit Pfeiltasten/**h**/**l**, einmal mit Textobjekten wie **d a w**, **c i (**, **c i s**. Teste beide Varianten auf leicht veränderten Zeilen und vergleiche, welche Version stabil bleibt.

9.5.7. Makro-Debugging

Makros lassen sich komfortabel analysieren über:

```
Exreg q
```

Damit werden alle gespeicherten Schritte sichtbar, inklusive eingegebener Texte und Sondertasten. Beim Debuggen gilt:

- Stellen mit Pfeiltasten identifizieren und durch strukturelle Bewegungen ersetzen
- unnötige Schritte (z. B. doppelte Bewegungen) konsequent entfernen

- Bewegungen so umbauen, dass sie unabhängig vom Cursorzustand sind (z. B. 0, ^, Textobjekte, f X)

Insbesondere lohnt es sich, das Makro einmal Schritt für Schritt gedanklich durchzugehen: „Was passiert, wenn die Zeile etwas anders aussieht?“ Wo das Makro dann scheitert, muss nachgeschärft werden.

9.5.8. Best Practices

Für stabile, pflegeleichte Makros haben sich folgende Grundsätze bewährt:

- Makro so kurz wie möglich halten
- keine Scrollbewegungen wie Ctrl-f, Ctrl-b, Ctrl-d, Ctrl-u
- keine Pfeiltasten, sondern Operator–Motion–Kombinationen verwenden
- Operatoren + Textobjekte nutzen statt „roher“ Löschbefehle an Positionen
- keine zählbasierten Wiederholungen mit l, h, j, k, wenn passende Textobjekte existieren
- lieber f X oder t X nutzen, statt l l l oder ähnliche Sequenzen

Ein Makro gilt als optimiert, wenn es in möglichst vielen unterschiedlichen Textumgebungen *vorhersagbar* das gleiche Resultat liefert, ohne dass der Cursor vorher manuell „zurechtgeschoben“ werden muss.

Übungen

- Zeichne ein Makro in q auf, das offensichtlich Pfeiltasten und zeichenweise Bewegungen verwendet. Analysiere es mit Exreg q und ersetze Schritt für Schritt alle instabilen Teile durch Textobjekte (i w, a s, i p) und strukturierte Bewegungen (f X, %). Vergleiche die ursprüngliche und die optimierte Version.
- Nimm ein Makro, das nur in *einer* konkreten Beispieldatei funktioniert. Teste es auf drei leicht veränderten Varianten derselben Datei (z. B. zusätzliche Leerzeilen, andere Einrückung) und protokolliere, wo es scheitert. Überarbeite es so, dass diese Bruchstellen verschwinden.

10. Faltung (Folding)

10.1. Faltungen erzeugen

Faltungen (*folds*) ermöglichen es, große Textbereiche ein- oder auszuklappen, sodass komplexe Dateien übersichtlicher werden. Statt Kilometer zu scrollen, kannst du ganze Blöcke temporär „zusammenklappen“ und nur bei Bedarf wieder öffnen.

VIM kennt verschiedene Strategien, um Faltungen zu erzeugen:

- *manuelle* Faltungen (explizit gesetzte Bereiche),
- *bereichsbezogene* Faltungen (z. B. über Einrückung),
- *syntaxbasierte* Faltungen (z. B. Funktionsblöcke in Quelltexten),
- Faltungen über explizite *Marker*.

In diesem Abschnitt beginnen wir mit der manuellen Faltung, weil sie am unabhängigsten vom Dateityp ist und das Grundprinzip gut sichtbar macht.

10.1.1. Manuelle Faltungen

Manuelle Faltungen eignen sich für beliebigen Text – unabhängig von Syntax, Einrückung oder Dateityp. Du sagst VIM explizit, welche Zeilen zusammengefaltet werden sollen.

Typisch wird das in Buffern verwendet, in denen `:set foldmethod=manual` aktiv ist: Du definierst die Faltungen gezielt dort, wo du sie brauchst.

Falten per Bewegung

Im Normal Mode erzeugst du eine Faltung mit einem Operator + Bewegung:

> **z** **f** gefolgt von einer Bewegung

Damit wird eine Faltung über genau den Bereich gelegt, den die Bewegung abdeckt.

Beispiele:

- z f a p faltet den gesamten Absatz unter dem Cursor.
- z f { faltet vom Cursor bis zum Beginn des aktuellen Blocks (z. B. Funktionsanfang, je nach Dateistruktur).
- z f j faltet die aktuelle Zeile und die direkt folgende Zeile.

Merke: z f verhält sich wie ein Operator (ähnlich d oder y) – nur dass das Ergebnis keine Löschung oder Kopie ist, sondern eine Faltung über den betroffenen Bereich.

Falten über Zeilenanzahl

Statt einer frei gewählten Bewegung kannst du auch eine feste Zeilenanzahl angeben:

- z f 3 j faltet die aktuelle Zeile und drei Zeilen darunter (insgesamt vier Zeilen).
- 2 z f j wirkt wie zwei aufeinanderfolgende Faltungen über jeweils j – weniger intuitiv, aber im Prinzip nur eine Kombination aus Zählpräfix und Bewegung.

Wichtiger ist in der Praxis die Kombination aus z f und klaren Bewegungen (Absatz, Block, Suchtreffer).

Falten aus dem Visual Mode

Wenn der Bereich bereits markiert ist, kannst du ihn direkt falten:

1. Wechsle in den Visual Mode (z. B. v oder Ctrl - v).
2. Markiere den gewünschten Bereich.
3. Drücke z f.

VIM legt eine Faltung über genau diesen Visual-Bereich. Die Ex-Variante dazu ist:

```
:'<,'>zf
```

– sie macht dasselbe, ist aber im normalen Alltag selten nötig, weil z f im Visual Mode schneller ist.

Übungen

1. Öffne eine längere Datei mit mehreren Absätzen oder Funktionsblöcken. Erzeuge Faltungen mit:
   - z f a p

- z f j
- z f {

Beobachte, wie sich die Faltgrenzen im Text verhalten.

2. Markiere im Visual Mode einen größeren Block (z. B. eine komplette Funktionsdefinition) und falte ihn mit z f. Öffne und schließe die Faltung anschließend mit z o (öffnen) und z c (schließen). Ziel: „Muskelgedächtnis“ für das Trio z f, z o, z c.
3. Setze `:set foldmethod=manual` und lege mehrere Faltungen auf derselben Seite an (z. B. zwei verschiedene Abschnitte). Navigiere mit dem Cursor über die Zeilen und öffne jeweils nur die Faltung, die du gerade brauchst. Achte darauf, wie die Anzeige sich verändert und wie stark das Scrolling reduziert wird.

10.1.2. Faltungen über Bereiche

Faltungen können auch direkt über Ex-Bereiche erzeugt werden – also ohne Bewegungen im Normal Mode, sondern rein über Zeilennummern, Marken oder Suchmuster:

```
:10,42fold
```

Dies erzeugt eine Faltung von Zeile 10 bis 42. Solche Befehle sind besonders nützlich in Skripten, in Mappings oder wenn du definierte logische Blöcke immer wieder gleich falten willst.

Typische Varianten

- `:10,42fold` legt eine Faltung über die Zeilen 10 bis 42.
- `:.,.+20fold` faltet von der aktuellen Zeile bis 20 Zeilen darunter.
- `:'a,'bfold` faltet den Bereich zwischen den Marken `a` und `b`. Praktisch, wenn du den Bereich vorher mit m a und m b markiert hast.
- `:\%fold` legt eine einzige große Faltung über die gesamte Datei (entspricht `1,$fold`).
- `:/BEGIN/,/END/fold` faltet den Block zwischen zwei Mustern inklusive der Zeilen mit `BEGIN` und `END`.

Wie bei anderen Range-Kommandos gilt: Die Range wird zuerst ausgewertet, `:fold` arbeitet dann exakt auf der so bestimmten Schnittmenge.

Wann Ex-Ranges sinnvoll sind

- Wenn du wiederkehrende Blöcke immer gleich falten willst (z. B. bestimmte Kapitel, Funktionsgruppen, Konfigurationsblöcke).
- Wenn du Faltungen in vimrc-Skripten oder Makros definierst.
- Wenn die Bewegungsvariante mit **z** **f** zu ungenau oder zu umständlich ist (lange Blöcke, komplexe Muster).

Übungen

1. Öffne eine Datei mit mindestens 100 Zeilen. Erzeuge eine Faltung über die Zeilen 10–30 mit:
   - `:10,30fold`

   Kontrolliere mit **z** **o** (öffnen) und **z** **c** (schließen), wie sich die Faltung verhält.
2. Setze zwei Marken in der Datei, z. B.:
   - **m** **a** am Beginn eines logischen Blocks
   - **m** **b** am Ende des Blocks

   Erzeuge dann eine Faltung mit:
   - `:'a,'bfold`

   Bewege den Cursor durch die Datei und beobachte, wie der gefaltete Block dargestellt wird.
3. Erzeuge mehrere Blöcke, die mit BEGIN und END eingefasst sind. Falte einen dieser Blöcke mit:
   - `:/BEGIN/,/END/fold`

   Wiederhole den Befehl, bis du sicher bist, wie VIM den Bereich bestimmt (inklusive der Markerzeilen) und wie sich das auf verschachtelte Blöcke auswirkt.

10.1.3. Markerbasierte Faltungen

Eine sehr kontrollierte Faltungsart sind *marker folds*. Hier bestimmt nicht die Syntax oder Einrückung die Faltungen, sondern explizite Markierungen im Text.

Typisches Muster (z. B. in C- oder Konfigurationsdateien):

```
// {{{   Beginn der Faltung
Inhalt...
// }}}   Ende der Faltung
```

Aktiviert wird diese Methode über den Faltmodus:

```
:set foldmethod=marker
```

Standardmäßig verwendet VIM dabei die Marker `{{{` und `}}}`. In vielen Dateitypen ist es üblich, diese Marker als Kommentare zu schreiben (z. B. `// {{{` in C, `" {{{` in Vimscript), damit der Code weiterhin gültig bleibt.

Konfiguration der Marker

Die Marker selbst sind einstellbar:

- `:set foldmarker={{{,}}}` verwendet explizit `{{{` als Start- und `}}}` als Endmarker (Standard).
- `:set foldmethod=marker` schaltet dauerhaft auf markerbasierte Faltungen um.
- Kombination in der vimrc:

```
set foldmethod=marker
set foldmarker={{{,}}}
```

Wichtig: Start- und Endmarker müssen *genau* übereinstimmen – VIM interpretiert sie rein textuell, nicht kontextsensitiv.

Verschachtelte Marker

Marker lassen sich beliebig verschachteln, etwa so:

```
// {{{ Top-Level
...
// {{{ Unterabschnitt
...
// }}} Unterabschnitt
...
// }}} Top-Level
```

VIM zeigt daraus verschachtelte Faltungen an. Über die Optionen `foldlevel` und `foldnestmax` kannst du steuern, wie tief die Faltungen initial geöffnet oder geschlossen sind:

- `:set foldlevel=0` – alles zu
- `:set foldlevel=1` – nur äußerste Ebene offen
- `:set foldnestmax=3` – maximal drei Verschachtelungsebenen

Typische Einsatzszenarien

Markerbasierte Faltungen sind besonders nützlich für:

- lange Konfigurationsdateien mit logisch getrennten Blöcken,
- große Skripte oder Quelltexte mit vielen Funktionsgruppen,
- strukturierte Dokumente (z. B. Abschnitte, Kapitel, Beispielblöcke),
- „Handbuch"-Dateien, in denen du einzelne Themenbereiche dynamisch ein- und ausklappen möchtest.

Im Gegensatz zu syntax- oder indentbasierten Faltungen bleiben markerbasierte Faltungen stabil, auch wenn sich Einrückung oder Syntax kurzfristig verändern – solange die Markerzeilen unverändert bleiben.

Bedienung im Alltag

Ist `foldmethod=marker` aktiv, bleibt die Bedienung identisch zu anderen Faltarten:

- z c – aktuelle Faltung schließen
- z o – aktuelle Faltung öffnen
- z M – alle Faltungen schließen
- z R – alle Faltungen öffnen
- z a – Faltung unter dem Cursor umschalten

Die Marker dienen nur zur *Definition* der Faltungen; das Falt-Verhalten selbst wird immer mit den z –Kommandos gesteuert.

Übungen

1. Erzeuge eine Testdatei mit mehreren logisch getrennten Blöcken (z. B. „Konfiguration", „Funktionen", „Tests") und setze darum herum Marker wie im Beispiel oben:

   ```
   // {{{ Konfiguration
   ...
   // }}} Konfiguration
   ```

   Aktiviere dann:

   - `:set foldmethod=marker`

   und spiele mit z M, z R, z c, z o.

2. Ergänze verschachtelte Marker in einem Block (z. B. Unterabschnitte innerhalb eines Moduls). Beobachte im Faltstatus, wie VIM die verschachtelten Faltungen darstellt und wie sich `:set foldlevel=1` gegenüber `:set foldlevel=99` verhält.
3. Lege dir in deiner vimrc einen „marker-Profil"-Abschnitt an:

   ```
   " Faltprofil für Konfigurationsdateien
   set foldmethod=marker
   set foldmarker={{{,}}}
   set foldlevel=1
   ```

   Öffne danach eine passende Datei erneut und prüfe, ob die Faltungen so initialisiert werden, wie du es dir wünschst.

10.1.4. Syntaxbasierte Faltungen

Hier übernimmt VIM das Erzeugen der Faltungen automatisch auf Basis der Syntaxregeln, z. B. bei Funktionen, Klassen, Blöcken in C, Python, Lua oder Vimscript. Strukturelemente, die der Syntax-Parser erkennt, werden als Faltungen angeboten, ohne dass du sie selbst definieren musst.

Aktiviert wird diese Methode mit:

```
:set foldmethod=syntax
```

oder dauerhaft in deiner vimrc:

```
set foldmethod=syntax
set foldenable
```

Voraussetzungen:

- Syntax-Highlighting muss aktiv sein (`:syntax on` bzw. `syntax on` in der vimrc).
- Die betreffende Sprache muss eine Faltdefinition besitzen (liefert meist das jeweilige Syntax- oder ftplugin-Skript).

Typische Eigenschaften syntaxbasierter Folds:

- Funktionen, Klassen, Blöcke und Kommentare werden automatisch als Faltungen angeboten.
- Änderungen am Code (z. B. neue Funktionen) werden beim Neuzeichnen des Bildschirms oder nach `:syntax sync` erkannt.

- Die Tiefe der Faltungen lässt sich mit `:set foldnestmax=...` begrenzen.
- Mit `:set foldlevel=0` werden alle Folds geschlossen, mit höheren Werten entsprechend mehr Ebenen geöffnet.

Übungen

1. Öffne eine größere Quellcode-Datei (z. B. in C oder Python) und aktiviere syntaxbasierte Faltung mit:
   - `:set foldmethod=syntax`
   - `:syntax on`, falls noch nicht aktiv

   Klappe danach einzelne Bereiche mit z c zu und mit z o wieder auf.
2. Probiere verschiedene Start-Level aus:
   - `:set foldlevel=0`
   - `:set foldlevel=1`
   - `:set foldlevel=2`

   Beobachte, wie stark die Datei jeweils „zusammenklappt“ und welche Übersicht du bevorzugst.
3. Nutze z M, um alle Folds zu schließen, und z R, um alle wieder zu öffnen. Beurteile, in welchen Dateitypen syntaxbasierte Folds für dich wirklich hilfreich sind (z. B. Code vs. Konfigurationsdateien).

10.1.5. Einrückungsbasierte Faltungen

Bei einrückungsbasierten Faltungen leitet VIM die Folds ausschließlich aus der Einrückungstiefe der Zeilen ab. Jede Gruppe von Zeilen mit gleicher Einrückung und darunterliegenden, weiter eingerückten Zeilen wird als zusammengehöriger Block behandelt.

Aktiviert wird diese Methode mit:

```
:set foldmethod=indent
```

oder dauerhaft in deiner vimrc:

```
set foldmethod=indent
set foldenable
```

Besonders geeignet ist diese Methode für Formate, bei denen die Einrückung bereits die logische Struktur ausdrückt:

- YAML
- `Python`
- Markdown-Listen und gegliederte Überschriften
- strukturierte Konfigurationsdateien mit konsistenter Einrückung

Wichtige Zusammenhänge:

- Die Breite einer Einrückungsstufe wird durch Optionen wie `:set shiftwidth` und `:set tabstop` praktisch beeinflusst.
- Gemischte Verwendung von Tabs und Spaces kann zu „schiefen“ Faltungen führen, wenn die Einrückung optisch stimmt, intern aber nicht konsistent ist.
- Leere Zeilen trennen häufig Faltblöcke – je nach Dateityp und Formatierung.
- Mit `:set foldlevel=` kannst du steuern, wie viel bereits aufgeklappt ist (z. B. `:set foldlevel=1` für eine grobe Übersicht).

Typische Keybindings im Alltag:

- **z** **c** – aktuellen Fold schließen
- **z** **o** – aktuellen Fold öffnen
- **z** **M** – alle Folds schließen („Maximal zu“)
- **z** **R** – alle Folds öffnen („Reset“)

Übungen

1. Erstelle eine kleine YAML- oder Python-Datei mit mehreren Einrückungsebenen (z. B. verschachtelte Dictionaries/Listen oder Funktionen mit inneren Blöcken). Aktiviere dann:
   - `:set foldmethod=indent`
   - `:set foldenable`

   Klappe die Folds mit **z** **c** zu und mit **z** **o** wieder auf. Beobachte, welche Struktur VIM automatisch erkennt.
2. Probiere verschiedene Start-Level aus:
   - `:set foldlevel=0`
   - `:set foldlevel=1`
   - `:set foldlevel=2`

   Vergleiche, wie stark der Text jeweils „zusammenklappt“ und welche Einstellung dir als Standard am angenehmsten erscheint.
3. Erzeuge absichtlich uneinheitliche Einrückung (z. B. mal Tabs, mal Spaces, unterschiedliche Breiten) und beobachte, wie sich die Folds verhalten. Repariere die Datei anschließend mit einem geeigneten Tool oder Ex-Befehl (z. B.

`:\%s/\textbackslash t/ /g`) und überprüfe, ob die Faltungen danach stabiler sind.

4. Öffne eine bestehende `Markdown`-Datei mit Überschriften und Listen. Aktiviere `:set foldmethod=indent` und schau dir an, ob die Einrückung für eine sinnvolle Faltstruktur ausreicht oder ob du das Format eher mit `:set foldmethod=marker` oder `:set foldmethod=syntax` behandeln würdest. Begründe für dich, warum.

10.1.6. Manuelle Kontrolle der Faltmethoden

Auch wenn VIM Faltungen automatisch erzeugen kann, ist es wichtig, die Mechanik jederzeit „per Hand“ kontrollieren zu können.

Die aktuelle Faltmethode lässt sich so anzeigen:

```
:set foldmethod?
```

Typische Werte sind z. B.:

- `manual` – nur manuell erzeugte Folds
- `marker` – Folds anhand von Markern (/)
- `syntax` – Folds anhand von Syntaxregeln
- `indent` – Folds anhand der Einrückungstiefe

Ob Folds überhaupt ausgewertet werden, steuert `:set foldenable` bzw. `:set nofoldenable`. Damit kannst du Faltungen global ein- oder ausschalten, ohne die eigentliche Faltmethode zu ändern.

Temporär alle Folds deaktivieren. Wenn du „einfach nur alles sehen“ willst:

```
:set nofoldenable
```

Damit bleiben bestehende Folds zwar definiert, werden aber nicht mehr zusammengeklappt angezeigt. Mit

```
:set foldenable
```

reaktivierst du die Faltung wieder.

Auf manuelle Faltung zurückschalten. Um jede automatische Faltlogik abzuschalten und nur noch bewusst gesetzte Folds zu verwenden:

```
:set foldmethod=manual
:set foldenable
```

Dies ist auch ein sinnvoller „Reset“, wenn `syntax`– oder `indent`–Folds unerwartete Ergebnisse liefern.

„Panikknöpfe“ im Alltag. Neben den `:set`-Optionen sind folgende Tastenfolgen in der Praxis wichtig:

- z R – alle Folds im aktuellen Buffer öffnen
- z M – alle Folds schließen („alles zu“)
- z o – aktuellen Fold öffnen
- z c – aktuellen Fold schließen
- z x – Folds neu berechnen (z. B. nach Änderungen)

Kombiniert mit `:set foldmethod?` und `:set foldenable` hast du damit eine komplette manuelle Kontrolle über das Faltverhalten.

Übungen

1. Öffne eine Datei mit bestehenden Folds (z. B. mit `marker`– oder `syntax`–Faltung). Prüfe die aktuelle Methode mit `:set foldmethod?` und spiele nacheinander:
   - `:set nofoldenable`
   - `:set foldenable`
   - `:set foldmethod=manual`

   Beobachte, was mit den vorhandenen Folds jeweils passiert.
2. Erzeuge in einer Testdatei ein paar manuelle Folds mit z f { Bewegung} (z. B. z f a p für einen Absatz). Öffne und schließe diese Folds mit z o, z c, z R und z M. Ziel: Die Grundtasten für Fold-Steuerung sollen „in Fleisch und Blut übergehen“.
3. Aktiviere `:set foldmethod=indent` in einer Datei mit Einrückungen, klappe alle Folds mit z M zu und öffne sie mit z R wieder. Schalte anschließend mit `:set foldmethod=manual` um und vergleiche das

Verhalten: Welche Variante passt besser zu deiner üblichen Arbeitsweise in dieser Datei?

10.2. Öffnen und Schließen

Nachdem Faltungen erstellt wurden, besteht der wichtigste Arbeitsschritt im gezielten Öffnen und Schließen einzelner oder mehrerer Folds. VIM bietet dazu eine kleine Gruppe von Befehlen, die wie Operatoren wirken und sich sehr gut in Makros und wiederholbare Arbeitsabläufe einfügen lassen.

10.2.1. Grundbefehle

Die drei zentralen Befehle zum Umgang mit der Faltung unter dem Cursor sind:

- **z** **o** – *open* Öffnet die Faltung unter dem Cursor um genau *eine* Ebene. Liegt eine verschachtelte Faltung vor, bleibt ein innerer Fold weiterhin zu, bis du ihn separat öffnest.
- **z** **c** – *close* Schließt die Faltung unter dem Cursor um eine Ebene. Befindest du dich innerhalb mehrerer ineinander verschachtelter Folds, wird zunächst nur die innerste Ebene geschlossen.
- **z** **a** – *toggle* Schaltet den Zustand der Faltung unter dem Cursor um: Ist sie geschlossen, öffnet **z** **a** sie; ist sie offen, schließt **z** **a** sie wieder.

Diese drei Kommandos sind die essenziellen „Fold-Operatoren“ im täglichen Arbeiten: Du bewegst den Cursor auf eine Faltung und steuerst mit **z** **o**, **z** **c** oder **z** **a**, was sichtbar sein soll – ohne den eigentlichen Text verändern zu müssen.

Zählervarianten. Wie viele andere VIM-Befehle akzeptieren auch die Fold-Kommandos einen numerischen Präfix:

- **2** **z** **o** öffnet zwei Faltungsebenen unterhalb des Cursors.
- **3** **z** **c** schließt drei Ebenen (z. B. bei tief verschachtelten Folds).

In der Praxis ist **z** **o** ohne Zähler oft ausreichend. Zählervarianten werden vor allem dann interessant, wenn du in fremden oder stark verschachtelten Dateien arbeitest und mehrere Ebenen auf einmal öffnen oder schließen willst.

Übungen

1. Erzeuge in einer Testdatei mehrere manuelle Folds (z. B. mit z f a p für ganze Absätze). Bewege den Cursor nacheinander auf verschiedene Folds und experimentiere mit:
   - z o
   - z c
   - z a

   Ziel: Du sollst „im Gefühl“ haben, wie sich der sichtbare Ausschnitt verändert, ohne den Text selbst zu verändern.
2. Lege verschachtelte Folds an (z. B. ein Block mit mehreren Unterblöcken). Öffne zunächst nur die äußere Faltung mit z o und beobachte, dass die inneren Folds geschlossen bleiben. Schließe dann mit z c wieder und probiere anschließend 2 z o, um mehrere Ebenen in einem Schritt zu öffnen.
3. Suche dir eine größere Konfigurationsdatei oder ein Skript und aktiviere eine automatische Faltmethode (z. B. `:set foldmethod=marker` oder `:set foldmethod=syntax`). Navigiere mit Bewegungen (j, k, g g, G) durch die Datei und steuere die Sicht mit z o, z c und z a. Ziel: Du arbeitest hauptsächlich über Folds und nicht mehr über reines Scrollen.

10.2.2. Alle Ebenen öffnen und schließen

Neben den „einfachen“ Befehlen z o, z c und z a gibt es zwei Varianten, die immer *alle* Ebenen eines Folds auf einmal bearbeiten:

- z O – öffnet *alle* Faltungs-Ebenen unter dem Cursor (rekursiv, vollständig). Auch tief verschachtelte Folds werden komplett aufgeklappt.
- z C – schließt *alle* Faltungs-Ebenen unter dem Cursor (rekursiv, vollständig). Der gesamte Block wird zu einem einzigen Fold „eingeklappt“.

Wichtiger Unterschied zu z o und z c:

- z o / z c arbeiten *ebenenweise*: pro Tastendruck wird genau eine Faltungsebene geöffnet bzw. geschlossen.
- z O / z C arbeiten *rekursiv*: alle inneren Folds unterhalb der Cursorposition werden in einem Schritt geöffnet bzw. geschlossen.

Typische Anwendungsfälle:

- große Funktionsblöcke mit verschachtelten Unterfunktionen,
- umfangreiche Konfigurationsdateien mit vielen Unterabschnitten,
- Skripte mit stark geschachtelten Kontrollstrukturen.

Mit z O verschaffst du dir schnell einen vollständigen Überblick über den Inhalt eines Folds, während z C denselben Bereich wieder auf ein kompaktes „Paket“ reduziert.

Übungen

1. Erzeuge in einer Testdatei mehrere verschachtelte Folds (z. B. einen äußeren Block mit mehreren inneren Abschnitten). Probiere nacheinander:
   - z o und z c
   - z O und z C

   Beobachte den Unterschied: Was bleibt bei z o noch gefaltet, was ist nach z O komplett offen?
2. Öffne in einer bestehenden Projektdatei (z. B. Quellcode oder Konfiguration) eine automatisch erzeugte Faltung (z. B. mit `:set foldmethod=syntax` oder `:set foldmethod=marker`). Nutze z O, um den kompletten Block sichtbar zu machen, editiere eine Stelle tief im Inneren und schließe den Block anschließend mit z C wieder.
3. Kombiniere Navigation und Folds: Springe mit Bewegungen (g g, G, { / }) zwischen größeren Blöcken und verwende jeweils z O, um kurzzeitig alles zu sehen, und z C, um die Datei wieder auf „Überflug-Niveau“ zu bringen.

10.2.3. Globale Fold-Operationen

Neben den lokalen Befehlen z o, z c, z O und z C gibt es vier „globale“ Fold-Kommandos, die das Faltungsniveau der *gesamten* Datei steuern. Sie arbeiten über das sogenannte `foldlevel`: Je höher der Level, desto mehr Folds sind geöffnet.

- z r – *reduce* Reduziert das Faltungslevel: global wird *eine* Ebene weiter geöffnet. Praktisch: „Zeig mir ein bisschen mehr von allen Folds.“

- z m – *more* Erhöht das Faltungslevel: global wird *eine* Ebene weiter geschlossen. Praktisch: „Klappt alles eine Stufe kompakter."
- z R – *reduce fully* Öffnet *alle* Folds in der gesamten Datei vollständig – egal, wie tief sie verschachtelt sind.
- z M – *mask fully* Schließt *alle* Folds in der gesamten Datei vollständig – maximal komprimierte Ansicht.

Damit ergeben sich zwei „Achsen" für Folds:

- z o, z c, z O, z C arbeiten lokal am Fold unter dem Cursor (Feinsteuerung).
- z r, z m, z R, z M arbeiten global auf der ganzen Datei (Grobsicht / Zoomstufe).

Typische Einsatzmuster:

- z M am Anfang, um aus einem riesigen File nur die Topstruktur zu sehen.
- Mehrfach z r, um schrittweise „tiefer in die Datei hineinzuzoomen".
- z R, um kurzzeitig alles sichtbar zu haben – danach mit z M oder lokalen Befehlen wieder aufräumen.

Übungen

1. Öffne eine größere Datei mit bereits vorhandenen Folds (z. B. durch `:set foldmethod=syntax` oder `:set foldmethod=marker`). Probiere nacheinander:
   - z M – alles maximal zuklappen,
   - mehrmals z r – stufenweise öffnen,
   - z R – alles vollständig öffnen.

   Beobachte, wie sich die „Zoomstufe" der Datei ändert.
2. Kombiniere globale und lokale Befehle:
   a) Klappe mit z M erst alles zu.
   b) Nutze Bewegungen (g g, G, { / }) um zu einem interessanten Block zu springen.
   c) Öffne nur dort lokal mit z O oder z o.
   d) Stelle den globalen Überblick wieder her mit z M oder einigen z m.
3. In einem Projekt mit vielen Dateien (z. B. Codebasis oder LaTeX-Projekt): Nutze in jeder Datei zunächst z M, um nur die Top-Level-Struktur zu sehen. Entscheide dann anhand der Faltungsübersicht, welche Bereiche du mit

lokalen Befehlen weiter aufklappst. Ziel: Folds als „Outliner“ für Navigation benutzen, nicht nur als optischen Gimmick.

10.2.4. Hilfreiche Sichtkontrolle

Die aktuelle Faltebene (Fold-Level) der einzelnen Zeilen lässt sich über eine zusätzliche Spalte am linken Rand sichtbar machen — die `foldcolumn`. Damit wird sofort klar, wo Folds beginnen, wie tief sie verschachtelt sind und ob eine Zeile innerhalb oder außerhalb eines Folds liegt.

Die Einstellung erfolgt über die Option `foldcolumn`:

- `:set foldcolumn=0` keine Fold-Spalte (Standard in vielen Setups)
- `:set foldcolumn=1` eine Spalte für Folds
- `:set foldcolumn=2` zwei Spalten (gut bei tiefen Verschachtelungen)

Allgemein gilt:

- `:set foldcolumn?` zeigt den aktuellen Wert an.
- Werte `1`–`3` reichen in der Praxis fast immer aus.
- Hohe Werte verschwenden nur Platz und machen die Darstellung unruhig.

Die Fold-Spalte erscheint links vom Text (typisch links von Zeilennummern/Signspalte) und zeigt je nach Zeile z. B.:

- Symbole für „Fold beginnt hier“,
- Symbole für „Zeile gehört zu einem Fold“,
- Symbole für „Fold endet hier“.

In Kombination mit den globalen Kommandos **z M**, **z R**, **z r**, **z m** entsteht damit eine Art „Mini-Outline“ am Rand: Man sieht nicht nur, *dass* Folds existieren, sondern auch, wie tief die jeweilige Struktur reicht.

Übungen

1. Aktiviere Folds in einer größeren Datei, z. B. mit `:set foldmethod=syntax` oder `:set foldmethod=marker`. Schalte dann nacheinander:
   - `:set foldcolumn=0`
   - `:set foldcolumn=1`
   - `:set foldcolumn=2`

   Beobachte, wie sich die Darstellung in der linken Spalte ändert.

2. Nutze z M, um alle Folds zu schließen, und laufe mit j / k durch die Datei. Achte gezielt auf die Fold-Spalte: Wo beginnen Folds? Wo enden sie? Welche Bereiche sind gar nicht gefaltet?
3. Stelle eine für dich angenehme Kombination ein, z. B.:
   - `:set number`
   - `:set foldcolumn=1`

   und arbeite eine Weile in diesem Layout. Ziel: herausfinden, ob dir die visuelle „Outline“ hilft oder ob du die Fold-Spalte nur bei Bedarf zuschaltest.

10.2.5. Makrotauglichkeit

Die Faltbefehle z o, z c, z a und insbesondere z O / z C sind sehr gut für Makros und automatisierte Abläufe geeignet, weil sie sich „brav“ verhalten:

- sie scrollen den Text nicht „wild“ durch die Gegend,
- sie ändern nur den Sichtbarkeitszustand des aktuellen Folds,
- sie hängen nicht von absoluten Bildschirmpositionen ab,
- sie verhalten sich bei gleichem Fold-Zustand immer gleich (deterministisch).

Dadurch lassen sie sich stabil mit Bereichen, Textobjekten, globalen Kommandos und `:normal!`-Sequenzen kombinieren. Typische Muster:

- `:g/\^\#\#\# / normal! zc` schließt alle Folds, deren Überschrift mit `###` beginnt (z. B. Markdown-Überschriften).
- `:\%g/\^class / normal! zO` öffnet für jede Klassendefinition im gesamten Puffer den zugehörigen Fold vollständig.
- `:10,80g/\^def / normal! za` toggelt Folds nur im Bereich Zeile 10–80 für alle Funktionsdefinitionen.

In Makros ist besonders wichtig:

- Vor der Aufzeichnung möglichst einen definierten Ausgangszustand herstellen (z. B. mit z M alles schließen, mit z R alles öffnen).
- Innerhalb des Makros nur „relative“ Aktionen verwenden (z o, z c, z a, z O, z C) und auf spontane Scrollbefehle verzichten.
- Den Fold-Zustand bewusst als Teil des „Layouts“ betrachten: Ein Makro sollte mit demselben Fold-Zustand immer dieselben Ergebnisse liefern.

Übungen

1. Nimm ein Makro [q][f] auf, das auf der aktuellen Zeile [z][a] ausführt und dann mit [j] zur nächsten Zeile springt. Wende das Makro mit [@][@] oder [1][0][@][f] auf mehrere Zeilen an. Ziel: gezieltes Umschalten einzelner Folds in einer Liste von Überschriften.
2. Erzeuge in einer Datei mehrere logisch zusammenhängende Blöcke (z. B. durch Überschriften oder Kommentare) und baue zuerst Folds dafür (manuell, per Marker oder Syntax). Führe dann nacheinander aus:
   - [z][M] (alles schließen)
   - [z][R] (alles öffnen)
   - [z][r] / [z][m] (Fold-Level schrittweise anpassen)

   Beobachte, wie sich der Fold-Zustand reproduzierbar verhält und wie gut sich daraus Makros ableiten lassen.
3. Kombiniere globale Kommandos mit Folds: Erzeuge z. B. eine Datei mit mehreren Funktionsköpfen `def` ... und führe dann
   - `:\%g/\^def / normal! zc`
   - `:\%g/\^def / normal! zo`

   aus. Ziel: verstehen, wie `:g` und `:normal!` zusammen mit Faltbefehlen deterministische „Batch-Faltung“ ermöglichen.

10.3. Faltungstiefe

Die Faltungstiefe beschreibt, wie viele Ebenen von Folds gleichzeitig angezeigt oder verborgen werden. VIM verwaltet diese Tiefe über ein Nummernsystem, das sowohl global (für das aktuelle Fenster) als auch pufferbezogen wirkt. Das Konzept ist für große Dateien entscheidend, um flexibel zwischen grober Übersicht und Detailarbeit wechseln zu können.

10.3.1. Foldlevel: zentrale Steuerung

Der aktuelle Faltungslevel wird über die Option `foldlevel` gesteuert:

```
:set foldlevel=0
```

Typische Werte:

- `foldlevel=0` Alles, was gefaltet werden kann, ist zu. Man sieht nur die äußerste Struktur, z. B. Kapitelüberschriften oder Top-Level-Funktionen.
- `foldlevel=1` Nur die oberste Ebene ist aufgeklappt, darunterliegende Folds bleiben geschlossen. Gut geeignet, um sich einen Überblick über die Hauptblöcke zu verschaffen.
- `foldlevel=2` Zwei Ebenen sind sichtbar, tiefere Folds bleiben zu. Praktisch für verschachtelte Code-Strukturen oder Kapitel/Abschnitt-Hierarchien.
- `foldlevel=99` Praktisch alles geöffnet. Nur explizit „hart“ geschlossene Folds bleiben zu (z. B. durch **z** **c** oder spezielle Methoden).

Die aktuelle Einstellung kann jederzeit abgefragt werden:

```
:set foldlevel?
```

und auch nur für das aktuelle Fenster gesetzt werden:

```
:setlocal foldlevel=1
```

Damit lässt sich derselbe Puffer in verschiedenen Fenstern mit unterschiedlichen Faltungstiefen betrachten – z. B. links nur Top-Level-Struktur, rechts Detailansicht.

Zusammenspiel mit **z** **r** und **z** **m**

Die Normal-Mode-Befehle **z** **r** und **z** **m** arbeiten direkt auf `foldlevel`:

- **z** **r** erhöht `foldlevel` schrittweise – es werden also mehr Ebenen sichtbar („Faltungen reduzieren“).
- **z** **m** verringert `foldlevel` – es werden wieder mehr Ebenen geschlossen.
- **z** **R** setzt `foldlevel` effektiv sehr hoch (alles öffnen).
- **z** **M** setzt `foldlevel` effektiv auf 0 (alles schließen).

Man kann sich merken: **z** **r** / **z** **m** *sind die „dynamische Bedienoberfläche“ zu* `foldlevel`. Direkte `:set`-Kommandos legen einen festen Zustand fest, **z** **r** / **z** **m** verändern ihn interaktiv.

Übungen

1. Öffne eine Datei mit mehreren verschachtelten Folds (z. B. Funktionen in einer Programmiersprache oder Abschnitte in einem LaTeX-Dokument). Probiere nacheinander:

   - `:set foldlevel=0`
   - `:set foldlevel=1`
   - `:set foldlevel=2`
   - `:set foldlevel=99`

   Beobachte jeweils, wie sich die sichtbare Struktur verändert.
2. Stelle `:set foldlevel=1` ein und verwende dann nur Normal-Mode-Befehle:
   - drücke mehrfach z r
   - anschließend mehrfach z m

   Vergleiche das Verhalten mit den direkten `:set`-Befehlen. Ziel: Das Gefühl entwickeln, dass z r / z m nichts „Magisches“ sind, sondern `foldlevel` nur schrittweise anpassen.
3. Teile das aktuelle Fenster mit `:vsplit` und zeige denselben Puffer in beiden Fenstern. Setze links:
   - `:setlocal foldlevel=1`

   und rechts:
   - `:setlocal foldlevel=3`

   Navigiere nun in beiden Fenstern parallel. Ziel: Verstehen, dass Faltungstiefe fensterlokal ist und du denselben Inhalt gleichzeitig als Übersicht *und* als Detailansicht sehen kannst.

10.3.2. Dynamische Steuerung

Neben der direkten Vorgabe eines festen Werts für `foldlevel` kann die Faltungstiefe auch inkrementell verändert werden:

```
:set foldlevel+=1
:set foldlevel-=1
```

Damit ist ein schrittweises „Rein- oder Herauszoomen“ möglich, ohne den aktuellen Wert genau zu kennen:

- `:set foldlevel+=1` öffnet eine zusätzliche Faltungsebene in der gesamten Datei (entspricht mehrfach z r).
- `:set foldlevel-=1` schließt eine Faltungsebene global (entspricht mehrfach z m).

Der aktuelle Wert lässt sich jederzeit prüfen mit:

```
:set foldlevel?
```

Diese Kommandos sind besonders nützlich in:

- eigenen Mappings (z. B. `:nnoremap <leader>zo :set foldlevel+=1<CR>`),
- Projekt-spezifischen Einstellungen,
- Skripten oder `autocmd`-Blöcken (z. B. für bestimmte Dateitypen).

Übungen

1. Öffne eine Datei mit mehreren Faltungsebenen und setze zunächst:
   - `:set foldlevel=0`

   Erhöhe dann schrittweise die Faltungstiefe mit:
   - `:set foldlevel+=1` (mehrfach ausführen)

   Beobachte, ab welcher Tiefe die für dich „richtige“ Übersicht entsteht.
2. Setze `:set foldlevel=3` und reduziere anschließend die Tiefe in Einzelschritten mit:
   - `:set foldlevel-=1`

   Vergleiche dieses Verhalten mit [z][m] und [z][M]. Ziel: Den Unterschied zwischen „hart alles zu“ ([z][M]) und feinstufigem Reduzieren der Sichtbarkeit verstehen.
3. Lege dir testweise Mappings an (z. B. in einer temporären Konfiguration):
   - `:nnoremap <leader>+ :set foldlevel+=1<CR>`
   - `:nnoremap <leader>- :set foldlevel-=1<CR>`

   Nutze anschließend <leader>+ und <leader>-, um in einer großen Datei interaktiv hinein- und herauszuzoomen. Beurteile, ob dir diese Art von „Zoomsteuerung“ im Alltag hilft.

10.3.3. Befehle zum Öffnen/Schließen aller Folds

Neben der direkten Steuerung über `:set foldlevel=...` bietet VIM Normal-Mode-Kommandos, die intern genau diese Option verändern:

- [z][r] — *reduce folding*
  Erhöht `'foldlevel'` um eins. Ergebnis: Mehr Folds werden geöffnet, die Darstellung wird „flacher“.

- z m — *more folding*
  Verringert `'foldlevel'` um eins. Ergebnis: Mehr Folds werden geschlossen, die Darstellung wird kompakter.
- z R — *reduce fully*
  Setzt `'foldlevel'` so hoch, dass alle Folds geöffnet sind (entspricht praktisch „alles aufklappen“).
- z M — *more fully*
  Setzt `'foldlevel'` auf 0 – alle Folds werden vollständig geschlossen.

Man kann sich merken:

- z r und z R öffnen mehr,
- z m und z M schließen mehr.

Übungen

1. Öffne eine Datei mit mehreren Faltungen (z. B. mit `::set foldmethod=indent` in einem Python- oder YAML-File). Prüfe zuerst den aktuellen Wert mit:
   - `:set foldlevel?`

   Drücke dann mehrmals z m und anschließend z r und beobachte jeweils den neuen Wert von `'foldlevel'`.
2. Führe nacheinander aus:
   - z M
   - z R

   und beobachte, wie sich die Darstellung verändert. Überprüfe mit `:set foldlevel?`, welcher Wert intern gesetzt wurde.
3. Lege dir ein kleines Testskript an und spiele das gleiche Szenario einmal mit expliziten Ex-Befehlen durch:
   - `:set foldlevel=0`
   - `:set foldlevel=1`
   - `:set foldlevel=99`

   Vergleiche das Verhalten mit den Tastenkombinationen z m, z r, z M, z R.

10.3.4. Initiale Faltstruktur beim Laden einer Datei

Häufig soll eine Datei beim Öffnen nicht komplett zu- oder aufgeklappt sein, sondern mit einer „sinnvollen“ Faltungstiefe starten. Dafür kombinierst du die Faltmethode mit einem Startlevel:

```
set foldmethod=syntax
set foldlevelstart=1
```

- `'foldmethod'` bestimmt, *wie* Folds erzeugt werden (z.B. `manual`, `indent`, `marker`, `syntax`).
- `'foldlevelstart'` legt fest, mit welchem `'foldlevel'` ein neuer Buffer startet. Der Defaultwert `-1` bedeutet: „verwende den aktuellen `'foldlevel'` unverändert“.

Typische Szenarien:

- `:set foldmethod=syntax` / `:set foldlevelstart=1` Funktionen oder Top-Level-Strukturen sind sichtbar, alles darunter ist gefaltet.
- `:set foldmethod=indent` / `:set foldlevelstart=2` Für YAML/Python: grobe Blöcke sichtbar, tiefe Details zunächst eingeklappt.
- `:set foldmethod=manual` / `:set foldlevelstart=99` Manuelle Folds bleiben grundsätzlich offen; du klappst nur bei Bedarf zu.

Übungen

1. Öffne eine typische „Code-Datei“ (z. B. Python, C oder ein Vimscript) und setze:
   - `:set foldmethod=syntax`
   - `:set foldlevelstart=1`

   Schließe die Datei mit `:q` und öffne sie erneut. Beobachte, welche Ebenen sichtbar sind und prüfe den aktuellen Wert mit `:set foldlevel?`.
2. Ändere `:set foldlevelstart=2` und wiederhole den Versuch. Vergleiche die Darstellung mit dem vorherigen Zustand (insbesondere bei verschachtelten Funktionen / Klassen).
3. Übernimm eine deiner bevorzugten Kombinationen dauerhaft in deine `vimrc`, z. B.:

   ```
   " Faltung für Quellcode-Dateien
   augroup MyFolds
   ```

```
    autocmd!
    autocmd FileType python,cpp,java,vim
          \ setlocal foldmethod=syntax foldlevelstart=1
augroup END
```

Prüfe anschließend an mehreren Dateien, ob VIM beim Laden erwartungsgemäß mit der gewünschten Faltstruktur startet.

10.3.5. Makros und Faltungstiefe

Makros sollen nicht daran scheitern, dass Teile der Datei gerade zugeklappt sind. Besonders Makros, die zeilenweise arbeiten (j, k, 0, $, Operatoren wie d, c, y usw.), können durch Folds „aus dem Tritt“ geraten, wenn Zeilen verborgen sind.

Zwei robuste Strategien:

- **Vorher alles öffnen:** z R vor dem Abspielen des Makros sorgt dafür, dass keine Zeile hinter einem Fold verschwindet.
- **Faltlevel großzügig setzen:** `:setlocal foldlevel=99` im aktuellen Buffer sorgt dafür, dass effektiv alle Folds geöffnet werden, ohne die Methodik (`:foldmethod`) zu ändern.

Damit wird verhindert, dass der Cursor in gefalteten Blöcken „hängen“ bleibt, unerwartet springt oder ganze Bereiche vom Makro gar nicht erst erreicht werden.

Praktische Muster

- **Ad-hoc-Ansatz für interaktive Arbeit:**
  1. Datei öffnen, gewünschte Faltung einstellen.
  2. Vor dem Makro: z R.
  3. Makro mit @ *Register* abspielen.
  4. Optional danach mit z M wieder alles schließen.
- **Buffer-spezifische Vorbereitung:** In Dateien, in denen du *immer* mit Folds arbeitest und trotzdem Makros zuverlässig laufen sollen:

  ```
  setlocal foldlevel=99
  ```

  Damit bleiben Folds „konzeptuell“ vorhanden, sind aber standardmäßig geöffnet.

Übungen

1. Erzeuge testweise mehrere Folds (z.B. mit z f a p oder über `:set foldmethod=marker`) und klappe sie mit z M ein. Nimm dann ein Makro auf, das jede Zeile nummeriert oder am Zeilenende ein Zeichen anhängt. *Beobachtung:* Was passiert, wenn du das Makro mit zugeklappten Folds abspielst?
2. Wiederhole denselben Versuch, öffne aber vorher mit z R alle Folds. Vergleiche das Verhalten des Makros mit dem ersten Durchlauf.
3. Setze für einen Testbuffer `:setlocal foldlevel=99` und arbeite mit deinen üblichen Faltmethoden weiter. Prüfe, ob deine Makros jetzt konsistenter laufen, obwohl du Folds angelegt hast.

10.4. Faltungen aktualisieren

Während der Bearbeitung einer Datei ändern sich laufend Textstrukturen, Kommentare oder Einrückungen. Damit VIM die Faltungen korrekt aktualisiert, müssen bestehende Fold-Informationen gelegentlich neu ausgewertet oder bewusst zurückgesetzt werden.

VIM unterscheidet dabei zwei Aspekte:

- *Neuberechnung der Folds im aktuellen Buffer* (z.B. bei `:foldmethod=syntax` oder `:foldmethod=indent`).
- *Speichern und Wiederherstellen* einer bestimmten Ansicht inklusive Faltungen (`:mkview`/`:loadview`).

10.4.1. Folds im aktuellen Buffer neu berechnen

Für die reine Neuberechnung der Faltungen gibt es zwei zentrale Befehle:

- z x Berechnet alle Folds im aktuellen Fenster neu, basierend auf der aktuellen `:foldmethod`. Man kann sich das als „Faltungs-Refresh“ vorstellen.
- z X Löscht alle *manuellen* Folds im aktuellen Fenster und berechnet anschließend die Faltungen neu. Das ist vor allem nützlich, wenn man experimentell viele z f-Folds gesetzt hat und wieder auf reine Syntax-/Indent-Folds zurück will.

Typische Einsatzfälle:

- Du hast größere Bereiche umstrukturiert, eingerückt oder auskommentiert und die Faltungen wirken „komisch“ → z x ausführen.
- Du hast testweise manuelle Folds angelegt, möchtest aber zur automatischen Faltung zurück → z X ausführen und dann ggf. erneut mit `:set foldmethod=syntax` oder `:set foldmethod=indent` arbeiten.

Übungen

1. Erzeuge in einer Datei mehrere manuelle Folds mit z f a p und z f 3 j. Strukturiere den Text danach um (Zeilen einfügen, löschen, Kommentare ergänzen) und führe z x aus. Beobachte, wie sich die Faltgrenzen anpassen.
2. Lege erneut manuelle Folds an und entferne sie dann vollständig mit z X. Prüfe mit z o / z c, ob wirklich keine manuellen Folds mehr übrig sind.
3. Stelle `:set foldmethod=indent` in einer eingerückten Datei (z.B. Python, YAML) ein, ändere die Einrückung und führe danach z x aus. Ziel: Verstehen, dass die Faltung an der Logik der Einrückung hängt und aktualisiert werden muss.

10.4.2. Fold-Ansicht speichern und laden (`:mkview`/`:loadview`)

Neben der reinen Neuberechnung der Folds kann VIM auch eine komplette „Ansicht“ (view) einer Datei sichern – inklusive:

- Cursorposition,
- Fensterlayout,
- lokalen Optionen,
- Faltzustand (welche Folds offen/geschlossen sind).

Dafür dienen `:mkview` und `:loadview`:

```
:mkview
:loadview
```

- `:mkview` speichert die aktuelle Ansicht der Datei (standardmäßig in einer View-Datei unterhalb von `/.vim/view`).
- `:loadview` stellt diese Ansicht beim nächsten Öffnen der Datei wieder her – Faltungen inklusive.

Das ist besonders hilfreich, wenn du:

- an großen Dateien über mehrere Sitzungen arbeitest,
- dir eine bestimmte Faltstruktur „merken“ willst,
- in Projekten mit wiederkehrenden Navigationsmustern arbeitest (z.B. immer nur bestimmte Kapitel eines langen Dokuments offen haben möchtest).

Übungen

1. Öffne eine große Datei, lege einige Folds an (manuell oder per `:set foldmethod=syntax`) und bringe die Ansicht in einen für dich angenehmen Zustand (einige Folds offen, andere zu). Führe dann `:mkview` aus, beende VIM und öffne die Datei erneut. Lade die Ansicht mit `:loadview` und überprüfe, ob die Faltzustände wiederhergestellt wurden.
2. Ändere nach einem `:loadview` gezielt einige Folds (zusätzliche Bereiche auf- oder zuklappen) und rufe erneut `:mkview` auf. Vergleiche das Verhalten beim nächsten `:loadview` mit der vorherigen Version.
3. Experimentiere in einem Testverzeichnis mit mehreren Dateien: Erzeuge in zwei verschiedenen Dateien unterschiedliche Faltstrukturen, speichere jeweils mit `:mkview` und prüfe, dass `:loadview` je Datei die passende Ansicht lädt.

10.4.3. Alle Folds neu erzeugen

Gerade bei `:set foldmethod=syntax` oder `:set foldmethod=indent` kann es passieren, dass die Faltungen „aus dem Tritt“ geraten: Du änderst größere Blöcke, kommentierst viel aus oder ein, verschiebst Funktionen – und die Folds passen nicht mehr sauber zur Struktur.

In solchen Fällen hilft es, die zugrunde liegenden Informationen vollständig neu auszuwerten. Ein robuster Workflow ist:

```
:syntax sync fromstart
:set foldmethod=syntax
```

Damit erzielst du:

- eine frische Auswertung aller Syntax-Regeln ab Dateianfang,
- ein sauberes Setzen der Faltungsgrenzen gemäß `:foldmethod`,

- eine vollständige Aktualisierung der Folds im aktuellen Fenster.

Statt `:set foldmethod=syntax` kannst du natürlich auch jede andere Faltmethode einsetzen, z. B. `:set foldmethod=indent` oder eine `:expr`-basierte Lösung. Entscheidend ist, dass du nach größeren Umbauten der Datei eine *konsistente* Faltbasis hast.

Übungen

1. Aktiviere in einer Quellcode-Datei `:set foldmethod=syntax` und klappe einige Bereiche mit z c und z M zu. Nimm anschließend umfangreiche Änderungen vor (z.B. mehrere Funktionen verschieben oder auskommentieren) und führe dann nacheinander
   - `:syntax sync fromstart`
   - z x

   aus. Beobachte, wie sich die Faltgrenzen an die neue Struktur anpassen.
2. Wiederhole das Experiment mit `:set foldmethod=indent` in einer Datei mit sauberer Einrückung (z.B. Python, YAML). Ändere Einrückung und Blockstruktur und erzeug danach mit z x alle Folds neu. Ziel: Verstehen, dass *Einrückung* bei dieser Methode die Faltungslogik bestimmt.
3. Kombiniere diesen „Neuaufbau“ der Folds mit `:mkview` und `:loadview`: Erzeuge zunächst eine sinnvolle Faltstruktur, speichere sie mit `:mkview`, zerstöre sie bewusst (z.B. mit z R / z M), baue sie mit `:syntax sync fromstart` und z x neu auf und vergleiche anschließend das Verhalten von `:loadview`. So erkennst du den Unterschied zwischen „Folds neu berechnen“ und „Ansicht wiederherstellen“.

10.4.4. Faltungen hart zurücksetzen

Manchmal reicht ein einfaches z x nicht mehr aus, weil sich über die Zeit verschiedene Faltmethoden, manuelle Folds und Syntax-Folds ineinander verhakt haben. In solchen Fällen hilft ein bewusster „Hard-Reset“ der Faltlogik.

Ein robuster Ablauf ist:

```
:set foldmethod=manual
:set foldmethod=syntax
```

Damit erreichst du:

- Alle bestehenden Folds werden verworfen (`:foldmethod=manual` kennt keine automatische Faltlogik).
- Beim Zurückschalten auf `:foldmethod=syntax` (oder eine andere Methode) baut VIM die Faltungen *komplett* neu auf.
- Alte, inkonsistente Fold-Grenzen oder Überbleibsel von früheren Methoden verschwinden.

Optional kannst du danach noch einmal explizit synchronisieren und aktualisieren:

```
:syntax sync fromstart
```
z x

So stellst du sicher, dass sowohl Syntax-Information als auch Folds frisch und konsistent sind.

Übungen

1. Aktiviere in einer Quellcode-Datei `:set foldmethod=syntax` und erzeuge einige Folds (z. B. mit z c, z M). Schalte dann bewusst zwischen verschiedenen Methoden um:
   - `:set foldmethod=indent`
   - `:set foldmethod=manual`
   - `:set foldmethod=syntax`

   Beobachte, ab wann die Faltungen „schief“ wirken. Führe anschließend den Hard-Reset mit `:set foldmethod=manual` und `:set foldmethod=syntax` aus und vergleiche das Ergebnis.
2. Erzeuge in einer Datei gemischte Faltungen:
   - einige manuell mit z f
   - einige über Marker (`:set foldmethod=marker`)

   Schalte danach zurück auf `:set foldmethod=syntax` und nutze den Hard-Reset. Ziel: Erkennen, dass manuelle und markerbasierte Folds zuverlässig verschwinden und die Syntax-Folds „sauber“ neu gesetzt werden.
3. Baue dir eine kleine „Rettungs-Sequence“ als Ex-Kommando, z. B.:
   - `:nnoremap <leader>F :set foldmethod=manual \textbackslash | set foldmethod=syntax \textbackslash | syntax sync fromstart \textbackslash | normal! zx<CR>`

Teste dieses Mapping in einer großen Datei mit vielen Folds und nutze es immer dann, wenn die Faltungen sichtbar inkonsistent werden.

10.4.5. Faltung nach Änderungen im Visual Mode

Nach umfangreichen Änderungen im *Visual Mode* (z. B. mit [v], [V] oder [Ctrl]-[v]) kann die bestehende Faltstruktur aus dem Tritt geraten: Einrückungen ändern sich, ganze Blöcke werden verschoben oder gelöscht, aber die Folds orientieren sich noch an der alten Struktur.

Für genau diesen Fall gibt es den „harten Refresh“ der aktuellen Faltmethode:

[z][X]

[z][X] erzwingt eine komplette Neuberechnung aller Faltungen basierend auf der aktuell gesetzten `:foldmethod`, ohne die Methode selbst zu ändern. Im Unterschied zu [z][x] (lokale Aktualisierung) werden hier alle Folds im Puffer neu aufgebaut.

Typische Situationen für [z][X]:

- Du hast mit [Ctrl]-[v] ganze Spalten verschoben oder eingerückt und die Folds decken sich nicht mehr mit den logischen Blöcken.
- Du hast große Bereiche im Visual Mode gelöscht oder kommentiert und VIM zeigt Folds an Stellen, an denen gar nichts mehr steht.
- Nach Makro-Läufen im Visual Mode, bei denen viele Zeilen gleichzeitig verändert wurden (z. B. Einrückung oder Einfügen von Kommentarzeichen).

Übungen

1. Öffne eine Datei mit eingerückten Blöcken (z. B. Python oder ein eingerücktes Konfigurationsfile) und aktiviere einrückungsbasierte Faltung:
   - `:set foldmethod=indent`

   Falte einige Bereiche mit [z][c] und [z][M]. Markiere anschließend mit [Ctrl]-[v] einen Block und verschiebe ihn mit [>][>]. Nutze zuerst [z][x] und dann [z][X] und vergleiche, ab wann die Folds wieder zur Struktur passen.
2. Nimm eine Datei mit Funktionsblöcken (z. B. C, Lua oder Vimscript) und aktiviere `:set foldmethod=syntax`. Erzeuge per Visual Mode ([V])

und Operatoren (d, >, <) kräftige Umbrüche in der Struktur. Prüfe danach mit z R und z M, wie „schief“ die Folds geworden sind, und setze sie dann mit z X sauber zurück.

3. Kombiniere z X mit einem Mapping, das du in deiner Konfiguration hinterlegst, z. B.:
   - `:nnoremap <leader>x zx\textbackslash {}|zX`

   Teste dieses Mapping in einer großen Datei mit vielen Folds, indem du zuerst im Visual Mode größere Blöcke umformatierst und anschließend per <leader>x die Faltstruktur bereinigen lässt.

10.4.6. Automatische Aktualisierung beim Speichern

Optional kann eine Auto-Kommandogruppe eingerichtet werden, die beim Speichern einer Datei automatisch alle Folds neu berechnet:

```
augroup foldrefresh
  autocmd!
  autocmd BufWritePost * silent! normal! zX
augroup END
```

Wirkung im Detail:

- `BufWritePost *` – nach jedem erfolgreichen Speichern (`:w`) einer beliebigen Datei (`*`).
- `silent!` – unterdrückt Meldungen, auch im Fehlerfall.
- `normal! zX` – führt z X im Normal Mode aus und erzwingt eine komplette Neuberechnung der Faltungen.

Vorteile:

- Faltungen sind nach jedem Speichern konsistent zum aktuellen Code.
- Keine „hängenden“ oder „geisterhaften“ Folds nach großen Umbauten, Einrückungsänderungen oder Refactorings.
- Sinnvoll bei Projekten mit komplexer Einrückungslogik oder stark verschachtelten Strukturen (z. B. große Codebasen, Konfigurationsbäume, LaTeX-Projekte).

Nachteile bzw. Dinge, die man im Blick behalten sollte:

- In sehr großen Dateien kann z X beim Speichern spürbar kosten – dann ggf. nur für bestimmte Dateitypen aktivieren.

- Die aktuelle Faltungssicht „gehört“ immer zum gespeicherten Stand; wer absichtlich mit „veralteten“ Folds arbeitet, wird hier ständig „aufräumen“.

Eine selektivere Variante beschränkt den Auto-Refresh z. B. auf Python- und Vimscript-Dateien:

```
augroup foldrefresh
  autocmd!
  autocmd BufWritePost *.py,*.vim silent! normal! zX
augroup END
```

Übungen

1. Richte die oben gezeigte `foldrefresh`-Gruppe ein und aktiviere `:set foldmethod=syntax` bzw. `:set foldmethod=indent` in einer größeren Datei. Nimm danach größere Umbauten im Visual Mode vor (z. B. Einrückungen ändern, Blöcke verschieben), speichere mit `:w` und beobachte, wie sich die Folds nach dem Auto-**z** **X** verhalten.
2. Passe die Auto-Kommandogruppe so an, dass sie nur für ein bestimmtes Projektverzeichnis gilt, z. B. mittels `autocmd BufWritePost /projekt/* ...`. Teste das Verhalten sowohl innerhalb dieses Verzeichnisses als auch in anderen Dateien.
3. Entferne die `foldrefresh`-Gruppe wieder mit:
   - `:augroup foldrefresh`
   - `:autocmd!`
   - `:augroup END`

   Vergleiche expliziten Einsatz von **z** **X** (manuell) mit der automatischen Variante und entscheide, welche Kontrolle du im Alltag bevorzugst.

10.5. Faltmethoden

VIM unterstützt mehrere Methoden, um Faltungen zu erzeugen. Jede Faltmethode (`foldmethod`) folgt einem eigenen Prinzip und eignet sich für unterschiedliche Anwendungsfälle. Die Wahl der richtigen Methode bestimmt maßgeblich, wie übersichtlich und stabil die Faltungen im Editor sind.

10.5.1. Manual (`manual`)

Bei der manuellen Methode werden Faltungen explizit durch den Benutzer gesetzt. VIM erzeugt oder entfernt dabei *keine* Folds von selbst, sondern reagiert nur auf direkte Befehle:

- **z** **f** <Motion> Erzeugt eine Faltung über den Bewegungsbereich. Beispiel: **z** **f** **a** **p** faltet den gesamten Absatz.
- **z** **f** [*n*] Faltet die aktuelle Zeile und die nächsten *n* Zeilen. Beispiel: **z** **f** **3** **j** faltet aktuelle Zeile plus drei weitere.
- `:'<,'>fold` oder **z** **f** im Visual Mode Faltet den markierten Bereich.
- **z** **d** Entfernt die Faltung unter dem Cursor.
- **z** **D** Entfernt alle Faltungen, die die aktuelle Zeile betreffen.
- **z** **E** Entfernt *alle* manuellen Folds in der aktuellen Datei.

Damit *manuelle* Folds verwendet werden, muss `foldmethod` entsprechend gesetzt sein:

```
:set foldmethod=manual
```

Typische Einsatzszenarien:

- lose strukturierte Notizen ohne feste Syntax,
- Log-Dateien oder Dump-Ausgaben, in denen man einzelne Blöcke temporär „wegklappen“ möchte,
- Textdateien, in denen man nur bestimmte Abschnitte manuell als logische Blöcke markieren will,
- punktuelle Folds in Sprachen ohne brauchbare `syntax`- oder `indent`-Faltung.

Vorteile:

- vollständige Kontrolle: nur die Folds, die explizit angelegt wurden,
- unabhängig von Syntax-Definitionen oder Einrückungstiefe,
- robust gegen „kaputte“ Syntax-Dateien.

Nachteile:

- Folds werden bei strukturellen Änderungen *nicht* automatisch angepasst,
- Pflegeaufwand bei stark veränderlichen Dateien,
- ohne `:mkview`/`:mksession` gehen manuelle Folds beim Schließen des Buffers verloren.

Übungen

1. Öffne eine längere Textdatei und lege drei manuelle Folds an:
   - einen Absatz mit z f a p
   - einen Block von fünf Zeilen mit z f 5 j
   - einen Visual-Bereich mit V und z f

   Öffne und schließe die Folds mit z o, z c und z a.
2. Entferne nacheinander:
   - nur den Fold unter dem Cursor mit z d,
   - alle Folds, die die aktuelle Zeile betreffen, mit z D,
   - alle manuellen Folds in der Datei mit z E.

   Beobachte, wann welcher Befehl wie „großflächig“ wirkt.
3. Setze `:set foldmethod=manual` und speichere deine aktuelle Sicht mit:
   - `:mkview`

   Schließe die Datei, öffne sie erneut und lade die Sicht mit:
   - `:loadview`

   Kontrolliere, ob alle manuellen Folds und der Faltzustand wiederhergestellt wurden.

10.5.2. Indent (`indent`)

Bei der Einrückungsmethode erzeugt VIM Folds automatisch aus der Einrückungstiefe der Zeilen. Je weiter eine Zeile eingerückt ist, desto tiefer liegt ihr Fold-Level:

- mehr Einrückung ⇒ „tiefer“ liegendes Fold
- weniger Einrückung ⇒ übergeordnetes Fold

Aktiviert wird diese Methode mit:

```
:set foldmethod=indent
```

Typische Einsatzfelder:

- Python (Blöcke werden rein über Einrückung definiert),
- YAML- und JSON-Konfigurationen,
- Markdown-Listen oder Gliederungen,
- generell alle Dateien, in denen Einrückung bereits die logische Struktur abbildet.

Wichtige Optionen im Zusammenhang mit `indent`-Folds:

- `:set foldnestmax=5` begrenzt, wie tief Folds ineinander verschachtelt werden dürfen.
- `:set foldminlines=2` erzeugt nur Folds, die mindestens zwei Zeilen umfassen (verhindert lauter Ein-Zeilen-Folds).
- `:set foldignore=\#` ignoriert Zeilen, die mit # beginnen, bei der Berechnung von Einrückungs-Folds (praktisch für Konfigdateien).

Typische Stolperfallen.

- Gemischte Tabs/Spaces: Wenn die Einrückung inkonsistent ist, entstehen „krumme“ Folds. Konsequent `:set expandtab` / `:set tabstop` / `:set shiftwidth` verwenden.
- Manuelle Einrückung per Space-Bashing statt mit [=] und Bewegungen (z. B. [=][a][p]) führt schnell zu ungleichmäßigen Blocks und damit zu unübersichtlichen Folds.
- Automatische Formatter (z. B. `:\%!black` für Python) können vorhandene Einrückungen verändern — danach Folds mit [z][X] neu berechnen.

Übungen

1. Erstelle eine kleine Python- oder YAML-Datei mit mehreren verschachtelten Blöcken (Funktion, innere Schleife, `if`-Block). Aktiviere anschließend:
   - `:set foldmethod=indent`
   - `:set foldlevel=0`

   Öffne und schließe die Folds mit [z][o], [z][c], [z][O] und [z][M]. Beobachte, wie die Einrückung die Falttiefe bestimmt.
2. Stelle `:set foldminlines=3` ein und füge in deiner Datei verschieden lange Blöcke ein. Prüfe, ab welcher Blocklänge VIM tatsächlich einen Fold anlegt und ab wann nicht mehr.
3. Erzeuge absichtlich „kaputte“ Einrückungen (z. B. eine Zeile im Block ohne Einrückung oder mit zu viel Einrückung) und beobachte, wie sich die Folds verändern. Repariere die Einrückung dann systematisch mit [=][a][p] oder [=][G] und vergleiche die Faltstruktur davor/danach.

10.5.3. Syntax (`syntax`)

Bei der syntaktischen Faltung (`foldmethod=syntax`) leitet VIM die Faltgrenzen direkt aus den Syntax-Definitionen ab, die auch für das Syntax-Highlighting verwendet werden. Funktionen, Klassen, Blöcke, lange Kommentare oder Dokumentationsabschnitte können so automatisch zu- und aufgeklappt werden.

Aktiviert wird diese Methode mit:

```
:set foldmethod=syntax
```

Voraussetzung ist, dass für den aktuellen Dateityp eine Syntax-Datei mit Faltungsregeln existiert und Syntax-Highlighting eingeschaltet ist:

```
:syntax on
```

Typische Einsatzfelder:

- Programmiersprachen mit klar definierten Blockstrukturen (C, C++, Java, Go, Rust, ldots),
- Skriptsprachen mit Funktions- oder Blocksyntax (Lua, Vimscript, JavaScript, TypeScript),
- große Konfigurations- oder Build-Dateien mit Abschnittsstruktur.

Vorteile:

- sehr präzise Faltgrenzen, orientiert an der tatsächlichen Sprache,
- robust gegen „kosmetische“ Änderungen wie reine Einrückungskorrekturen,
- Funktionen, Klassen und Kommentarblöcke lassen sich oft direkt als logische Einheiten auf- und zuklappen.

Wichtige Optionen.

- `:set foldnestmax=5` begrenzt die maximale Verschachtelungstiefe der syntaxbasierten Folds.
- `:set foldminlines=2` unterdrückt sehr kleine Folds; erst Blöcke mit mindestens zwei Zeilen werden gefaltet.
- `:set foldenable`/`:set nofoldenable` schaltet die Anzeige der Folds ein bzw. aus, ohne die Methode zu ändern.
- `:set foldlevelstart=1` legt fest, wie „tief“ die Folds beim Öffnen einer Datei standardmäßig sichtbar sind.

Typische Stolperfallen.

- Unvollständige oder fehlerhafte Syntax-Dateien führen zu merkwürdigen Faltgrenzen (z. B. wird eine ganze Datei als ein großer Fold erkannt).
- Bei stark verändertem Code (z. B. nach Massen-Ersetzungen) kann die Syntax-Analyse „hinterherhinken“. Hier hilft `z` `X` oder eine Kombination aus `:syntax sync fromstart` und `:set foldmethod=syntax`.
- Eigene Syntax-Erweiterungen (`after/syntax/...`) können Faltungen verbessern, aber auch ungewollte Folds produzieren, wenn `fold` zu großzügig verwendet wird.

Übungen

1. Öffne eine größere Quelltextdatei in einer Sprache mit vorhandener Syntax-Definition (z. B. C oder Lua). Aktiviere:
   - `:syntax on`
   - `:set foldmethod=syntax`
   - `:set foldlevel=0`

   Navigiere mit `z` `o`, `z` `c`, `z` `O` und `z` `M` durch die Folds und identifiziere, welche Sprachelemente jeweils gefaltet werden (Funktionen, Klassen, Kommentare).
2. Erhöhe schrittweise:
   - `:set foldnestmax=1`
   - `:set foldnestmax=3`
   - `:set foldnestmax=10`

   Beobachte, wie sich die sichtbare Falttiefe verändert und welche Blöcke neu als Fold erscheinen.
3. Füge in einer Datei bewusst lange Kommentarblöcke oder Dokumentationsabschnitte ein und prüfe, ob die Syntax-Datei dafür Faltungen anlegt. Falls nicht, recherchiere im `:help` unter dem betreffenden Dateityp, ob zusätzliche Optionen oder eigene Syntax-Erweiterungen nötig sind.

10.5.4. Expr (`expr`)

Die Ausdrucksmethode (`foldmethod=expr`) erlaubt eine komplett freie Definition von Folds über eine VIM-Funktion. Für jede Zeile wird eine Funktion aufgerufen, die einen *Fold-Level* zurückliefert – damit kannst du exakt steuern, wo Folds beginnen, enden oder gar nicht existieren.

Aktivierung:

```
:set foldmethod=expr
:set foldexpr=MyFoldFunction(v:lnum)
```

Dabei gilt:

- `foldexpr` ist ein VIM-Ausdruck, der für jede Zeile ausgewertet wird.
- `v:lnum` ist die aktuelle Zeilennummer, die an die Funktion übergeben wird.
- Die Funktion `MyFoldFunction()` entscheidet anhand des Zeileninhalts, welchen Fold-Level die jeweilige Zeile bekommt.

Typische Rückgabewerte der `foldexpr`-Funktion:

- 0 Kein Fold an dieser Zeile.
- 1, 2, ...Absoluter Fold-Level (Verschachtelungstiefe) dieser Zeile.
- `">N"` Beginn eines Folds auf Level N an dieser Zeile.
- `"="` Fold-Level der vorherigen Zeile unverändert übernehmen.
- `"-1"` VIM soll den automatisch bestimmten Level beibehalten (z. B. in Kombination mit anderen Methoden).

Ein einfaches Beispiel. Falte alle Überschriften, die mit ## beginnen, zu einem Fold Level 1:

```
function! MyFoldFunction(lnum)
  let line = getline(a:lnum)

  " Leerzeilen: keine Faltung
  if line =~ '^\s*$'
    return 0
  endif

  " Überschriften: Fold-Level 1
  if line =~ '^## '
    return 1
  endif

  " alles andere bleibt auf Level 0
  return 0
endfunction
```

Aktivierung:

```
:set foldmethod=expr
:set foldexpr=MyFoldFunction(v:lnum)
```

Performance-Hinweis. `foldexpr` wird für jede Zeile aufgerufen. Die Funktion sollte deshalb:

- nur mit `:getline()` und einfachen Regexen arbeiten,
- keine teuren Operationen (`:search()`, Shell-Befehle, ...) ausführen,
- *keine Seiteneffekte* haben (nur berechnen, nichts ändern).

Hilfreiche Optionen:

- `:set foldminlines=2` verhindert Mini-Folds mit nur einer Zeile.
- `:set foldnestmax=5` begrenzt die maximale Verschachtelung.
- `:set foldlevelstart=1` legt fest, wie viel beim Öffnen einer Datei sichtbar ist.

Übungen

1. Schreibe eine eigene `foldexpr`-Funktion, die alle Zeilen, die mit `TODO:` beginnen, als Beginn eines Folds behandelt und bis zur nächsten Leerzeile falten lässt.
   - Definiere die Funktion in deiner `vimrc`.
   - Aktiviere sie mit `:set foldmethod=expr` und `:set foldexpr=DeineFunktion(v:lnum)`.
   - Teste die Folds mit [z][M] und [z][R].
2. Baue eine `foldexpr`, die Einrückung und Marker kombiniert:
   - Zeilen mit `// SECTION` sollen Level 1 sein.
   - Tiefer eingerückte Zeilen innerhalb dieses Abschnitts sollen Level 2 bekommen.

   Überprüfe anschließend mit [z][c], [z][o] und [z][O], ob die Folds so reagieren, wie du erwartest.
3. Experimentiere mit Fehlersituationen: Füge in deine Funktion bewusst einen Bug ein (z. B. ein ungültiges Regex) und beobachte, welche Meldungen VIM ausgibt und wie sich die Folds verhalten. Korrigiere den Fehler und erzwinge eine Neuberechnung mit [z][X].

10.5.5. Marker

Bei der Marker-Methode werden explizite Start- und Endmarkierungen im Text gesetzt. Standardmäßig verwendet VIM:

```
{{{    …    }}}
```

Aktiviert wird diese Methode mit:

```
:set foldmethod=marker
:set foldmarker={{{,}}}
```

Damit interpretiert VIM jede Zeile, die {{{ enthält, als Faltbeginn und jede Zeile mit }}} als Faltende. Alles dazwischen wird zu einem Fold zusammengefasst.

Marker eignen sich besonders für:

- Dokumentationen und Howtos,
- Konfigurationsdateien,
- LaTeX- oder Skript-Kommentarblöcke,
- strukturierte Notizen mit klaren Abschnittsgrenzen.

Typische Beispiele (jeweils in Kommentaren, damit der Code gültig bleibt):

```
" {{{  Hauptabschnitt
" … Inhalt …
" }}}  Hauptabschnitt Ende

// {{{ Initialisierung
// … C-/C++-Code …
 // }}} Initialisierung Ende

% {{{ Präambel
% … LaTeX-Einstellungen …
% }}} Präambel Ende
```

Wichtig ist, dass die Marker in der jeweiligen Sprache als Kommentare geschrieben werden, damit die Datei weiterhin sauber parsbar bleibt.

Marker anpassen. Die verwendeten Marker lassen sich beliebig ändern, solange ein Start-/Endpaar definiert ist. Beispiel:

```
:set foldmarker=<<<,>>>
```

Dann würden die folgenden Zeilen einen Fold bilden:

```
# <<< Logging
# … Konfiguration …
# >>> Logging
```

Sinnvoll ist es, Marker zu wählen, die in der Sprache sonst nicht vorkommen, damit keine „zufälligen“ Folds entstehen.

Übungen

1. Richte in einer kleinen Konfigurationsdatei markerbasierte Folds ein:
   - Aktiviere Marker-Faltung mit `:set foldmethod=marker`.
   - Nutze die Standardmarker `{{{` und `}}}`.
   - Erzeuge mindestens drei logisch getrennte Abschnitte.
   - Öffne und schließe die Folds mit z o, z c, z O und z M.
2. Ändere die Marker auf ein eigenes Paar:
   - Setze z. B. `:set foldmarker=<<<,>>>`.
   - Passe deine Kommentarzeilen entsprechend an.
   - Prüfe mit z R und z M, ob alle Folds korrekt erkannt werden.
3. Kombiniere markerbasierte Folds mit einem Workflow:
   - Erzeuge Abschnitte wie `Konfiguration`, `Funktionen`, `Debug`.
   - Falte alle Bereiche mit z M und öffne gezielt nur den Abschnitt, an dem du arbeitest, mit z o / z O.
   - Achte darauf, wie sich dein Navigationsverhalten bei großen Dateien damit verändert.

10.5.6. Diff

In Diff-Ansichten faltet VIM automatisch alle unveränderten Bereiche und zeigt nur die tatsächlich unterschiedlichen Blöcke an. Technisch geschieht dies über `foldmethod=diff`, das aktiv ist, sobald der Diff-Modus eingeschaltet wird.

Typische Einstiege in den Diff-Modus:

- `:vert diffsplit` – aktuelle Datei vertikal mit einer zweiten Version vergleichen.
- `:diffsplit <Datei>` – Vergleich mit einer explizit angegebenen Datei.
- `:tab diffsplit` – Diff in neuem Tab öffnen.
- `vim -d file1 file2` – zwei Dateien direkt im Diff-Modus starten.

Im Diff-Modus gelten zusätzlich spezielle Bewegungen und Kommandos:

- [c /] c – zum vorherigen / nächsten Unterschied springen.
- `:diffget` – Änderungen aus dem anderen Fenster übernehmen.
- `:diffput` – eigene Änderungen ins andere Fenster übertragen.

Die Faltung im Diff-Modus sorgt dafür, dass du nicht durch große unchanged-Blöcke scrollen musst: Unveränderte Bereiche werden automatisch gefaltet und lassen sich bei Bedarf mit z o, z O kurz öffnen.

Übungen

1. Lege zwei Varianten derselben Datei an (z. B. `config_alt` und `config_neu`). Öffne sie mit:
   - `vim -d config_alt config_neu`

   Navigiere dann mit [c und] c durch die Unterschiede und beobachte, wie unveränderte Bereiche automatisch gefaltet sind.
2. Öffne eine Datei ganz normal, nimm Änderungen vor und speichere sie als zweite Version. Starte danach VIM nur mit der neuen Datei und rufe:
   - `:vert diffsplit <alte-Datei>`

   Spiele mit z o, z c, z O und z M, um ein Gefühl dafür zu bekommen, wie Diff-Folds die Sicht strukturieren.
3. Teste `:diffget` und `:diffput`: Übernimm einzelne Blöcke aus der „anderen Seite“ und kontrolliere danach mit u und Enter, wie sich der Inhalt verändert hat.

10.5.7. Foldmethoden wechseln

Ein Wechsel der Faltmethode ist jederzeit möglich und erfolgt über die Option `foldmethod`:

```
:set foldmethod=indent
:set foldmethod=syntax
```

```
:set foldmethod=marker
:set foldmethod=manual
```

Beim Wechsel werden vorhandene Folds verworfen und auf Grundlage der neuen Methode neu erzeugt. In großen Dateien kann dies einige Zeit in Anspruch nehmen – insbesondere bei `syntax` oder komplexen `expr`-Definitionen.

Typischer Workflow:

- „Lesemodus“ mit `foldmethod=syntax` oder `indent`, um automatisch eine grobe Struktur zu erhalten.
- Temporär auf `manual` umschalten, um eigene Faltungen mit z f, z d, z E zu setzen.
- Später wieder auf `syntax` oder `indent` zurückkehren, wenn sich die Datei stark verändert hat.

Praktisch ist es, sich pro Dateityp ein klares Standardprofil anzugewöhnen, etwa:

- Code (C, Python, Lua, ...): `syntax` oder `indent`
- Konfigurationsdateien: `marker` oder `indent`
- freie Notizen / Logs: `manual` oder `marker`

Übungen

1. Öffne eine größere Quelldatei und setze nacheinander:
   - `:set foldmethod=syntax`
   - `:set foldmethod=indent`
   - `:set foldmethod=manual`

   Beobachte, wie sich die Faltstruktur jeweils ändert und welche Methode dir für diesen Dateityp am sinnvollsten erscheint.
2. Wähle eine Konfigurationsdatei und richte markerbasierte Folds ein (`foldmethod=marker`). Wechsle anschließend kurz auf `manual` und wieder zurück auf `marker`, um zu sehen, wie VIM die Marker jedes Mal neu interpretiert.
3. Experimentiere mit Performance: Öffne eine wirklich große Datei und miss grob die Zeit für:
   - `:set foldmethod=syntax`
   - `:set foldmethod=indent`

   Ziel ist ein Gefühl dafür zu bekommen, ab welchen Dateigrößen du in der Praxis lieber auf leichtere Methoden ausweichst.

11. Fenster und Splits

11.1. Fenster teilen

VIM erlaubt es, die aktuelle Ansicht in mehrere Fenster zu unterteilen. Jedes Fenster zeigt einen Puffer – das kann dieselbe Datei oder eine andere Datei sein. Durch Splits lässt sich parallel arbeiten, vergleichen oder kopieren, ohne ständig zwischen Dateien wechseln zu müssen.

11.1.1. Horizontale Splits

Ein horizontaler Split teilt das aktuelle Fenster in eine obere und eine untere Hälfte. Beide Fenster bleiben logisch unabhängig, teilen sich aber denselben Bildschirmbereich.

Wichtige Ex-Befehle:

- `:split` Teilt den aktuellen Puffer horizontal. Die Datei erscheint danach in zwei Fenstern übereinander.
- `:sp` Kurzform von `:split` – praktisch identisch, nur schneller zu tippen.
- `:split config.yaml` Öffnet `config.yaml` im *unteren* Fenster, während das obere Fenster die bisherige Datei behält.

Zu den Ex-Befehlen gibt es äquivalente Normal-Mode-Kurzformen:

- **Ctrl**-**w** **s** Entspricht `:split`: aktuelles Fenster horizontal teilen.
- **Ctrl**-**w** **n** Öffnet ein neues leeres Fenster unterhalb (ähnlich `:new`).

Beispiel:

```
:split config.yaml
```

Oben bleibt z.B. dein Quelltext sichtbar, unten siehst du gleichzeitig die Konfiguration. Änderungen sind in beiden Fenstern live sichtbar, da sie denselben Puffer betreffen.

Übungen

1. Öffne eine beliebige Datei und erzeuge einen horizontalen Split:
   - Zuerst mit `:split`
   - Danach im Normal Mode mit **Ctrl**-**w** **s**

   Beobachte, dass beide Fenster dieselbe Datei zeigen, aber unabhängig scrollen.
2. Öffne im oberen Fenster deinen Quelltext, im unteren Fenster eine Konfigurationsdatei mit:
   - `:split config.yaml`

   Scrolle oben und unten getrennt und kopiere eine Zeile per **y** **y** im oberen Fenster und **p** im unteren. Ziel: Gefühl für „ein Puffer, zwei Blickwinkel“ bzw. „zwei Dateien, zwei Fenster“.
3. Erzeuge einen Split mit **Ctrl**-**w** **s** und prüfe, dass beide Fenster denselben Undo-Stack teilen: Ändere im oberen Fenster eine Zeile, führe dann im unteren Fenster **u** aus und beobachte, dass die Änderung dort ebenfalls zurückgenommen wird.

11.1.2. Vertikale Splits

Ein vertikaler Split teilt das aktuelle Fenster in eine linke und eine rechte Hälfte. Das ist ideal, um zwei Dateien oder zwei Bereiche derselben Datei nebeneinander zu vergleichen.

Wichtige Ex-Befehle:

- `:vsplit` Teilt das aktuelle Fenster vertikal. Die Datei erscheint danach in zwei Fenstern nebeneinander.
- `:vs` Kurzform von `:vsplit` – identische Wirkung, weniger zu tippen.
- `:vsplit main.c` Öffnet `main.c` im rechten Fenster, während links der bisherige Puffer erhalten bleibt.

Dazu kommen die Normal-Mode-Kurzbefehle über **Ctrl**-**w**:

- **Ctrl**-**w** **v** Vertikal teilen; entspricht `:vsplit`.
- **Ctrl**-**w** **n** In manchen Setups: neues vertikales Fenster mit leerem Puffer (abhängig von Konfiguration, z. B. Mapping in der `vimrc`).

Beispiel:

```
:vsplit main.c
```

Links bleibt etwa eine Header-Datei sichtbar, rechts `main.c`. Änderungen im einen Fenster sind sofort im anderen sichtbar, sofern beide denselben Puffer anzeigen.

Übungen

1. Öffne eine Datei `main.c` und erzeuge einen vertikalen Split mit:
   - `:vsplit`
   - danach im Normal Mode mit **Ctrl**-**w** **v**

   Scrolle beide Fenster unabhängig voneinander und beobachte, dass die Statuszeilen jeweils den gleichen Puffer anzeigen.
2. Öffne links `main.c` und rechts `main.h`:
   - `:edit main.c`
   - `:vsplit main.h`

   Springe mit **Ctrl**-**w** **w** zwischen den Fenstern und kopiere eine Funktionsdeklaration per **y** **y** aus `main.h` nach `main.c` mit **p**.
3. Erzeuge nacheinander einen horizontalen und einen vertikalen Split:
   - **Ctrl**-**w** **s** (horizontal)
   - dann im oberen Fenster: **Ctrl**-**w** **v** (vertikal)

   Du erhältst drei Fenster. Übe das Navigieren mit:
   - **Ctrl**-**w** **w** (zum nächsten Fenster)
   - **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l** für die gezielte Richtung.

   Ziel: Fensterwechsel soll sich genauso selbstverständlich anfühlen wie Cursorbewegungen.

11.1.3. Splits beim Öffnen von Dateien

VIM kann bereits beim Start mehrere Dateien in getrennten Fenstern öffnen. So sparst du dir nachträgliche `:split`- oder `:vsplit`-Befehle und landest direkt in einer aufgeteilten Oberfläche.

Grundformen:

```
vim -O2 datei1 datei2     " vertikale Splits
vim -o2 datei1 datei2     " horizontale Splits
```

Dabei gilt:

- `-O` (großes O) erzeugt vertikale Splits – Dateien erscheinen nebeneinander.
- `-o` (kleines o) erzeugt horizontale Splits – Dateien erscheinen übereinander.
- Die Zahl dahinter (2, 3, ...) gibt an, wie viele Fenster beim Start erzeugt werden sollen.

Beispiele:

- `vim -O2 main.c main.h` Öffnet `main.c` und `main.h` in zwei vertikalen Fenstern.
- `vim -o3 a.txt b.txt c.txt` Öffnet drei Dateien untereinander in drei horizontalen Fenstern.
- `vim -O3 a.txt b.txt c.txt` Drei Dateien nebeneinander – gut für breite Monitore.

Wenn du weniger Dateien angibst als Fenster angefordert sind, füllt VIM die restlichen Fenster mit leeren Buffern. Gibst du mehr Dateien an als angeforderte Fenster, werden nur so viele Dateien angezeigt, wie Fenster erzeugt wurden – der Rest ist über die Bufferliste erreichbar.

Übungen

1. Starte VIM mit zwei Dateien nebeneinander:
   - `vim -O2 datei1.txt datei2.txt`

   Navigiere anschließend mit **Ctrl**-**w** **w** zwischen den Fenstern und ändere in jedem Fenster eine Zeile. Ziel: Du sollst sofort sehen, welche Datei in welchem Split liegt.
2. Starte VIM mit drei Dateien untereinander:
   - `vim -o3 alpha.txt beta.txt gamma.txt`

   Nutze danach im Normal Mode die Befehle **Ctrl**-**w** **j** und **Ctrl**-**w** **k**, um gezielt nach unten bzw. oben zu wechseln. Schließe das mittlere Fenster mit `:close` und beobachte, wie sich die übrigen Splits anpassen.
3. Kombiniere Start-Splits mit weiteren Splits im Editor:
   - `vim -O2 main.c main.h`
   - Im linken Fenster: **Ctrl**-**w** **s**, um horizontal zu splitten.

   Du erhältst drei Fenster (zwei links, eines rechts). Übe:
   - Fensterwechsel mit **Ctrl**-**w** **w**
   - gezieltes Ansteuern mit **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l**

Ziel: Du sollst sicher zwischen mehreren Splits navigieren können, egal ob sie beim Start oder nachträglich erzeugt wurden.

11.1.4. Split mit Zeilensprung

Mit einem *Plus-Argument* kannst du in einem neuen Split nicht nur eine Datei öffnen, sondern gleichzeitig zu einer bestimmten Position springen. Das funktioniert sowohl mit Zeilennummern als auch mit Mustern.

Grundform mit Zeilennummer:

```
:split +120 server.log
```

Bedeutung:

- `:split` – horizontaler Split mit neuer Datei
- `+120` – nach dem Laden sofort zu Zeile `120` springen
- `server.log` – zu öffnende Datei

Varianten:

- `:vsplit +42 main.c` Öffnet `main.c` rechts und springt direkt zu Zeile `42`.
- `:split +10` Teilt das aktuelle Fenster, lädt denselben Buffer und springt in der unteren Hälfte auf Zeile `10`.
- `:split +/ERROR/ server.log` Öffnet `server.log` im Split und springt zur nächsten Zeile, die ERROR enthält (statt einer festen Zeilennummer).

Übungen

1. Öffne eine Logdatei im Split und springe direkt in den interessanten Bereich:
   - `:split +200 server.log`

   Scrolle im unteren Fenster ein wenig und prüfe mit **G**, ob du tatsächlich etwa in der Mitte der Datei gelandet bist.
2. Nutze eine Mustersuche im Plus-Argument:
   - `:vsplit +/TODO/ main.c`

   Navigiere mit **Ctrl**-**w** **w** zwischen den Fenstern und vergleiche, wie der Cursor jeweils positioniert ist.
3. Simuliere eine „Schnellnavigation“ für lange Dateien:
   - `:split +50 README.md`

- `:split +200 README.md`

Beobachte, wie du mit zwei Splits dieselbe Datei an verschiedenen Stellen gleichzeitig im Blick behältst (oben z. B. Einleitung, unten Details).

11.1.5. Splits mit unterschiedlichen Puffern

Ein Split ist immer nur ein *Fenster* – der darin angezeigte Inhalt kommt aus einem *Buffer*. Mehrere Fenster können also *denselben* oder *verschiedene* Buffer zeigen. Das ist der Kern des Split-Konzepts in VIM.

Ein typischer Ablauf:

1. Aktuellen Buffer splitten:
   - `:split`

   Jetzt siehst du denselben Buffer in zwei Fenstern.
2. Im neuen Fenster einen anderen Buffer wählen:
   - `:buffer 3`

   Fenster bleibt gleich – nur der angezeigte Buffer wechselt.

Hilfreiche Ex-Kommandos dazu:

- `:ls` Listet alle offenen Buffer mit Nummer, Status und Dateinamen.
- `:buffer 3` Wechselt im aktuellen Fenster zu Buffer `3`.
- `:bnext`, `:bprev` Zum nächsten bzw. vorherigen Buffer springen.
- `:b\#` Zum zuletzt verwendeten Buffer zurückkehren („Toggle“).

Wichtig ist die Unterscheidung:

- Fenster (*window*): visuelle Ansicht – wo du hinschaust.
- Buffer: geladene Datei bzw. Inhalt – *was* du siehst.

Die Splits steuern nur die Fensterstruktur; welcher Buffer in welchem Fenster liegt, wird mit `:buffer`-Kommandos bestimmt.

Übungen

1. Öffne eine Datei `main.c`, erzeuge einen Split und lade oben und unten unterschiedliche Buffer:
   - `:edit main.c`
   - `:split`
   - **Ctrl**-**w** **w** (ins andere Fenster wechseln)
   - `:edit util.c`

   Prüfe mit `:ls`, welche Buffernummern `main.c` und `util.c` haben.
2. Arbeite mit Buffernummern:

- `:ls`
- Merke dir eine Nummer (z. B. 3) und wechsle im aktuellen Fenster gezielt dorthin: `:buffer 3`.

Beobachte, dass der Split selbst unverändert bleibt – nur der Inhalt wechselt.

3. Nutze Splits als „Zweifenster-Ansicht“ derselben Datei:
   - `:edit README.md`
   - `:split`
   - Scrolle im oberen Fenster ans Dokumentende (**G**), unten an den Anfang (**g** **g**).

   Wechsle mit **Ctrl**-**w** **w** hin und her und beobachte, dass beide Fenster denselben Buffer zeigen, aber an unterschiedlichen Positionen.

11.1.6. Mehrere Splits kombinieren

VIM erlaubt es, horizontale und vertikale Splits frei zu kombinieren. Aus einem einzelnen Fenster kann so ein komplettes Raster aus mehreren Ansichten entstehen.

Ein klassischer Ablauf:

```
:vsplit                 " vertikal teilen (links/rechts)
:split                  " unterhalb des aktuellen Fensters teilen
:vsplit                 " aktuelles Fenster noch einmal vertikal tei
```

Nach diesen drei Kommandos hast du bereits vier Fenster, die alle unabhängig scollen können und jeweils ihren eigenen Buffer anzeigen dürfen. Typische Layouts:

- Links Quelltext, rechts Tests
- Oben Code, unten Log-Datei
- Links Doku, rechts Konfiguration und darunter Log

Wichtig ist dabei die Trennung von:

- *Fenstern* (Splits) – visuelle Aufteilung des Bildschirms
- *Buffern* – eigentliche Inhalte (Dateien, temporäre Puffer)

Die Splits definieren nur die Anordnung. Welcher Buffer in welchem Fenster liegt, bestimmst du mit Befehlen wie `:edit`, `:buffer`, `:bnext`, `:bprev`.

Größe und Anordnung anpassen

Sobald mehrere Splits geöffnet sind, kannst du die Größen feinsteuern:

- **Ctrl**-**w** **=** alle Fenster gleich groß machen

- Ctrl-w+, Ctrl-w- aktuelle Fensterhöhe erhöhen/verringern
- Ctrl-w>, Ctrl-w< aktuelle Fensterbreite verändern

Diese Kommandos wirken immer relativ auf das Fenster, in dem sich der Cursor befindet. Mit Ctrl-ww oder Ctrl-wh/j/k/l wechselst du schnell zwischen den Fenstern.

Für gezielte Anpassungen gibt es zusätzlich Ex-Kommandos:

- `:resize 20` aktuelle Fensterhöhe auf 20 Zeilen setzen
- `:vertical resize 80` aktuelle Fensterbreite auf 80 Spalten setzen

Übungen

1. Erzeuge ein 2x2-Raster:
   - Öffne eine Datei mit `:edit main.c`.
   - Erzeuge zunächst einen vertikalen Split mit `:vsplit`.
   - Erzeuge in beiden Fenstern jeweils einen horizontalen Split mit `:split`.

   Du solltest jetzt vier Fenster sehen. Wechsle mit Ctrl-ww im Kreis durch alle Fenster.
2. Weise den vier Fenstern unterschiedliche Dateien zu:
   - Lade z. B. `main.c`, `util.c`, `config.yaml`, `README.md`.
   - Nutze `:ls`, um die Buffernummern zu sehen, und dann `:buffer <nr>`, um in jedem Fenster gezielt einen anderen Buffer anzuzeigen.
3. Optimiere das Layout:
   - Mache alle Fenster mit Ctrl-w= gleich groß.
   - Vergrößere das Fenster mit dem Log (z. B. `server.log`) mit Ctrl-w+ so lange, bis du genug Kontext siehst.
   - Nutze `:resize` oder `:vertical resize`, um eine feste Größe zu setzen (z. B. `:resize 15` für ein flaches Log-Fenster).

   Ziel: Ein Setup, in dem du Code, Tests und Log gleichzeitig komfortabel im Blick hast.

11.2. Fenster wechseln

Sobald mehrere Splits geöffnet sind, wird das effiziente Wechseln zwischen den Fenstern zentral. VIM stellt dafür zwei Ebenen bereit:

- schnelle Tastenkombinationen für den Alltag,

- Ex-Befehle für Skripte, Mappings und Makros.

11.2.1. Wechseln per Tastenkombination

Die Standardkombinationen bestehen aus **Ctrl**-**w** gefolgt von einer Richtungstaste:

- **Ctrl**-**w** **h** — in das Fenster **links** wechseln
- **Ctrl**-**w** **j** — in das **untere** Fenster wechseln
- **Ctrl**-**w** **k** — in das **obere** Fenster wechseln
- **Ctrl**-**w** **l** — in das Fenster **rechts** wechseln

Diese Kombinationen funktionieren unabhängig davon, ob die Fenster horizontal, vertikal oder gemischt angeordnet sind. Wichtig ist: **Ctrl**-**w** ist hier ein *Prefix-Operator* für Fensteraktionen – vergleichbar mit **g** oder **z** für andere Befehlsgruppen.

Weitere nützliche Varianten:

- **Ctrl**-**w** **w** — zum *nächsten* Fenster im Kreis springen
- **Ctrl**-**w** **W** — zum nächsten Fenster *in anderer Tabpage*
- **Ctrl**-**w** **t** — zum *obersten linken* Fenster springen
- **Ctrl**-**w** **b** — zum *untersten rechten* Fenster springen

Damit lassen sich typische Layouts schnell ablaufen: etwa von oben links nach unten rechts, oder immer im Kreis mit **Ctrl**-**w** **w**.

Übungen

1. Erzeuge ein Layout mit vier Fenstern (z. B. über Splits) und navigiere einige Minuten nur mit:
   - **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l**
   - **Ctrl**-**w** **w** zum zyklischen Durchlaufen

   Ziel: Du solltest jederzeit „blind“ ins gewünschte Fenster wechseln können, ohne über die Tasten nachzudenken.
2. Öffne in jedem Fenster eine andere Datei (z. B. `main.c`, `util.c`, `config.yaml`, `README.md`) und springe gezielt:
   - von beliebiger Position mit **Ctrl**-**w** **t** in das „Masterfenster“ oben links,
   - danach mit **Ctrl**-**w** **b** in das diagonale Gegenstück unten rechts.

Beobachte, wie sich diese „Eckensprünge“ für typische Arbeitslayouts einsetzen lassen.

11.2.2. Wechseln per Ex-Befehl

Jeder **Ctrl**-**w**-Befehl besitzt eine Ex-Entsprechung über `:wincmd`:

- `:wincmd h` – entspricht **Ctrl**-**w** **h**
- `:wincmd j` – entspricht **Ctrl**-**w** **j**
- `:wincmd k` – entspricht **Ctrl**-**w** **k**
- `:wincmd l` – entspricht **Ctrl**-**w** **l**
- `:wincmd w` – entspricht **Ctrl**-**w** **w**

Das ist vor allem nützlich für:

- Mappings in der `vimrc`, die Fensterwechsel mit anderen Aktionen kombinieren,
- Skripte und Functions, die gezielt in bestimmte Fenster springen müssen,
- Makros, die *explizit* mit `:wincmd` arbeiten sollen, statt **Ctrl**-**w** im aufgezeichneten Key-Stream zu verwenden.

Übungen

1. Rufe nacheinander im Command-Line-Modus auf:
   - `:wincmd h`
   - `:wincmd j`
   - `:wincmd k`
   - `:wincmd l`

   und vergleiche das Verhalten mit den entsprechenden **Ctrl**-**w**-Kombinationen.
2. Zeichne ein Makro auf, das:
   a) im aktuellen Fenster eine kleine Änderung vornimmt,
   b) dann in das rechte Fenster wechselt,
   c) dort eine andere Änderung ausführt,
   d) und schließlich wieder ins Ausgangsfenster zurückkehrt.

   Probiere eine Variante mit **Ctrl**-**w** **l** im Makro und eine zweite Variante, die explizit `:wincmd l` nutzt. Ziel: Verstehen, wie Fensterwechsel im Makro-Stream erscheinen und wie du sie bei Bedarf lesbar über Ex-Kommandos steuern kannst.

11.2.3. Wechsel zum zuletzt oder nächsten Fenster

Neben den richtungsbasierten Wechseln gibt es zyklische Fensterwechsel sowie den Sprung zurück zum zuvor aktiven Fenster:

- **Ctrl**-**w** **w** – wechselt zyklisch zum *nächsten* Fenster im Fenster-Ring.
- **Ctrl**-**w** **W** – wechselt zyklisch in die *andere Richtung* durch alle Fenster.
- **Ctrl**-**w** **p** – springt in das *zuletzt benutzte* Fenster zurück (praktisch, wenn man zwischen zwei Splits hin- und herpendelt).

Diese Befehle sind besonders hilfreich, wenn viele Splits geöffnet sind und eine reine Navigation über **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l** zu mühsam wird.

Übungen

1. Öffne zwei Dateien in vertikalen Splits:
   - `:vsplit datei1.txt`
   - `:vsplit datei2.txt`

   Wechsle nun mehrfach mit **Ctrl**-**w** **h** und **Ctrl**-**w** **l** zwischen linkem und rechtem Fenster, ohne die Maus zu benutzen.
2. Öffne drei Dateien:
   - `:edit datei1.txt`
   - `:vsplit datei2.txt`
   - `:split datei3.txt`

   Pendle nun zwischen zwei Fenstern mit **Ctrl**-**w** **p**. Ziel: Du sollst „blind“ zwischen genau diesen beiden Fenstern springen können, ohne nachdenken zu müssen.
3. Erzeuge vier Splits (z. B. zwei vertikale und zwei horizontale). Nutze anschließend:
   - **Ctrl**-**w** **w**, um zyklisch durch alle Fenster zu laufen,
   - **Ctrl**-**w** **W**, um in die Gegenrichtung zu laufen.

   Beobachte, in welcher Reihenfolge Vim die Fenster durchläuft und überlege dir, welches Muster dahintersteckt (Fenster-Ring).

11.2.4. Direkter Sprung per Puffer-ID

Wichtig ist die Unterscheidung zwischen *Puffer* und *Fenster*:

- Ein *Puffer* ist der geladene Inhalt einer Datei.
- Ein *Fenster* ist nur die Sicht auf einen Puffer.

Die Puffer haben interne Nummern, die über `:ls` oder `:buffers` angezeigt werden:

```
:ls
:buffers
```

Mit:

```
:buffer <nummer>
```

lädst du gezielt den gewünschten Puffer in das *aktuelle* Fenster. Typische Varianten:

- `:buffer 3` lädt Puffer 3 in das aktuelle Fenster.
- `:bnext` zum nächsten Puffer wechseln.
- `:bprev` zum vorherigen Puffer wechseln.

Gezielt ins richtige Fenster springen

Das *Fenster* bestimmst du weiterhin mit den normalen Fenster-Kommandos:

- **Ctrl**-**w** **w** zum nächsten Fenster springen
- **Ctrl**-**w** **p** zum zuvor aktiven Fenster zurückkehren
- **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l** gezielt nach links/unten/oben/rechts

Die Ex-Variante dazu lautet:

`:wincmd w` (entspricht **Ctrl**-**w** **w**)
`:wincmd h` (entspricht **Ctrl**-**w** **h**)
`:wincmd l` (entspricht **Ctrl**-**w** **l**)

Ein typischer Workflow ist:

1. Mit **Ctrl**-**w** **h**/**j**/**k**/**l** in das gewünschte Fenster springen.
2. Dort mit `:buffer <nummer>` den gewünschten Puffer laden.

11.2.5. Fenster wechseln mit Maus

Wenn die Maus aktiviert ist, kannst du Fenster auch per Klick wechseln statt nur mit **Ctrl**-**w** **h**/**j**/**k**/**l**.

Die Mausunterstützung schaltest du z. B. so ein:

```
:set mouse=a
```

- `mouse=a` – Maus in allen Modi aktiv (Normal, Insert, Visual ...)
- `mouse=n` – nur im Normal Mode
- `mouse=` – Maus vollständig deaktivieren

Ist die Maus aktiv, gilt:

- Ein einfacher Linksklick in ein Fenster setzt den Cursor dort und macht dieses Fenster aktiv.
- In Visual-ähnlichen Modi können Drag-Operationen Text markieren (abhängig vom Terminal/GUI).
- Mausrad oder Trackpad-Scroll wirken nur im aktuell aktiven Fenster.

Hinweise

- In reinen Terminal-Umgebungen hängt das Verhalten stark vom Terminalemulator ab (xterm, iTerm2, Alacritty ...).
- Wer VIM hauptsächlich per Tastatur bedient, nutzt die Maus oft nur situativ – z. B. bei vielen Splits oder auf sehr großen Monitoren.
- Ein schneller „Notausstieg“ ist immer:

  ```
  :set mouse=
  ```

  falls die Maus unerwartet Auswahl oder Einfügemarken beeinflusst.

Übungen

1. Aktiviere die Maus mit `:set mouse=a` und öffne drei Splits (z. B. `:vsplit`, `:split`). Wechsle abwechselnd:
   - nur mit **Ctrl**-**w** **h**/**j**/**k**/**l**
   - nur mit Mausklick in das gewünschte Fenster

   Beobachte, welche Variante dir in welcher Situation schneller vorkommt.
2. Deaktiviere die Maus mit `:set mouse=` und prüfe, ob dein Muskelgedächtnis für **Ctrl**-**w** **h**/**j**/**k**/**l** ausreichend sitzt. Ziel: Maus als Option kennen, aber nicht davon abhängig sein.

11.2.6. Fensterwechsel nach Öffnen neuer Splits

VIM fokussiert nach einem neuen Split automatisch das *neu erzeugte* Fenster. Das gilt z. B. für:

- `:split` / `:vsplit`
- `:new` / `:vnew`
- Start mit mehreren Dateien, etwa `:vim -O datei1 datei2`

Oft möchte man danach aber sofort in das *vorher aktive* Fenster zurückspringen – zum Beispiel, wenn man nur kurz eine Referenzdatei, Logdatei oder Konfigurationsdatei im neuen Split prüfen wollte.

Dafür gibt es:

- **Ctrl**-**w** **p** – wechselt zurück zum zuvor aktiven Fenster

Damit bildet **Ctrl**-**w** **p** ein Paar zu **Ctrl**-**w** **w**:

- **Ctrl**-**w** **w** – zyklisch durch alle Fenster
- **Ctrl**-**w** **p** – zwischen *aktuellen* und *vorigen* Fenster hin- und herschalten (Toggeln)

Typischer Workflow:

1. Hauptarbeit im linken Fenster (Quelltext, Dokument, Konfiguration).
2. `:vsplit logfile.log` öffnen – Fokus liegt nun auf dem neuen Split.
3. Kurz scrollen, etwas nachsehen.
4. Mit **Ctrl**-**w** **p** zurück ins ursprüngliche Arbeitsfenster springen.

Übungen

1. Öffne eine Datei und erzeuge zwei weitere Splits (z. B. `:vsplit`, `:split`). Wechsle nun:
   - ein paar Mal mit **Ctrl**-**w** **w** durch alle Fenster,
   - danach mehrmals mit **Ctrl**-**w** **p** hin und her.

   Ziel: den Unterschied zwischen „rundherum springen“ und „zwischen zwei Fenstern toggeln“ im Muskelgedächtnis verankern.
2. Öffne im Hauptfenster eine Quelltextdatei und in einem Split eine Log- oder Konfigurationsdatei. Nutze **Ctrl**-**w** **p**, um wiederholt kurz in den Log-Split zu springen und dann zurück in den Code. Beobachte, wie viel schneller sich das anfühlt im Vergleich zu **Ctrl**-**w** **h**/**j**/**k**/**l**.

11.2.7. Fensterliste anzeigen

Neben den Bewegungsbefehlen mit **Ctrl**-**w** kann es hilfreich sein, sich einen kurzen Überblick über alle geöffneten Fenster zu verschaffen. Dafür dient das Ex-Kommando:

```
:windows
```

Dieses Kommando listet alle aktuell geöffneten Fenster mit:

- interner Fensternummer,
- zugeordnetem Puffer,
- Dateinamen bzw. Puffernamen,
- ggf. Markierungen wie + (geändert).

Gerade in komplexeren Layouts (mehrere vertikale und horizontale Splits, zusätzlich noch Tabs) verliert man sonst schnell den Überblick, *welches* Fenster welchen Inhalt zeigt.

Typische Einsatzsituationen:

- Du hast vier oder mehr Splits geöffnet und weißt nicht mehr, wo genau eine bestimmte Datei liegt.
- Du arbeitest mit mehreren Projekten gleichzeitig und willst kurz prüfen, welche Fenster noch aktive Änderungen enthalten.
- Du baust dir Makros oder Mappings, die gezielt auf bestimmten Fenstern arbeiten sollen, und brauchst dafür die Zuordnung.

Übungen

1. Öffne drei verschiedene Dateien und verteile sie auf mehrere Splits (z. B. `:vsplit`, `:split`). Führe anschließend `:windows` aus und notiere dir:
   - welche Fensternummer zu welcher Datei gehört,
   - welches Fenster gerade aktiv ist.
2. Wechsel mit **Ctrl**-**w** **w** und **Ctrl**-**w** **h**/**j**/**k**/**l** durch die Splits und rufe zwischendurch immer wieder `:windows` auf. Beobachte, wie sich die Markierung des aktiven Fensters in der Liste verändert.
3. Schließe eines der Fenster mit **Ctrl**-**w** **c** und rufe erneut `:windows` auf. Prüfe, wie sich die Liste verändert und welche Fensternummern noch übrig sind.

11.3. Fenstergrößen ändern

Bei mehreren Splits wird das Arbeiten oft angenehmer, wenn die Fenstergrößen angepasst werden. VIM bietet dafür sowohl relative Änderungen (größer/kleiner) als auch absolute Setzungen (exakte Höhe oder Breite).

11.3.1. Relative Größenänderungen

Die schnellen „auf/zu“-Anpassungen laufen alle über **Ctrl** - **w** als Fenster-Operator, gefolgt von einem Größenbefehl:

- **Ctrl** - **w** **+** macht das aktuelle Fenster **höher** (vergrößert die Höhe, die anderen werden entsprechend kleiner).
- **Ctrl** - **w** **-** macht das aktuelle Fenster **niedriger** (verkleinert die Höhe, die anderen werden höher).
- **Ctrl** - **w** **>** macht das aktuelle Fenster **breiter** (vergrößert die Breite zulasten der Nachbarfenster).
- **Ctrl** - **w** **<** macht das aktuelle Fenster **schmaler** (verkleinert die Breite, die Nachbarfenster gewinnen Platz).

Wichtig dabei:

- **+** und **-** wirken auf die *Höhe* des aktuellen Fensters.
- **<** und **>** wirken auf die *Breite* des aktuellen Fensters.
- Die Richtung (oben/unten bzw. links/rechts) ergibt sich aus der aktuellen Split-Anordnung; du musst nichts Besonderes beachten.

Mit einer vorangestellten Zahl lassen sich größere Schritte machen:

5 **Ctrl** - **w** **+**

macht das aktuelle Fenster in einem Schritt deutlich höher, statt fünfmal **Ctrl** - **w** **+** zu drücken. Analog funktionieren:

- **3** **Ctrl** - **w** **-**
- **1** **0** **Ctrl** - **w** **>**
- **8** **Ctrl** - **w** **<**

Die relative Änderung passt jeweils nur *das aktive Fenster* an; alle anderen Fenster werden proportional mitgezogen, sodass der verfügbare Bildschirmplatz immer vollständig genutzt bleibt.

Übungen

1. Öffne zwei horizontale Splits (z. B. `:split`) und probiere:
   - Ctrl - w + , Ctrl - w -
   - 5 Ctrl - w +

   Beobachte, wie sich die Höhe beider Fenster zueinander verändert.
2. Erzeuge zwei vertikale Splits mit `:vsplit` und passe die Breite an:
   - Ctrl - w >
   - Ctrl - w <
   - 1 0 Ctrl - w >

   Achte darauf, wie sich der Textumbruch in langen Zeilen ändert.
3. Baue ein Layout mit drei oder vier Fenstern (horizontal und vertikal gemischt). Versuche, mit möglichst wenigen Befehlen ein „Hauptfenster“ groß zu ziehen, in dem du arbeitest, während die anderen als kleine Referenzfenster sichtbar bleiben. Ziel: *Gefühl* dafür bekommen, wie schnell du Layouts nur mit Ctrl - w -Größenbefehlen umformen kannst.

11.3.2. Absolute Größen setzen

Neben den relativen Änderungen gibt es auch Befehle, die die Fenstergröße *absolut* setzen – entweder maximal in eine Richtung oder auf einen konkreten Wert.

Maximale Höhe/Breite per Tastenkombination

Über Ctrl - w und zwei Sondertasten lassen sich Fenster maximal ausdehnen:

- Ctrl - w | macht das aktuelle Fenster **maximal breit** (nutzt die gesamte verfügbare Bildschirmbreite für dieses Fenster).
- Ctrl - w _ macht das aktuelle Fenster **maximal hoch** (nutzt die gesamte verfügbare Bildschirmhöhe für dieses Fenster).

Mit vorangestellter Zahl wird eine *konkrete* Größe eingestellt:

- 4 0 Ctrl - w | setzt die **Breite** des Fensters auf exakt 40 Spalten.
- 1 5 Ctrl - w _ setzt die **Höhe** des Fensters auf exakt 15 Zeilen.

Diese Varianten sind ideal, wenn ein Fenster „Arbeitsfenster“ sein soll, während andere Splits nur als Referenz am Rand geöffnet bleiben.

Exakte Größen mit Ex-Befehlen

Die gleichen Operationen lassen sich auch über Ex-Kommandos steuern:

```
:resize 20
```

setzt die **Höhe** des aktuellen Fensters auf 20 Zeilen.

```
:vertical resize 80
```

setzt die **Breite** des aktuellen Fensters auf 80 Spalten (egal, wie viele vertikale Splits existieren).

Typische Einsatzfälle:

- Logs oder Konsolenbereiche auf eine definierte Höhe bringen.
- Quelltextfenster auf eine feste Breite einstellen, um Zeilenlängen („80-Spalten-Regel“) kontrolliert einzuhalten.
- In Makros oder Mappings reproduzierbare Layouts aufbauen.

Übungen

1. Erzeuge zwei horizontale Splits und probiere:
   - Ctrl-w _
   - 1 0 Ctrl-w _
   - `:resize 5`, `:resize 20`

   Beobachte, wie sich die Gesamthöhe auf beide Fenster verteilt.
2. Erzeuge zwei vertikale Splits mit `:vsplit` und teste:
   - Ctrl-w |
   - 4 0 Ctrl-w |
   - `:vertical resize 100`

   Achte darauf, wie sich der Umbruch langer Zeilen verändert.
3. Baue ein Layout mit drei oder mehr Splits und stelle dir ein „Hauptfenster“ ein, indem du mit Ctrl-w | bzw. Ctrl-w _ und `:resize`/`:vertical resize` arbeitest. Ziel: ein reproduzierbares Standardlayout, das du später in ein Mapping oder eine Session übernehmen kannst.

11.3.3. Fensterverteilung automatisch ausbalancieren

Wenn mehrere Fenster nach diversen Größenanpassungen „aus dem Ruder“ gelaufen sind, bringt ein einziger Befehl alle Splits wieder in eine saubere Ausgangsform:

Ctrl-w =

- Ctrl-w = verteilt alle offenen Fenster automatisch so, dass sie wieder *gleich groß* sind.

Damit lassen sich Layout-Experimente schnell zurücksetzen, ohne einzelne Fenster manuell anpassen zu müssen. Besonders hilfreich ist das bei:

- vielen gleichzeitig geöffneten Splits,
- „verzogenen“ Layouts nach mehreren Ctrl-w + / Ctrl-w - oder `:resize`-Befehlen,
- kurzen Sessions, in denen man nur vorübergehend an einem Fenster „zieht“ und danach wieder eine neutrale Ausgangssituation braucht.

Als Ex-Variante existiert:

```
:wincmd =
```

die denselben Effekt wie Ctrl-w = hat, sich aber gut in Skripten und Mappings verwenden lässt.

Übungen

1. Öffne drei Dateien in vertikalen Splits:
   - `:edit datei1.txt`
   - `:vsplit datei2.txt`
   - `:vsplit datei3.txt`

   Verändere die Breiten mit Ctrl-w >, Ctrl-w < und bringe das Layout anschließend mit Ctrl-w = wieder auf gleiche Breite.
2. Erzeuge zwei horizontale und einen vertikalen Split (z. B. über `:split` und `:vsplit`). Passe die Fensterhöhen und -breiten mit Ctrl-w +, Ctrl-w -, Ctrl-w | und Ctrl-w _ an. Setze alles mit Ctrl-w = zurück und beobachte, wie VIM die Fläche neu verteilt.

3. Baue ein komplexeres Layout (mindestens vier Splits), in dem du ein „Hauptfenster“ per Ctrl-w | und Ctrl-w _ hervorhebst. Nutze anschließend Ctrl-w = und `:wincmd =` und vergleiche, ob beide Varianten sich identisch verhalten.

11.3.4. Fenstergrößen sperren

Manchmal soll ein Fenster seine Größe behalten, auch wenn andere Splits vergrößert oder verkleinert werden. Typische Beispiele sind:

- `quickfix`-Fenster,
- `help`-Fenster,
- Log- oder Terminalfenster am unteren Rand.

Dafür stellt VIM zwei Optionen bereit:

```
:setlocal winfixheight
:setlocal winfixwidth
```

- `winfixheight` verhindert, dass die *Höhe* des Fensters durch Layout-Operationen (weitere Splits, Ctrl-w =, `:resize` in anderen Fenstern) automatisch angepasst wird.
- `winfixwidth` verhindert entsprechend automatische Änderungen an der *Breite*.

Wichtig:

- Die Optionen wirken pro Fenster (`:setlocal`), nicht global.
- Direkte Befehle im aktuellen Fenster wie `:resize 20` bzw. `:vertical resize 80` sind weiterhin erlaubt – gesperrt sind nur *indirekte* Änderungen durch andere Fenster.
- Ctrl-w = respektiert `winfixheight` / `winfixwidth`: gesperrte Fenster bleiben stabil, andere passen sich an.

Übungen

1. Öffne eine Quelltextdatei und teile das Fenster horizontal:
   - `:edit main.c`
   - `:split`

   Aktiviere im unteren Fenster `:setlocal winfixheight` und verändere im oberen Fenster die Höhe mit Ctrl-w + und Ctrl-w -. Beobachte, welches Fenster seine Höhe beibehält.

2. Öffne ein `quickfix`-Fenster, z. B. über `:copen`. Setze in diesem Fenster `:setlocal winfixheight` und verwende anschließend **Ctrl**-**w** **=** sowie weitere Splits. Ziel: Das Quickfix-Fenster soll optisch stabil bleiben, während die übrigen Fenster „atmen“.
3. Erzeuge zwei vertikale Splits mit unterschiedlichen Dateien und setze im linken Fenster `:setlocal winfixwidth`. Ändere die Breite über **Ctrl**-**w** **>** und **Ctrl**-**w** **<** aus dem *rechten* Fenster heraus. Prüfe, wie sich die Breiten verteilen und welche Rolle `winfixwidth` dabei spielt.

11.3.5. Fenstergrößen mit Maus ändern

Wenn die Maus aktiviert ist (`:set mouse=a`), können Fenstergrößen auch direkt per Drag-and-Drop an den Splitleisten verändert werden. Das Verhalten entspricht dabei funktional den **Ctrl**-**w**-Größenbefehlen:

- horizontale Leiste nach oben/unten ziehen ⟶ Fensterhöhe ändern
- vertikale Leiste nach links/rechts ziehen ⟶ Fensterbreite ändern

Praktische Hinweise:

- In Terminal-VIM hängt die Präzision von der Terminalemulation ab; in *GVim* oder *Neovim-GUIs* ist die Maussteuerung meist deutlich flüssiger.
- Mausgesten ändern immer nur die Größe der beiden beteiligten Fenster – nicht der gesamten Split-Struktur.
- In sehr feinen Layouts (viele Splits) ist die Maus praktisch, um „mal eben“ nachzujustieren; für reproduzierbare Setups bleiben `:resize`-Befehle die stabilere Variante.

Übungen

1. Aktiviere die Maus mit `:set mouse=a` und erzeuge vier Fenster:
   - `:vsplit`
   - `:split`
   - `:split`

   Passe danach nur mit der Maus die Fenstergrößen an, bis eine sinnvolle Aufteilung für dich entsteht. Notiere dir anschließend eine äquivalente Folge von **Ctrl**-**w**-Befehlen, mit der du dieselbe Aufteilung auch tastaturbasiert erreichen würdest.
2. Deaktiviere die Maus temporär mit `:set mouse=` und vergleiche das Verhalten beim Anfassen der Splitleisten. Ziel: Verstehen, wann die Maus wirklich

einen Mehrwert bringt und wann sie eher versehentliche Verschiebungen produziert.

3. Lege dir ein eigenes Profil an, z. B.:
   - `:set mouse=a` für „exploratives Arbeiten“
   - `:set mouse=` für „Makro- und Tastaturfokus“

   Wechsle bewusst zwischen den Modi und beobachte, wie sich das auf dein Fenster- und Split-Handling auswirkt.

11.4. Buffer in Splits

Fenster und Buffer werden in VIM strikt getrennt behandelt: **Ein Fenster zeigt genau einen Buffer an, aber ein Buffer kann in mehreren Fenstern gleichzeitig sichtbar sein.** Genau diese Entkopplung macht Splits so mächtig – du kannst denselben Inhalt parallel an verschiedenen Stellen betrachten und bearbeiten.

Typische Szenarien:

- oben der Beginn einer Funktion, unten das Ende derselben Funktion,
- links die Testdatei, rechts derselbe Buffer an anderer Stelle,
- Vergleich von zwei weit auseinanderliegenden Stellen im Log.

11.4.1. Buffer in neuem Fenster öffnen

Ein bereits vorhandener Buffer lässt sich in einem neuen Split anzeigen:

- `:sb <Nummer>` öffnet Buffer `<Nummer>` in einem *horizontalen* Split.
- `:vert sb <Nummer>` öffnet denselben Buffer in einem *vertikalen* Split.

Die Nummer des Buffers erhältst du zum Beispiel mit:

- `:ls` – listet alle Buffer mit Nummer, Status und Dateinamen,
- `:buffers` – alternative Darstellung derselben Information.

Wichtig ist: Der ursprüngliche Buffer bleibt unverändert bestehen. Du erzeugst lediglich ein weiteres Fenster, das denselben Buffer aus einer anderen Perspektive zeigt.

Praxis-Hinweise.

- Änderungen am Buffer wirken *sofort* in allen Fenstern, die diesen Buffer anzeigen – es gibt nur eine Kopie im Speicher.
- Die Cursorposition ist fensterabhängig: jedes Fenster hat seine *eigene* Position im Buffer.

- Für „mal eben“ denselben Buffer in einem neuen Split reicht auch: `:split` gefolgt von `:b <Nummer>` – ist funktional identisch zu `:sb <Nummer>`, nur etwas expliziter.

Übungen

1. Öffne eine Datei mit `:edit datei.txt` und erzeuge einen vertikalen Split mit demselben Buffer:
   - `:vsplit`
   - `:ls` – Buffer-Nummer merken
   - `:sb <Nummer>`

   Scrolle in beiden Fenstern unabhängig und beobachte, dass der Inhalt identisch bleibt, aber die Cursorposition pro Fenster unterschiedlich ist.
2. Erzeuge zwei horizontale Splits mit demselben Buffer:
   - `:split`
   - `:sb <Nummer>`

   Navigiere oben an den Dateianfang (**g** **g**) und unten ans Ende (**G**). Führe nun eine Änderung im oberen Fenster durch (z. B. füge eine Zeile ein) und beobachte die Auswirkung im unteren Fenster.
3. Wiederhole das obige Szenario, diesmal mit zwei *verschiedenen* Buffern: Erzeuge einen Split mit `:split` und lade dort eine andere Datei mit `:edit andere.txt`. Ziel: Bewusst unterscheiden zwischen
   - „ein Buffer in mehreren Fenstern“ und
   - „jedes Fenster zeigt einen anderen Buffer“.

11.4.2. Zwischen Buffern im gleichen Fenster wechseln

Auch ohne Splits ist das saubere Wechseln zwischen Buffern wichtig – vor allem, wenn mehrere Dateien parallel geöffnet sind.

- `:bnext` / `:bprev` springt zum nächsten bzw. vorherigen Buffer in der internen Bufferliste. Entspricht den Kurzformen `:bn` und `:bp`.
- `:buffer <Nummer>` wechselt direkt zu einem bestimmten Buffer anhand seiner Nummer, wie sie in `:ls` oder `:buffers` angezeigt wird.
- `:b <Name-Fragment>` erlaubt das Anspringen eines Buffers anhand eines eindeutigen Namensfragments (Tab-Completion mit **<Tab>** nutzbar).

Diese Befehle betreffen immer nur *das aktuelle Fenster*: Der angezeigte Buffer wird ausgetauscht, die Anzahl der Fenster bleibt gleich.

Übungen

1. Öffne drei Dateien, z. B.:
   - `:edit alpha.txt`
   - `:edit beta.txt`
   - `:edit gamma.txt`

   Lass dir mit `:ls` die Bufferliste anzeigen und wechsle dann mit `:bnext` und `:bprev` vor- und zurück. Beobachte, wie sich nur der angezeigte Buffer ändert, nicht aber die Fensteranzahl.
2. Springe gezielt zu *beta.txt*, einmal über die Nummer und einmal über den Namen:
   - `:buffer <Nummer-aus-ls>`
   - `:b beta<Tab>`

   Ziel: Unterschied zwischen numerischem Sprung und Namenssprung verinnerlichen.
3. Erzeuge testweise einen Split mit `:vsplit` und wechsle in *jedem* Fenster separat den Buffer mit `:bnext`. Kontrolliere, dass Fenster und Buffer wirklich getrennte Konzepte sind: Jedes Fenster „lebt“ auf seinem eigenen Buffer, auch wenn alle aus derselben Bufferliste stammen.

11.4.3. Buffer in mehreren Fenstern gleichzeitig

Ein großer Vorteil des Fenster-/Buffer-Modells in VIM ist, dass *derselbe* Buffer in mehreren Fenstern parallel sichtbar sein kann. So kannst du verschiedene Stellen derselben Datei vergleichen, reviewen oder gleichzeitig bearbeiten.

Typische Vorgehensweise:

```
:split
:buffer 5
```

- `:split` erzeugt ein neues horizontales Fenster.
- `:buffer 5` lädt in diesem Fenster den Buffer mit der Nummer `5`.

Oder vertikal:

```
:vsplit
:buffer main.c
```

- `:vsplit` teilt das aktuelle Fenster vertikal.

- `:buffer main.c` zeigt denselben Buffer `main.c` auch im neuen Fenster.

Beide Fenster zeigen nun *denselben* Buffer, können aber an unterschiedlichen Positionen im Text stehen. Änderungen in einem Fenster sind sofort im anderen sichtbar – es gibt nur *eine* Datei im Speicher, aber mehrere „Blicke“ darauf.

Übungen

1. Öffne eine Datei mit einigen hundert Zeilen, z. B. ein Log oder Quelltext. Erzeuge dann:
   - einen horizontalen Split mit `:split`
   - und im neuen Fenster denselben Buffer noch einmal mit `:buffer \%`

   Scrolle die Fenster bewusst an unterschiedliche Stellen und beobachte, dass Änderungen in einem Fenster sofort im anderen erscheinen.
2. Öffne dieselbe Datei erneut mit einem vertikalen Split:
   - `:vsplit`

   Nutze **Ctrl**-**w** **h** und **Ctrl**-**w** **l**, um zwischen linkem und rechtem Fenster zu wechseln, und kontrolliere, dass überall derselbe Buffer aktiv ist (`:echo bufname(\%)`).
3. Erzeuge ein Setup mit drei Fenstern:
   - oben: Funktionsübersicht (z. B. mit `:setlocal number`)
   - links unten: Implementierungsteil A
   - rechts unten: Implementierungsteil B

   Nutze dafür Kombinationen aus `:split`, `:vsplit` und `:buffer \%`. Ziel: Die Trennung „Fenster = Sicht“, „Buffer = Inhalt“ verinnerlichen.

11.4.4. Unabhängige Cursorpositionen

Wichtig: **Jedes Fenster speichert seine eigene Cursorposition, Ansicht und Scroll-Region** — selbst dann, wenn alle dasselbe Buffer-Objekt anzeigen.

Das ermöglicht Setups wie:

- oben: Funktionsdefinition
- unten: Aufruf derselben Funktion
- links: Tests
- rechts: Log-Ausgabe

ohne dass du ständig hin- und herspringen oder dir Marker merken musst.

Typische Situationen, in denen die unabhängigen Positionen nützlich sind:

- Du editierst oben eine Funktion und beobachtest unten gleichzeitig, wie sich der Aufrufkontext verändert.
- Du hältst in einem Fenster die Definition einer Datenstruktur offen, während du in einem zweiten Fenster ihre Verwendung an verschiedenen Stellen verfolgst.
- Du scrollst in einem Fenster schnell durch die Datei, während ein anderes Fenster als „Anker“ an einer wichtigen Stelle stehen bleibt.

Der Wechsel zwischen den Fenstern erfolgt wie gewohnt mit:

- **Ctrl**-**w** **h**, **Ctrl**-**w** **j**, **Ctrl**-**w** **k**, **Ctrl**-**w** **l** — Wechsel in die Nachbarfenster nach links/unten/oben/rechts
- **Ctrl**-**w** **w** — zyklisch durch alle Fenster
- **Ctrl**-**w** **p** — zurück zum zuletzt aktiven Fenster

Jedes Fenster merkt sich dabei:

- seine eigene Cursorposition,
- die aktuelle Scroll-Position,
- lokale Anzeigeoptionen (z. B. `:setlocal number`, `:setlocal relativenumber`).

Übungen

1. Öffne eine längere Datei, z. B. Quelltext oder ein Logfile. Erzeuge eine horizontale Teilung mit `:split`. Scrolle das obere Fenster an den Anfang der Datei und das untere Fenster ans Ende. Wechsle mit **Ctrl**-**w** **w** hin und her und beobachte, dass jede Ansicht ihre eigene Position behält.
2. Erzeuge eine vertikale Teilung mit `:vsplit`. Navigiere im linken Fenster zu einer Funktionsdefinition, im rechten Fenster zu einem ihrer Aufrufe. Bearbeite die Funktion links und speichere mit `:w`. Kontrolliere rechts, wie sich der Kontext darstellt, ohne die Position zu verlieren.
3. Aktiviere in nur *einem* Fenster Zeilennummern mit `:setlocal number` und belasse das andere Fenster ohne Nummern. Scrolle und wechsle mehrfach mit **Ctrl**-**w** **w**. Ziel: Verinnerlichen, dass Anzeigeoptionen und Cursorpositionen pro Fenster gelten, nicht pro Buffer.

11.4.5. Bufferliste anzeigen

Zur Übersicht über alle geöffneten Buffer dienen die Kommandos:

`:ls`

oder ausführlicher:

`:buffers`

Die angezeigten Nummern lassen sich direkt mit `:buffer <Nummer>` oder `:sb <Nummer>` nutzen.

Typische Anzeige (Beispiel):

```
 1 %a   "main.c"          line 120
 2 #    "util.c"          line  45
 3      "README.md"       line   3
```

Bedeutung der Spalten:

- erste Spalte: Buffer-Nummer (wird in `:buffer`, `:sb` usw. verwendet)
- `%`: aktueller Buffer
- `#`: alternativer Buffer (der „vorherige“)
- `a`: Buffer ist in einem Fenster *aktiv* angezeigt
- Dateiname: logischer Name des Buffers
- `line <n>`: aktuelle Cursorzeile in diesem Buffer

Praktische Folgebefehle:

- `:buffer 2` springt direkt zu Buffer 2
- `:sb 3` öffnet Buffer 3 in einem neuen Split
- `:b\#` springt zum alternativen Buffer (mit # markiert)

Übungen

1. Öffne drei verschiedene Dateien in VIM (z. B. `:edit main.c`, `:edit util.c`, `:edit README.md`). Rufe anschließend `:ls` auf und identifiziere:
   - den aktuellen Buffer (`%`),
   - den alternativen Buffer (`#`),
   - alle Buffer mit `a`-Markierung.
2. Wechsle mit `:buffer <Nummer>` gezielt zwischen den Buffern hin und her. Nutze dann `:b\#`, um nachvollziehbar zwischen aktuellem und alternativem Buffer zu pendeln.

3. Öffne einen horizontalen Split mit `:split` und hole dir mit `:sb <Nummer>` einen anderen Buffer in das neue Fenster. Ziel: In beiden Fenstern unterschiedliche Buffer anzeigen, aber mit `:ls` jederzeit die Übersicht behalten.

11.4.6. Buffer sauber geschlossen halten

Damit Splits und Bufferlisten übersichtlich bleiben, sollten nicht mehr benötigte Buffer gezielt entfernt werden.

`:bd` (`:bdelete`) Entfernt den aktuellen Buffer aus dem Arbeitsspeicher, lässt aber das Fenster bestehen. Das Fenster zeigt danach:
- entweder den nächsten verfügbaren Buffer,
- oder bleibt leer, falls keine weiteren Buffer existieren.

Splits bleiben also erhalten – nur der Inhalt (Buffer) wechselt.

`:bwipeout` Entfernt den Buffer vollständig aus der internen Liste. Im Gegensatz zu `:bd` verschwinden dabei auch alle Spuren des Buffers aus `:ls` / `:buffers` (z. B. hidden-Einträge).

Typische Verwendung:
- `:bd` für temporär geöffnete Dateien, die im aktuellen Projekt keine Rolle mehr spielen (z. B. kurz geöffnete Logfiles).
- `:bwipeout` für veraltete oder versehentlich erzeugte Buffer, die nicht einmal mehr in der Liste auftauchen sollen.

Übungen

1. Öffne drei Dateien (z. B. `:edit a.txt`, `:edit b.txt`, `:edit c.txt`) und rufe `:ls` auf. Schließe dann den aktuellen Buffer mit `:bd` und prüfe erneut mit `:ls`, wie sich die Anzeige verändert hat.
2. Öffne eine Datei, wechsle in einen anderen Buffer und rufe `:ls` auf. Identifiziere einen hidden-Buffer (ohne `a`-Markierung) und entferne ihn mit `:bwipeout <Nummer>`. Kontrolliere, dass er anschließend aus der Liste verschwunden ist.
3. Baue dir eine kleine Routine:
   - `:ls` anzeigen,
   - mit `:buffer <Nummer>` zu einem alten Buffer springen,
   - entscheiden: behalten oder löschen,
   - entsprechend `:bd` oder `:bwipeout` ausführen.

Ziel: Eine aufgeräumte Bufferliste ohne Altlasten.

11.5. Terminal-Splits

VIM (und insbesondere Neovim) erlaubt das Öffnen eines interaktiven Terminals direkt in einem Split-Fenster. Damit lassen sich Build-Tools, Shell-Kommandos oder Skripte ausführen, ohne den Editor zu verlassen.

Typische Einsatzszenarien:

- Tests oder `:make`-Läufe direkt unter dem Quelltext beobachten
- Logs tailen, während man Konfigurationsdateien bearbeitet
- Git-Kommandos, Linter oder Formatierer ausführen

11.5.1. Terminal starten

Ein Terminal kann auf mehrere Arten geöffnet werden (Vim 8+ und Neovim):

- `:terminal` Terminal im aktuellen Fenster starten.
- `:split \textbar {} terminal` Terminal in einem *horizontalen* Split unterhalb des aktuellen Fensters öffnen.
- `:vsplit \textbar {} terminal` Terminal in einem *vertikalen* Split rechts vom aktuellen Fenster öffnen.

In Neovim sind zusätzlich Kurzformen üblich:

- `:term` (entspricht grob `:terminal`)
- `:vert term` vertikaler Split mit Terminal

11.5.2. Modi im Terminalfenster

Ein Terminal-Fenster verhält sich anders als ein normaler Vim-Buffer:

- Im *Terminal-Job-Modus* werden Tasten direkt an die Shell weitergereicht (so wie in einem normalen Terminal-Emulator).
- Im *Normal Mode* steuert man wieder Vim selbst (Bewegungen, **y**, **p**, **/**, `:q` usw.).

Der Wechsel zwischen diesen Welten hängt von Vim/Neovim ab:

Vim (klassisch)

- Von Terminal-Job-Modus nach Normal Mode: **Ctrl**-**w** **N** (damit betrittst du den Terminal-Normal-Mode).
- Zurück in die Shell: **i** oder **a** im Terminal-Normal-Mode.

Neovim
- Von Terminal-Mode nach Normal Mode: Ctrl - \ Ctrl - n (Control-Backslash, dann Control-n).
- Zurück in die Shell: i, a, o wie gewohnt.

11.5.3. Terminal schließen

Ein Terminal-Split kann auf zwei Arten beendet werden:

- Shell regulär verlassen (z. B. `exit` eintippen und mit Enter bestätigen) – der Terminalprozess endet, das Fenster bleibt ggf. als leerer Buffer bestehen.
- Fenster schließen, z. B. mit Ctrl - w c oder `:close`. Damit verschwindet der Split, der Terminalprozess wird beendet.

11.5.4. Typische Workflows

- **Build und Test im Split:**
  1. Quelltext-Datei öffnen.
  2. `:vsplit \textbar {} terminal` ausführen.
  3. Im Terminal `make test` oder `pytest` laufen lassen.
  4. Mit Ctrl - w h / Ctrl - w l zwischen Code und Terminal pendeln.
- **Logs beobachten:**
  1. Konfigurationsdatei bearbeiten.
  2. Unten einen Split mit `:split \textbar {} terminal` öffnen.
  3. Im Terminal `tail -f logfile.log` ausführen.
  4. Änderungen in der Config speichern, im Terminal sofort die Wirkung sehen.
- **Git-Status und Diff:**
  1. Oben Quelltext, unten Terminal-Split.
  2. Im Terminal `git status`, `git diff` usw.
  3. Per Ctrl - w w zwischen Editor und Terminal hin- und herspringen.

Übungen

1. Öffne eine Datei und starte einen vertikalen Terminal-Split mit `:vsplit \textbar {} terminal`. Führe im Terminal ein einfaches Kommando wie `ls` aus und wechsle mit Ctrl - w h zurück in das Editorfenster.
2. In Neovim: Öffne ein Terminal mit `:term`, wechsle in den Normal Mode mit Ctrl - \ Ctrl - n und bewege den Cursor mit h, j, k, l durch die

Terminal-History. Kehre anschließend mit [i] wieder in den Terminal-Mode zurück.

3. Starte oben eine Quelltextdatei, unten ein Terminal mit `:split \textbar {} terminal`. Lasse im Terminal einen Langläufer laufen (z. B. `ping localhost`) und bearbeite gleichzeitig oben den Text. Ziel: Flüssig zwischen Editor ([Ctrl]-[w][k]) und Terminal ([Ctrl]-[w][j]) wechseln, ohne den Fokus zu verlieren.

11.5.5. Modi: Terminal-Insert- und Normalmodus

Nach dem Öffnen startet das Terminal im *Terminal-Insert-Modus*: Tastendrücke werden direkt an die Shell weitergereicht – wie in einem gewöhnlichen Terminal-Emulator.

Um in den normalen VIM-Modus zu wechseln (abhängig von der Version):

- Vim 8: [Ctrl]-[w][N] – schaltet in den Terminal-Normal-Mode.
- Neovim: [Ctrl]-[\][Ctrl]-[n] – wechselt vom Terminal-Mode zurück in den Normalmodus.

Im Terminal-Normal-Mode gelten wieder die üblichen VIM-Befehle: Bewegungen ([h], [j], [k], [l]), Suchen, [y]/[p], `:q` usw. – du steuerst also *VIM*, nicht mehr die Shell.

Zurück in den Terminal-Insert-Modus:

- [i], [a] oder [o] – kehren aus dem Normalmodus in den Terminal-Insert-Modus zurück; ab dann geht alles wieder direkt an die Shell.

Übungen

1. Öffne ein Terminal im Split (`:split \textbar {} terminal` oder `:vsplit \textbar {} terminal`). Wechsle mehrfach zwischen Terminal-Insert- und Normalmodus hin und her, bis sich die Tastenkombinationen automatisiert anfühlen.
2. Gehe in den Terminal-Normal-Mode, navigiere mit [h], [j], [k], [l] durch die History und kopiere eine Zeile mit [y][y]. Wechsle in ein anderes Fenster und füge den Inhalt mit [p] in einen normalen Textbuffer ein.

11.5.6. Fenstersteuerung

Terminal-Splits verhalten sich bei der Fenstersteuerung exakt wie normale Splits. Wichtig ist nur: Die Ctrl-w-Kombinationen funktionieren immer im Normal- bzw. Terminal-Normal-Mode, nicht im reinen Terminal-Insert-Modus.

- Ctrl-w h, Ctrl-w j, Ctrl-w k, Ctrl-w l – in das linke / untere / obere / rechte Fenster wechseln.
- Ctrl-w c – aktuelles Terminal-Fenster schließen (Fenster wird beendet, andere Splits bleiben erhalten).
- Ctrl-w = – alle Fenster auf gleiche Größe bringen.

Typischer Ablauf im Alltag:

1. Terminal im Split öffnen (z. B. `:vsplit \textbar {} terminal`).
2. Mit Ctrl-w h oder Ctrl-w l zwischen Code- und Terminal-Fenster wechseln.
3. Mit Ctrl-w = die Fenster wieder sauber ausrichten.

Übungen

1. Erzeuge zwei Splits: links Quelltext, rechts Terminal. Wechsle für einige Minuten ausschließlich mit Ctrl-w h/Ctrl-w l hin und her, bis die Kombination *ohne Nachdenken* sitzt.
2. Öffne drei oder mehr Fenster, ändere ihre Größen beliebig mit Ctrl-w +, Ctrl-w -, Ctrl-w >, Ctrl-w <. Stelle anschließend mit Ctrl-w = wieder eine saubere, gleichmäßige Verteilung her.
3. Starte ein Terminal im unteren Split, führe dort ein Build- oder Testkommando aus und springe mit Ctrl-w k/Ctrl-w j zwischen Log-Ausgabe und Quelltext hin und her, ohne die Maus zu benutzen.

11.5.7. Scrollback und Suche

Im Terminal-Split ist der Bildschirminhalt ein ganz normaler VIM-Puffer. Um darin zu blättern oder zu suchen, musst du zuerst aus dem Terminal-Insert-Modus zurück in den Normalmodus wechseln:

- Ctrl-\ Ctrl-n! – in den Terminal-Normal-Mode wechseln

Danach stehen die gewohnten Navigations- und Suchbefehle zur Verfügung:

- g g – an den Anfang der Terminal-History springen

- G – ans Ende (zur aktuellen Eingabe) springen
- / pattern Enter – nach `pattern` suchen
- Ctrl - f , Ctrl - b – seitenweise vor/zurück blättern

Wichtig ist die Trennung:

- Im Terminal-Insert-Modus „gehört“ die Eingabe der Shell (oder dem jeweiligen Programm).
- Im Terminal-Normal-Mode steuert VIM den Scrollback – du bewegst dich im bereits ausgegebenen Text.

Übungen

1. Öffne einen Terminal-Split und lasse ein Kommando mit viel Ausgabe laufen (z. B. `ls -R` in einem größeren Verzeichnis). Wechsle mit Ctrl - \ Ctrl - n ! in den Normalmodus und blättere mit Ctrl - f und Ctrl - b durch die History.
2. Suche im Terminal-Output nach einem bestimmten Dateinamen oder Log-Tag (z. B. ERROR) mit / ERROR Enter . Springe mit n und N durch die Treffer.
3. Wechsle mehrfach zwischen Code-Fenster und Terminal-Split:
   a) Führe im Terminal ein Build-Kommando aus.
   b) Scrolle im Terminal hoch zur ersten Fehlermeldung.
   c) Springe mit Ctrl - w h / Ctrl - w l zurück ins Code-Fenster und korrigiere die Stelle.

   Ziel: Scrollback, Suche und Fensterwechsel ohne Maus verinnerlichen.

11.5.8. Terminal schließen

Ein Terminal-Split lässt sich auf unterschiedliche Weise „aufräumen“ – je nach gewünschter Wirkung:

- Ctrl - d — beendet die Shell im Terminal Wirkt wie in einer normalen POSIX-Shell: Wenn kein Programm mehr läuft und nur die Eingabeaufforderung sichtbar ist, schließt Ctrl - d die Shell. Der Terminal-Job endet, das Fenster bleibt zunächst bestehen, bis VIM den leeren Terminal-Buffer räumt.
- `:bd` — Terminal-Buffer schließen Entfernt den Terminal-Buffer aus der aktuellen Ansicht (und ggf. auch aus der Bufferliste). Das Fenster selbst bleibt erhalten und kann anschließend einen anderen Buffer anzeigen.

- **Ctrl** - **w** **c** — Split schließen Schließt das aktuelle Fenster (den Split). Andere Fenster bleiben unverändert. Der zugrunde liegende Buffer kann dabei weiter existieren (z. B. wenn er in einem anderen Fenster noch geöffnet ist).

Faustregeln:

- **Ctrl** - **d**: „Programm/Session beenden“ (im Terminal selbst).
- `:bd`: „Diesen Terminal-Buffer aus VIM entfernen“.
- **Ctrl** - **w** **c**: „Nur dieses Fenster loswerden“.

Übungen

1. Öffne einen Terminal-Split (`:term`) und starte ein lang laufendes Kommando (z. B. `ping localhost`). Probiere aus, wie sich **Ctrl** - **d** verhält, während das Programm läuft, und was passiert, wenn du vorher das Programm mit **c** **Ctrl** - **c** abbrichst.
2. Öffne zwei Terminal-Splits nebeneinander. Beende im linken Split die Shell mit **Ctrl** - **d**, im rechten Split schließe stattdessen das Fenster mit **Ctrl** - **w** **c**. Beobachte den Unterschied in der Bufferliste mit `:ls`.
3. Starte ein Terminal, führe darin ein Build- oder Testskript aus und schließe den Terminal-Buffer anschließend mit `:bd`. Wechsle dann zurück in dein Codefenster und prüfe mit `:ls`, welche Buffer noch aktiv sind.

12. Datei und Projekt

12.1. Speichern

Speichern gehört zu den grundlegendsten Operationen in VIM, ist aber flexibler als in klassischen Editoren. VIM unterscheidet zwischen *Schreiben* (Datei auf Disk aktualisieren) und *Benennen* (neue Datei anlegen oder Pfad ändern). Alle folgenden Befehle werden im Normal Mode mit einem führenden Doppelpunkt `:` als Ex-Kommando eingegeben.

12.1.1. Grundbefehle

- `:w` Schreibt den aktuellen Buffer auf die zugehörige Datei. Existiert die Datei noch nicht, wird sie neu angelegt.
- `:wq` Schreibt die Datei und schließt anschließend das aktuelle Fenster. Praktischer „Alles-in-einem"-Befehl nach einer abgeschlossenen Änderung.
- `:x` Funktioniert wie `:wq`, speichert aber nur, wenn wirklich Änderungen vorhanden sind. Bei unverändertem Buffer wird nur geschlossen.
- `:q` Versucht, das aktuelle Fenster zu schließen. Wenn ungespeicherte Änderungen vorhanden sind, verweigert VIM den Befehl mit einer Warnung.
- `:q!` Schließt das aktuelle Fenster und verwirft alle ungespeicherten Änderungen im aktuellen Buffer.
- `:qa` Schließt alle Fenster der aktuellen VIM-Sitzung. Ungespeicherte Änderungen verhindern den Befehl (analog zu `:q`).
- `:w \emph {Dateiname}` Schreibt den aktuellen Buffer unter einem *neuen* Namen. Der Buffer ist danach an *diesen* Dateinamen gebunden (künftige `:w`-Befehle verwenden den neuen Namen).

Übungen

1. Öffne eine Testdatei, nimm eine sichtbare Änderung vor und probiere:
   - `:w`
   - `:q`

   - `:q!`

   Beobachte, wann VIM eine Warnung ausgibt und wann nicht.
2. Erzeuge eine neue Datei direkt aus VIM heraus:
   a) Starte VIM ohne Dateinamen (z. B. `vim` im Terminal).
   b) Schreibe ein paar Zeilen Text.
   c) Speichere die Datei mit `:w testdatei.txt`.
   d) Beende VIM anschließend mit `:x`.
3. Öffne mehrere Dateien gleichzeitig (z. B. mit `vim datei1 datei2`) und beende die Sitzung mit:
   - `:q` im aktuellen Fenster
   - `:qa`, um alle Fenster auf einmal zu schließen

   Prüfe, wie sich VIM bei geänderten und ungeänderten Buffern verhält.

12.1.2. Speichern trotz Schreibschutz

Wenn eine Datei schreibgeschützt ist, muss man unterscheiden zwischen:

1. **VIM-internem Schreibschutz** (`readonly`-Option gesetzt),
2. **Schreibschutz auf Dateisystem-Ebene** (keine Rechte, z. B. nur root).

VIM-interner Schreibschutz

Ist nur in VIM die `readonly`-Option aktiv (z. B. nach `vim -R`):

- `:w!` Ignoriert den VIM-internen `readonly`-Status und schreibt die Datei trotzdem, sofern das Dateisystem die Rechte dafür erlaubt.
- `:set noreadonly` Hebt den internen Schreibschutz auf. Danach genügt wieder ein normales `:w`.

Typische Fehlermeldung in diesem Fall ist z. B. `E45: 'readonly' option is set`.

Schreibschutz auf Dateisystem-Ebene

Wenn das Dateisystem das Schreiben verbietet (z. B. Datei gehört `root`, nur Lese-Rechte, schreibbares Verzeichnis fehlt), hilft `:w!` allein nicht. Dann muss mit erhöhten Rechten gearbeitet oder an einen anderen Ort geschrieben werden.

Ein gängiges Muster unter Unix-Systemen ist:

```
:w !sudo tee \% >/dev/null
```

Erklärung:

- `:w !` — schreibt den aktuellen Buffer auf `stdin` eines externen Befehls,
- `sudo tee %` — schreibt den Inhalt mit `sudo`-Rechten in genau die Datei, die derzeit im Buffer geöffnet ist,
- `>/dev/null` — unterdrückt die doppelte Ausgabe von `tee` im VIM-Fenster (optional).

Alternativ kann man den Inhalt zunächst in eine Datei schreiben, auf die man Schreibrechte hat, und diese später mit Systemmitteln (z. B. `sudo mv`) an den Zielort verschieben.

Hinweis zu `:w!!`

In VIM hat `:w!!` *nicht* die Bedeutung „noch härter erzwingen“, sondern: „wiederhole den letzten `:w !`...-Befehl mit demselben externen Kommando“. Es ist also ein Komfortkürzel für bestehende Filter, kein Schreibschutz-Bypass.

Übungen

1. Öffne eine Datei und setze sie in VIM auf `readonly`:
   - `:set readonly`
   - nimm eine kleine Änderung vor
   - versuche zu speichern mit `:w` und beobachte die Fehlermeldung
   - speichere anschließend mit `:w!`
   - hebe den Status wieder auf mit `:set noreadonly`
2. Öffne eine Datei, die dir normal gehört, und simuliere einen „fremden“ Schreibvorgang:
   a) nimm Änderungen vor,
   b) führe `:w !tee /tmp/vim-test.txt` aus,
   c) prüfe, dass der Inhalt in `/tmp/vim-test.txt` geschrieben wurde.

   Überlege: wie müsste das Kommando aussehen, wenn `sudo` nötig wäre?

12.1.3. Speichern in nicht existierende Verzeichnisse

VIM erzeugt *keine* Verzeichnisse automatisch. Versucht man, in einen Pfad zu speichern, dessen Verzeichnisstruktur nicht existiert, erscheint typischerweise:

```
E212: Can't open file for writing
```

In solchen Fällen muss zunächst das Zielverzeichnis außerhalb des eigentlichen Speicherbefehls angelegt werden, z. B. über einen Shell-Aufruf:

```
:!mkdir -p path/to/dir
:w path/to/dir/filename.txt
```

Erläuterung:

- `:!mkdir -p path/to/dir` ruft das externe Programm `mkdir` auf und legt den kompletten Pfad `path/to/dir` an (`-p` erzeugt fehlende Zwischenverzeichnisse).
- `:w path/to/dir/filename.txt` speichert den aktuellen Buffer anschließend in die nun existierende Datei.

Typische Stolpersteine:

- Schreibversuche in nicht beschreibbare Verzeichnisse (z. B. `/etc`) schlagen zusätzlich an fehlenden Rechten fehl – hier ist ggf. `sudo` nötig (vgl. vorherigen Abschnitt).
- Relative Pfade beziehen sich auf das aktuelle Arbeitsverzeichnis von VIM (`:pwd` zeigt es an). Gegebenenfalls zuerst mit `:cd` ins gewünschte Projektverzeichnis wechseln.

Übungen

1. Öffne einen neuen Buffer und versuche, direkt in ein nicht existierendes Unterverzeichnis zu schreiben, z. B.:
   - `:w tmp/subdir/test.txt`

   Notiere dir die genaue Fehlermeldung (`E212`).
2. Erzeuge anschließend das Verzeichnis aus VIM heraus:
   - `:!mkdir -p tmp/subdir`
   - `:w tmp/subdir/test.txt`

   Prüfe mit `:!ls tmp/subdir`, ob die Datei tatsächlich angelegt wurde.
3. Variiere den Pfad mit absoluter Schreibweise (`/tmp/...`) und beobachte, wie sich das Fehlerverhalten ändert, wenn das Verzeichnis existiert bzw. nicht existiert.

12.1.4. Automatisches Speichern

VIM speichert niemals ungefragt automatisch – *es sei denn*, man schaltet diese Funktion bewusst ein. Zwei Optionen steuern das Verhalten:

- `:set autowrite` speichert geänderte Buffer automatisch vor bestimmten Aktionen, z. B. bei `:next`, `:make` oder beim Laden einer anderen Datei.
- `:set autowriteall` speichert *alle* geänderten Buffer an noch mehr Stellen automatisch (z. B. vor `:quit`, `:split`, diversen Plugin-Kommandos).

Praktische Varianten zur Kontrolle:

- `:set noautowrite` / `:set noautowriteall` – automatische Saves abschalten.
- `:set autowrite?` – aktuellen Status der Option anzeigen.

Wichtig: Diese Optionen ersetzen kein bewusstes Speichern, sondern reduzieren nur die Anzahl der „Vergiss-Speichern"-Fälle im Alltag.

Manuelles Speichern aller Buffer Unabhängig von `autowrite` gibt es Befehle zum expliziten Speichern *aller* geänderten Buffer:

- `:wa` – alle geänderten Buffer speichern („write all").
- `:wall` – wie `:wa`, bricht aber weniger schnell ab und versucht aggressiver, alle Dateien zu schreiben (z. B. bei mehreren Fenstern/Tabs).

Diese Befehle sind besonders nützlich:

- vor größeren Refactorings oder globalen Ersetzungen,
- vor dem Start eines Build-Laufs (`:make`, externe Tools),
- wenn viele Dateien parallel geöffnet sind und man einen definierten „Save-Punkt" setzen will.

Übungen

1. Öffne drei verschiedene Dateien (z. B. `a.txt`, `b.txt`, `c.txt`) in Buffern, ändere in jeder Datei eine Zeile und führe dann:
   - `:wa`

   aus. Prüfe anschließend mit `:ls`, welche Buffer noch als verändert (+) markiert sind.
2. Deaktiviere automatisches Speichern explizit mit:
   - `:set noautowrite`

   Navigiere mit `:bnext` / `:bprev` durch mehrere geänderte Buffer und beobachte, dass VIM dich wieder auf ungespeicherte Änderungen hinweist.
3. Aktiviere nun:
   - `:set autowriteall`

Führe danach typische Kommandos wie `:make`, `:next` oder `:quit` aus und überprüfe mit `:ls`, welche Dateien automatisch gespeichert wurden. Ziel: Ein Gefühl dafür bekommen, wann VIM *von selbst* speichert – und wann nicht.

12.1.5. Wechselnde Dateiformate / Encoding

VIM unterscheidet zwei Aspekte beim Speichern einer Datei:

- *Encoding* (`encoding` / `fileencoding`) – also die Zeichencodierung, z. B. `utf-8`, `latin1`, `cp1252`.
- *Fileformat* (`fileformat`) – also die Art der Zeilenenden, z. B. `unix` (LF), `dos` (CRLF), `mac` (altes Mac-Format).

Beides lässt sich beim Speichern gezielt steuern.

Einmaliges Speichern mit anderem Encoding

- `:w ++enc=utf-8`

Schreibt die aktuelle Datei *nur für diesen Speichervorgang* in `utf-8`, ohne die Option `fileencoding` dauerhaft zu ändern. Nützlich, wenn eine Datei ausnahmsweise in einem anderen Encoding benötigt wird (z. B. Export).

Einmaliges Speichern mit anderem Zeilenformat

- `:w ++ff=dos`

Speichert die Datei einmalig mit „DOS-Line-Endings“ (CRLF). Die interne Darstellung bleibt davon unberührt; VIM konvertiert beim Schreiben.

Dauerhafte Umstellung für den Buffer Statt jedes Mal `++enc` bzw. `++ff` anzugeben, können die Buffer-Optionen gesetzt werden:

- `:set fileencoding=utf-8` – zukünftige Writes in `utf-8`.
- `:set fileformat=unix` – zukünftige Writes mit „Unix-Line-Endings“.

Der aktuelle Zustand lässt sich prüfen mit:

- `:set fileencoding?`
- `:set fileformat?`

Wichtige Hinweise

- Die Konvertierung von Encodings kann *irreversible* Verluste erzeugen, wenn Zeichen im Ziel-Encoding nicht darstellbar sind. Im Zweifel immer eine Sicherungskopie anlegen.
- `fileencoding` wirkt auf den aktuellen Buffer – unterschiedliche Buffer können unterschiedliche Encodings haben.
- `encoding` ist die interne VIM-Codierung und wird meist global auf `utf-8` gesetzt; sie sollte selten im Alltag geändert werden.

Übungen

1. Öffne eine Testdatei und prüfe zunächst:
   - `:set fileencoding?`
   - `:set fileformat?`

   Notiere dir die Werte.
2. Speichere dieselbe Datei einmalig mit DOS-Line-Endings:
   - `:w ++ff=dos`

   Lade die Datei neu mit `:edit` und prüfe erneut `:set fileformat?`. Ziel: Verstehen, dass sich das Zeilenformat sichtbar ändert.
3. Stelle nun den Buffer dauerhaft auf Unix-Line-Endings um:
   - `:set fileformat=unix`
   - `:w`

   Öffne die Datei in einem externen Tool (z. B. `file` oder einem Hex-Viewer) und kontrolliere, dass nur noch LF verwendet wird.
4. Erzeuge eine Datei mit Umlauten und Sonderzeichen (ä, ö, ü, ß, €). Speichere sie einmal mit `:set fileencoding=utf-8` und `:w`, danach probeweise mit:
   - `:w ++enc=latin1`

   Öffne die latin1-Variante in einem anderen Editor und beobachte, wie die Zeichen dargestellt werden. Ziel: Gefühl für mögliche Encoding-Probleme und Datenverlust entwickeln.

12.1.6. Backup- und Swap-Dateien

VIM verwendet mehrere Schutzmechanismen, um Datenverlust zu verhindern. Drei zentrale Optionen sind:

- `:set backup` aktiviert Backup-Dateien (meist mit Suffix ~). Beim Speichern wird die *alte* Version der Datei als Kopie behalten.

- `:set writebackup` erzeugt nur während des Schreibvorgangs eine Sicherungskopie. Bei erfolgreichem Schreiben wird diese Kopie wieder entfernt.
- `:set swapfile` erzeugt und nutzt Swap-Dateien (meist `.swp`), um den aktuellen Bearbeitungszustand zwischendurch zu sichern.

Typische Kombination in der Praxis:

- `:set backup` – dauerhafte Sicherung der alten Version,
- `:set writebackup` – zusätzlicher Schutz beim Schreiben,
- `:set noswapfile` – nur in Sonderfällen (z. B. auf sehr langsamen oder schreibempfindlichen Medien; im Normalfall *nicht* empfohlen).

Swap-Dateien und Abstürze Swap-Dateien sind der wichtigste Notfallmechanismus: Stürzt VIM oder das System ab, bleibt der letzte Zustand im `.swp`-File erhalten. Beim nächsten Öffnen derselben Datei meldet VIM:

```
E325: ATTENTION
Found a swap file ...
```

In diesem Fall sollte man bewusst entscheiden:

- `:recover` lädt den Inhalt aus der Swap-Datei und versucht, die letzte bearbeitete Version zu rekonstruieren.
- `:edit!` ignoriert die Swap-Datei und lädt die Version von der Platte (Gefahr: ungesicherte Änderungen gehen verloren).
- `:swapname` zeigt den Pfad zur zugehörigen `.swp`-Datei an.

Nach einem erfolgreichen `:recover` sollte die Datei normal gespeichert und die Swap-Datei anschließend gelöscht bzw. von VIM aufgeräumt werden.

Wo landen Backup- und Swap-Dateien? Das Verhalten hängt von folgenden Optionen ab:

- `:set backupdir?` – Verzeichnis für Backup-Dateien,
- `:set directory?` – Verzeichnis für Swap-Dateien.

Standardmäßig liegen sie oft im gleichen Verzeichnis wie die bearbeitete Datei. Für Projekte lohnt sich häufig ein zentrales Verzeichnis (z. B. `~/.vim/swap`), um Arbeitsordner sauber zu halten.

Übungen

1. Aktiviere Backups und speichere eine Testdatei:

   - `:set backup`
   - bearbeite eine Datei und speichere mit `:w`

   Prüfe im Dateisystem, dass neben der eigentlichen Datei eine Backup-Datei mit Suffix ~ existiert.
2. Simuliere eine Swap-Situation:
   - Öffne eine Datei, ändere einige Zeilen *ohne* zu speichern.
   - Beende VIM *nicht sauber*, z. B. durch Schließen des Terminals oder *harten* Abbruch.
   - Öffne dieselbe Datei erneut und beobachte die „E325“-Meldung.
   - Wähle nun `:recover` und speichere anschließend mit `:w`.

   Ziel: Verstehen, wie Swap-Recovery im Ernstfall funktioniert.
3. Prüfe und ändere die Verzeichnisse für Sicherungen:
   - `:set backupdir?`
   - `:set directory?`

   Setze testweise:
   - `:set backupdir\^ =~/vim-backup/`
   - `:set directory\^ =~/vim-swap/`

   Erzeuge diese Verzeichnisse im Dateisystem und bearbeite erneut eine Datei. Kontrolliere, dass Backup- und Swap-Dateien nun dort abgelegt werden.

12.1.7. Speichern und Kommandos kombinieren

Da Ex-Befehle flexibel kombinierbar sind, lassen sich *Speichern* und weitere Aktionen in einem Schritt ausführen. Mehrere Kommandos werden dabei mit dem Pipe-Zeichen | verknüpft:

- `:write | !sh script.sh`
- `:w | !make`
- `:w | edit \%`

Typische Anwendungsfälle:

- *Build nach dem Speichern:*
  - `:w | !make`
- *Tests ausführen:*
  - `:w | !pytest`
- *Datei speichern und neu laden (z.B. nach externen Filtern):*
  - `:w | edit \%`

Wichtig ist die Reihenfolge: Zuerst schreibt `:w` den Buffer auf die Platte, danach wird der Shell-Befehl (`:!make`, `:!pytest`, ...) ausgeführt oder der Buffer neu geladen.

Für häufige Kombinationen lohnt sich ein „speichern & bauen“ mit `:update`:

- `:update | !make`

`:update` speichert nur, wenn der Buffer verändert wurde. Das vermeidet unnötige Schreibvorgänge.

Hinweise

- Jeder Teil vor und nach `|` ist ein *vollwertiger* Ex-Befehl.
- Fehler im ersten Kommando (z. B. fehlende Schreibrechte) verhindern, dass die nachfolgenden Befehle ausgeführt werden.
- `:silent` kann genutzt werden, um Ausgaben zu unterdrücken, wenn nur das Ergebnis des externen Programms interessiert.

Übungen

1. Erzeuge eine kleine C-Datei oder ein Skript mit zugehörigem `Makefile`. Probiere nacheinander:
   - `:w \textbar \ !make`
   - `:update \textbar \ !make`

   Beobachte den Unterschied, wenn du *keine* Änderungen vor dem zweiten Befehl machst.
2. Schreibe eine Datei `test.txt` und führe:
   - `:w \textbar \ edit \%`

   aus. Ändere die Datei extern (z. B. mit einem anderen Editor oder per `sed`) und wiederhole den Befehl. Ziel: Verstehen, wann und wie VIM den Buffer von der Platte neu lädt.
3. Lege dir eine kleine „Ein-Kommando-Build-Pipeline“ an, z. B.:
   - `:command! -nargs=0 WMake update \textbar \ !make`

   Speichere die Definition testweise in einer Session oder deiner Konfiguration und rufe dann `:WMake` in einem Projekt auf. Ziel: Speichern und Bauen als *einen* gedanklichen Schritt etablieren.

12.2. Datei öffnen

Das Öffnen von Dateien ist in VIM flexibler als in klassischen Editoren. VIM lädt Dateien grundsätzlich als *Buffer*, unabhängig davon, ob sie in einem Fenster sichtbar sind. Dadurch lassen sich beliebig viele Dateien parallel öffnen und verwalten, ohne dass jede Datei ein eigenes Fenster haben muss.

12.2.1. Datei direkt öffnen

Die einfachste Variante ist das direkte Öffnen oder Neuladen einer Datei:

- `:edit <Datei>` öffnet die angegebene Datei im aktuellen Fenster und lädt sie in einen Buffer.
- `:e <Datei>` Kurzform von `:edit <Datei>`; funktional identisch, aber schneller zu tippen.
- `:e` lädt die aktuell geöffnete Datei von der Festplatte neu ein (Refresh). Das ist vor allem dann wichtig, wenn die Datei von einem anderen Programm oder Prozess verändert wurde.

Hinweise:

- Ist die Datei bereits als Buffer geöffnet, springt VIM nur in diesen bestehenden Buffer, anstatt sie ein zweites Mal zu laden.
- Mit `:edit!` (`:e!`) kann die Datei neu geladen werden, wobei *lokale, ungespeicherte Änderungen verworfen* werden. Das ist praktisch, wenn man sich bewusst gegen die eigenen Änderungen entscheidet oder einen externen Stand übernehmen will.
- Tab-Completion funktioniert wie in der Shell: Nach `:e` kann mit der **Tab** der Pfad vervollständigt werden.

Übungen

1. Öffne eine Datei `test.txt` mit:
   - `:edit test.txt`

   Ändere eine Zeile und speichere *nicht*. Öffne anschließend dieselbe Datei mit:
   - `:edit test.txt`

   Beobachte, dass VIM auf ungespeicherte Änderungen hinweist.
2. Wiederhole das Szenario und verwende diesmal:
   - `:edit!`

Kontrolliere, dass deine ungespeicherten Änderungen damit verworfen und der Dateistand von der Festplatte geladen wird.

3. Öffne nacheinander zwei verschiedene Dateien, z. B. `a.txt` und `b.txt`, jeweils mit:
   - `:edit a.txt`
   - `:edit b.txt`

   Nutze anschließend:
   - **Ctrl** - **6**

   um zwischen den beiden zuletzt verwendeten Dateien hin- und herzuschalten.
   Ziel: Das Zusammenspiel von Buffern und Dateinamen *beim Öffnen* verstehen.

12.2.2. Relative und absolute Pfade

VIM versteht alle gängigen Pfadvarianten. Entscheidend ist dabei immer das aktuelle Arbeitsverzeichnis (`:pwd`):

- `:e ./config.ini` öffnet `config.ini` im aktuellen Verzeichnis (das `./` kann weggelassen werden, ist aber oft zur Klarheit nützlich).
- `:e ../logs/app.log` nutzt einen relativen Pfad zum Elternverzeichnis.
- `:e ~/notes/todo.txt` öffnet eine Datei im Home-Verzeichnis des aktuellen Users (Platzhalter ~).
- `:e /usr/local/etc/hosts` absolutes Dateisystem-Path, unabhängig vom aktuellen Arbeitsverzeichnis.

Mit Tab-Completion lassen sich Pfade bequem vervollständigen: Nach der Eingabe von `:e` einfach die **Tab** drücken, um Verzeichnisse und Dateinamen schrittweise zu ergänzen. Mehrfaches Drücken zeigt jeweils weitere Kandidaten.

Praktische Ergänzungen:

- `:pwd` zeigt das aktuelle Arbeitsverzeichnis an, auf das sich relative Pfade beziehen.
- `:cd <pfad>` setzt das Arbeitsverzeichnis global für die aktuelle VIM-Instanz.
- `:lcd <pfad>` setzt das Arbeitsverzeichnis *lokal* für das aktuelle Fenster (Split-spezifisch).

Übungen

1. Wechsle in ein Projektverzeichnis in der Shell und starte dort VIM. Öffne nacheinander:
   - `:e ./README.md`
   - `:e src/main.c`

   Vergleiche mit der Ausgabe von `:pwd` und überprüfe, wie sich relative Pfade verhalten.
2. Öffne eine Datei im Home-Verzeichnis mit:
   - `:e ~/test.txt`

   Ändere anschließend mit:
   - `:cd ~`

   das Arbeitsverzeichnis und wiederhole `:e test.txt`. Ziel: Verstehen, wann ~ nötig ist und wann der aktuelle Pfad ausreicht.
3. Nutze Tab-Completion: Tippe im Ex-Modus
   - `:e /usr/lo`

   und drücke mehrfach die **Tab**. Beobachte, wie VIM Schritt für Schritt auf `/usr/local/` und darunterliegende Verzeichnisse vervollständigt.

12.2.3. Datei in neuem Fenster öffnen

Statt eine Datei im aktuellen Fenster zu laden, kann sie direkt in einem neuen Split geöffnet werden. So bleibt der ursprüngliche Kontext sichtbar und man kann Dateien nebeneinander betrachten.

- `:split <Datei>` Öffnet *<Datei>* in einem neuen horizontalen Fenster unterhalb des aktuellen. Der aktuelle Buffer bleibt oben sichtbar.
- `:vsplit <Datei>` Öffnet *<Datei>* in einem neuen vertikalen Fenster rechts. Ideal für Vergleiche (links Code, rechts Tests).
- `:new <Datei>` Erzeugt zunächst ein leeres, horizontal gesplittetes Fenster und lädt dann *<Datei>* hinein. Ohne Dateinamen entsteht ein neuer, unbenannter Buffer.
- `:vnew <Datei>` Entspricht `:new`, aber als vertikaler Split. Praktisch, wenn links der ursprüngliche Kontext erhalten bleiben soll.

Wichtig ist der Unterschied zwischen „Fenster“ und „Buffer“: *Das Fenster* ist nur die Ansicht, *der Buffer* ist die eigentliche Datei im Speicher. Mit `:split` und `:vsplit` erzeugst du neue Ansichten, die jeweils einen Buffer anzeigen.

Typische Einsatzszenarien:

- links: Spezifikation oder Dokumentation, rechts: Implementierung im Quellcode;
- oben: Logdatei, unten: Konfigurationsdatei oder laufendes Terminal;
- links: Originaldatei, rechts: eine leicht modifizierte Variante.

Übungen

1. Öffne eine Datei (z. B. `main.c`) mit:
   - `:e main.c`

   Erzeuge danach:
   - `:vsplit main.h`

   Wechsle mit **Ctrl**-**w** **h** und **Ctrl**-**w** **l** zwischen den Fenstern hin und her und beobachte, dass beide denselben Projektkontext abdecken, aber unterschiedliche Dateien zeigen.
2. Starte VIM in einem beliebigen Verzeichnis und führe nacheinander aus:
   - `:new notizen.txt`
   - `:vnew scratch.txt`

   Schreibe in jedes Fenster etwas Text und prüfe mit `:ls`, welche Buffer dadurch angelegt wurden. Speichere beide Dateien anschließend mit `:w` und verlasse VIM.
3. Öffne eine Logdatei im Vollbild (`:e app.log`) und erzeuge dann:
   - `:split`

   Scrolle oben in ältere Logzeilen, unten in jüngere. Ziel: Verstehen, dass beide Fenster denselben Buffer zeigen, aber unterschiedliche Scrollpositionen und Cursorstände besitzen.

12.2.4. Von Datei zu Datei springen

Da VIM dateibasiert über Buffer arbeitet, ist der Wechsel zwischen bereits geöffneten Dateien extrem schnell. Typischer Ablauf: `:ls` anzeigen, Nummer merken, mit `:b <Nummer>` springen – oder mit **Ctrl**-**6** einfach zwischen den letzten zwei Dateien hin- und herschalten.

- `:b <Nummer>` Wechselt zum Buffer mit der angegebenen Nummer. Die Nummer stammt aus der Buffer-Liste (`:ls`).
- `:ls` Zeigt alle aktuell bekannten Buffer an, inklusive Status: „%“ für den aktuellen, „#“ für den alternativen Buffer, „+“ für geänderte Dateien usw.

- **Ctrl** - **6** Wechselt zwischen aktuellem und alternativem Buffer. Praktisch wie ein kleiner „Alt-Tab“ zwischen zwei Dateien.

Typische Szenarien:

- Wechsel zwischen Implementierung und Header (z. B. `main.c` ↔ `main.h`)
- Wechsel zwischen Testdatei und Produktivcode
- Wechsel zwischen Konfigurationsdatei und Logfile

Übungen

1. Öffne drei Dateien nacheinander:
   - `:e a.txt`
   - `:e b.txt`
   - `:e c.txt`

   Lass dir mit `:ls` die Buffer-Liste anzeigen und wechsle dann gezielt:
   - `:b 1`
   - `:b 2`
   - `:b 3`

   Beobachte, wie sich die Markierungen „%“ und „#“ in der Liste verändern.
2. Öffne zwei Dateien, z. B.:
   - `:e config.yaml`
   - `:e app.log`

   Wechsle nun mehrmals mit **Ctrl** - **6** hin und her. Ziel: Die Tastenkombination soll sich wie ein Reflex anfühlen, wenn du spontan zwischen zwei Dateien springen musst.
3. Erzeuge zusätzlich einen neuen, unbenannten Buffer mit:
   - `:enew`

   Schreibe kurz Text hinein und prüfe erneut mit `:ls`, welche Nummer dieser neue Buffer erhalten hat. Wechsle dann gezielt zwischen den drei wichtigsten Arbeitsdateien mit `:b <Nummer>` und **Ctrl** - **6**.

12.2.5. Dateien aus dem Dateibrowser öffnen

Neben `:edit` bietet VIM mit dem integrierten Dateibrowser (netrw) eine komfortable Möglichkeit, Dateien ohne Verlassen des Editors zu öffnen.

Die wichtigsten Kommandos:

- `:Ex` Öffnet den Dateibrowser im *aktuellen* Fenster (Kurzform von `:Explore`).

- `:Sex` Öffnet den Dateibrowser in einem *horizontalen Split* (Kurzform von `:Sexplore`).
- `:Vex` Öffnet den Dateibrowser in einem *vertikalen Split* (Kurzform von `:Vexplore`).

Im Dateibrowser selbst gelten u. a. folgende Tasten:

- **Enter** Verzeichnis öffnen oder Datei laden.
- **-** Ein Verzeichnis nach oben wechseln.
- **u** Zur vorherigen Verzeichnisebene zurückspringen (Verlauf).
- **R** Ansicht neu einlesen (Refresh).
- **i** Anzeigeformat umschalten (kompakt, detailliert, Baumansicht).

Auf diese Weise lässt sich sehr schnell durch Projektverzeichnisse navigieren, ohne `:edit`-Kommandos mit kompletten Pfaden eintippen zu müssen.

Übungen

1. Wechsle in ein Verzeichnis mit mehreren Unterordnern und Dateien. Öffne den Dateibrowser mit:
   - `:Ex`

   Navigiere mit den Cursortasten durch die Liste und öffne einige Dateien mit **Enter**. Kehre anschließend mit **-** jeweils eine Ebene nach oben und beobachte, wie sich der angezeigte Pfad ändert.
2. Öffne denselben Ordner in einem Split:
   - `:Sex`

   Lass im oberen Fenster den Dateibrowser geöffnet und lade Dateien im unteren Fenster. Ziel: Dateibrowser als „Projektübersicht“ nutzen, während im unteren Fenster gearbeitet wird.
3. Probiere nun:
   - `:Vex`

   und vergleiche das Arbeiten mit einem vertikalen Dateibrowser links/rechts gegenüber der horizontalen Variante. Finde heraus, welche Anordnung für deinen Bildschirm und dein Projektlayout angenehmer ist.

12.2.6. Nicht vorhandene Dateien öffnen

VIM erlaubt das Öffnen neuer Dateien sofort, auch wenn sie auf dem Dateisystem noch nicht existieren:

- `:e neuescript.py` öffnet einen neuen Buffer mit dem Namen `neuescript.py`.

Wichtig: **Der Buffer existiert sofort, die Datei auf der Platte aber noch nicht.** Erst beim Speichern mit `:w` oder `:w neuescript.py` wird die Datei tatsächlich angelegt.

Typische Varianten:

- `:e src/main.c` legt die Struktur `src/` *nicht* automatisch an. Das Verzeichnis muss vorher existieren.
- `:enew` erzeugt einen vollständig neuen, unbenannten Buffer (ohne Pfad). Erst mit `:file name.txt` oder `:w name.txt` bekommt der Buffer einen Dateinamen.
- `:file notes.md` gibt einem existierenden, unbenannten Buffer nachträglich einen Namen, ohne sofort zu schreiben. Das tatsächliche Anlegen erfolgt dann mit `:w`.

Übungen

1. Öffne einen neuen Buffer mit:
   - `:e sandbox.txt`

   Prüfe mit `:ls`, dass der Buffer existiert, und kontrolliere im Dateisystem, dass `sandbox.txt` noch nicht angelegt wurde. Speichere anschließend mit `:w` und überprüfe, dass die Datei nun physisch vorhanden ist.
2. Erzeuge einen komplett unbenannten Buffer mit:
   - `:enew`

   Füge Text ein und setze danach mit
   - `:file scratch.log`

   nur den Dateinamen. Speichere *erst danach* mit `:w` und beobachte den Unterschied zu `:e name`.
3. Versuche, eine Datei in einem nicht existierenden Unterordner zu öffnen:
   - `:e tmp/demo/newfile.txt`

   Notiere dir die Fehlermeldung (z. B. „E212“) und lege das Verzeichnis anschließend mit einem externen Kommando an (z. B. `:!mkdir -p tmp/demo`). Wiederhole dann `:e tmp/demo/newfile.txt` und speichere mit `:w`. Ziel: Verstehen, dass VIM keine Verzeichnisse anlegt, aber sehr wohl neue Dateien in vorhandenen Pfaden.

12.2.7. Dateien sicher neu laden

Wenn eine Datei sowohl in VIM als auch extern (z. B. in einem anderen Editor oder durch ein Build-Tool) verändert wird, erkennt VIM diesen Zustand und warnt:

```
W11: Warning: File changed since editing
```

In diesem Moment muss bewusst entschieden werden, ob der *VIM-Stand* oder der *Dateisystem-Stand* maßgeblich sein soll.

Typische Kommandos:

- `:e` lädt die aktuelle Datei explizit neu von der Platte. Ungespeicherte Änderungen im Buffer gehen dabei verloren.
- `:edit!` ist die „harte“ Variante und verwirft ohne Rückfrage alle lokalen Änderungen am Buffer, um den aktuellen Disk-Stand zu laden.
- `:checktime` prüft alle geöffneten Dateien, ob sie extern verändert wurden, und zeigt entsprechende Warnungen an. Besonders nützlich, wenn externe Tools (Git, Formatter, Generatoren) Dateien neu schreiben.

In Kombination mit der Option `:set autoread` kann VIM geänderte Dateien automatisch neu einlesen, sobald der Editor wieder in den Vordergrund kommt oder `:checktime` ausgeführt wird. Trotzdem bleibt die Entscheidung immer bewusst: Man kann externe Änderungen ignorieren (eigenen Stand behalten) oder gezielt neu laden.

Übungen

1. Öffne eine Datei in VIM, merke dir eine markante Zeile und ändere sie außerhalb von VIM (z. B. in einem anderen Editor oder per `sed`). Kehre zu VIM zurück und führe:
   - `:checktime`

   aus. Beobachte die Warnung und lade die Datei mit `:e` neu.
2. Wiederhole das Experiment, ändere die Datei diesmal *in VIM* (aber speichere nicht) und *außerhalb* gleichzeitig. Führe dann `:checktime` aus und entscheide bewusst, ob du mit `:edit!` den externen Stand akzeptierst oder deine VIM-Änderungen vorher mit `:w` sicherst.
3. Aktiviere:
   - `:set autoread`

   Wechsle zwischen VIM und einem externen Tool, das die Datei überschreibt (z. B. ein Code-Formatter oder `git checkout`). Nutze `:checktime`,

um zu überprüfen, wann VIM die Änderungen bemerkt und wie sich der Workflow mit „automatischem Neu-Laden“ anfühlt.

12.2.8. Ganze Verzeichnisse öffnen

Statt einzelne Dateien manuell zu öffnen, kann VIM über die *Argumentliste* (`:args`) ganze Verzeichnisse oder Dateimengen auf einmal laden. Typisch ist die Arbeit mit Wildcards:

- `:args *.txt` lädt alle `.txt`-Dateien des aktuellen Verzeichnisses in die Argumentliste. Jede dieser Dateien wird dadurch „Teil der Sitzung“, auch wenn sie noch nicht in einem Fenster angezeigt wird.
- `:args` zeigt die aktuelle Argumentliste an. Die gerade aktive Datei ist mit einem `[current]`-Marker oder einem ähnlichen Hinweis gekennzeichnet.
- `:next` / `:prev` springt zur nächsten bzw. vorherigen Datei in der Argumentliste.

Weitere nützliche Kommandos im Kontext:

- `:first` / `:last` springt zur ersten bzw. letzten Datei der Argumentliste.
- `:argadd foo.c` fügt `foo.c` zusätzlich zur bestehenden Argumentliste hinzu, ohne sie komplett zu überschreiben.
- `:argdelete` entfernt die aktuelle Datei aus der Argumentliste (der Buffer kann dennoch weiter existieren).

Besonders mächtig wird `:args` in Kombination mit `:argdo`:

```
:argdo s/foo/bar/ge \textbar {} update
```

führt die Ersetzung `foo` → `bar` *in allen Dateien der Argumentliste* aus und speichert nur dort, wo Änderungen erfolgt sind.

Übungen

1. Wechsle in ein Verzeichnis mit mehreren `.txt`-Dateien und führe nacheinander aus:
   - `:args *.txt`
   - `:args`

   Navigiere dann mit `:next` und `:prev` durch die Dateien und beobachte jeweils den Dateinamen in der Statuszeile.
2. Ergänze die bestehende Argumentliste um eine zusätzliche Datei:
   - `:argadd notes.md`

Kontrolliere mit `:args`, ob `notes.md` korrekt hinzugefügt wurde und springe mit `:last` dorthin.

3. Führe eine einfache globale Änderung über alle Argumentdateien aus:
   - `:args *.log`
   - `:argdo s/ERROR/WARN/ge \textbar {} update`

   Öffne anschließend einige der Dateien und überprüfe, ob die Ersetzung überall erfolgt ist. Ziel: Ein Gefühl dafür bekommen, wie VIM „ein ganzes Verzeichnis“ als Arbeitsmenge behandelt.

12.3. Beenden

VIM stellt mehrere Varianten bereit, um Dateien oder die gesamte Sitzung zu schließen. Der Editor verlangt dabei grundsätzlich eine bewusste Entscheidung über ungespeicherte Änderungen.

12.3.1. Standardbefehle

Die klassischen Ex-Befehle zum Beenden orientieren sich daran, ob noch Änderungen im Buffer vorhanden sind und ob nur ein Fenster oder die gesamte Sitzung betroffen ist:

- `:q` schließt *das aktuelle Fenster*. Wenn der zugehörige Buffer ungespeicherte Änderungen enthält, verweigert VIM mit einer Warnung („No write since last change“).
- `:wq` speichert die aktuelle Datei und schließt anschließend das Fenster. Entspricht funktional `:w` gefolgt von `:q`.
- `:x` verhält sich ähnlich wie `:wq`, speichert aber nur, *wenn* tatsächlich Änderungen vorhanden sind. Bei unverändertem Buffer wird nur beendet, ohne zu schreiben.
- `:wqall` speichert *alle* geänderten Buffer und beendet danach VIM vollständig. Praktisch, wenn mehrere Dateien gleichzeitig geöffnet und bearbeitet wurden.
- `:qall` versucht, alle Fenster/Buffer zu schließen. Existieren noch ungespeicherte Änderungen, bricht VIM mit einer Warnung ab und beendet *nicht*, solange nicht bewusst gespeichert oder erzwungen wird.

Merke: `:q` und `:qall` sind *defensiv* und brechen bei ungespeicherten Buffers ab, während `:wq` und `:wqall` explizit speichern und danach beenden.

Übungen

1. Öffne eine Datei, nimm eine kleine Änderung vor und teste nacheinander:
   - `:q`
   - `:wq`

   Beobachte, wie VIM sich im ersten Fall weigert zu schließen und im zweiten Fall erst speichert und dann beendet.
2. Öffne drei Dateien (z. B. mit `:args *.txt`) und ändere in jeder eine Zeile. Führe anschließend aus:
   - `:qall`
   - `:wqall`

   Notiere dir, in welchem Fall VIM abbricht und in welchem Fall alle Änderungen in einem Rutsch gespeichert werden.
3. Öffne eine Datei, nimm *keine* Änderungen vor und führe nacheinander aus:
   - `:x`
   - `:wq`

   Vergleiche beide Fälle: `:x` schreibt die Datei nicht unnötig neu, `:wq` schreibt immer. Ziel: Ein Gefühl dafür bekommen, wann `:x` sinnvoll ist (z. B. bei großen Dateien auf langsamen Medien).

12.3.2. Beenden erzwingen

Manchmal soll VIM *trotz* ungespeicherter Änderungen beendet werden, etwa bei Testdateien, temporären Buffern oder bewusst verworfenen Änderungen. Dafür gibt es die „Force"-Varianten der Quit-Befehle:

- `:q!` schließt das aktuelle Fenster und verwirft alle ungespeicherten Änderungen des zugehörigen Buffers. Entspricht: „Beenden, egal was geändert wurde".
- `:qall!` beendet VIM vollständig und verwirft alle Änderungen in *allen* offenen Buffern. Vorsicht: Es erfolgt *keine* Rückfrage pro Datei – alles Nichtgespeicherte geht verloren.

Wichtig: Diese Befehle umgehen bewusst den üblichen Schutzmechanismus gegen Datenverlust. Sie sind hilfreich in Wegwerf-Sessions (z. B. Log-Analyse, Temp-Dateien), sollten aber nicht zur Gewohnheit werden.

Übungen

1. Öffne eine Testdatei, füge eine Zeile hinzu und versuche zunächst:

- `:q`

Notiere die Warnung, die VIM ausgibt. Führe danach:

- `:q!`

aus und beobachte, dass die Änderungen ohne Nachfrage verworfen werden.

2. Öffne drei Dateien parallel (z. B. mit `:args *.txt`) und ändere in jeder Datei eine Kleinigkeit. Führe anschließend:
   - `:qall!`

   aus. Überprüfe nach dem erneuten Öffnen der Dateien, dass *keine* der Änderungen übernommen wurde.
3. Überlege dir eigene Kriterien, wann du `:q!` und `:qall!` bewusst einsetzen willst (z. B. nur in temporären Arbeitsverzeichnissen, nie im Projekt-Repository). Ziel: Ein klarer mentaler Unterschied zwischen „sicher beenden“ (`:wq`, `:wqall`) und „hart abbrechen“ (`:q!`, `:qall!`).

12.3.3. Beenden über ZZ und ZQ

VIM bietet zwei klassische, besonders schnelle Befehle im Normalmodus, die sich wie Ex-Kommandos verhalten, aber *ohne Doppelpunkt* und ohne Eingabezeile auskommen:

- **ZZ** — speichern (falls nötig) und aktuelles Fenster schließen (entspricht in der Praxis grob `:x`).
- **ZQ** — schließen, *ohne* zu speichern (entspricht grob `:q!`).

Wichtig:

- Beide Befehle werden im Normalmodus durch Drücken der Tasten **Z** **Z** bzw. **Z** **Q** ausgelöst (also *ohne* Doppelpunkt und ohne **Enter**).
- **ZZ** speichert nur dann, wenn der Buffer tatsächlich geändert wurde.
- **ZQ** verwirft alle ungespeicherten Änderungen des aktuellen Buffers, ohne Rückfrage.
- Bei mehreren Fenstern kann **ZZ** bzw. **ZQ** dazu führen, dass das aktuelle Fenster geschlossen wird, während VIM selbst noch weiterläuft (abhängig von Setup und GUI).

Typischer Einsatzzweck:

- **ZZ**: schnelles „Speichern und raus“ aus einer Datei im Normalmodus.
- **ZQ**: schnelles „Wegwerfen und raus“, z. B. bei temporären Test-Dateien.

Übungen

1. Öffne eine Datei, nimm eine kleine Änderung vor und drücke im Normalmodus:
   - **Z** **Z**

   Öffne die Datei danach erneut und überprüfe, dass die Änderung gespeichert wurde.
2. Öffne die gleiche Datei, nimm wieder eine Änderung vor und verlasse sie diesmal mit:
   - **Z** **Q**

   Starte VIM erneut und kontrolliere, dass diese Änderung *nicht* übernommen wurde.
3. Vergleiche dein eigenes Arbeitsmuster: In welchen Situationen verwendest du lieber „klassische“ Ex-Kommandos wie `:wq` / `:q!`, und wann sind **ZZ** / **ZQ** im Normalmodus schneller und praktischer? Ziel: Ein Gefühl dafür entwickeln, wann welche Variante deine typische Arbeitsweise am besten unterstützt.

12.3.4. Beenden trotz Swap-Dateien

Swap-Dateien (`.swp`) dienen als Sicherheitsnetz, falls VIM oder das System unerwartet abstürzen. Beim erneuten Öffnen derselben Datei erkennt VIM die vorhandene Swap-Datei und warnt typischerweise mit einer Meldung wie:

```
E325: ATTENTION
Found a swap file by the name ".foo.txt.swp"
...
```

In dieser Situation muss bewusst entschieden werden, wie fortgefahren werden soll:

- `:recover` — versucht, den Inhalt aus der Swap-Datei wiederherzustellen und in den aktuellen Buffer zu laden. Dies ist die *sichere* Wahl, wenn unklar ist, ob noch ungespeicherte Änderungen enthalten sind.
- `:q!` — VIM beenden bzw. das Öffnen abbrechen und die Wiederherstellung ignorieren. Die Swap-Datei bleibt dabei im Dateisystem erhalten, bis sie manuell gelöscht oder durch einen erfolgreichen Schreibvorgang ersetzt wird.

Typischer Ablauf nach einem Crash:

1. Datei wie gewohnt mit `:edit` öffnen. VIM meldet das Vorhandensein einer `.swp`-Datei.
2. `:recover` ausführen, um den letzten Stand aus der Swap-Datei zu laden.
3. Datei normal prüfen, dann mit `:write` speichern. Danach kann die alte `.swp`-Datei in der Regel gefahrlos gelöscht werden.

Übungen

1. Simuliere einen Crash in einer Testdatei: Öffne eine Datei, nimm Änderungen vor und beende VIM *nicht* sauber (z. B. mit einem erzwungenen Editor-Abbruch in einer VM). Öffne danach dieselbe Datei neu und beobachte die Swap-Warnmeldung. Führe `:recover` aus und speichere die Datei anschließend regulär.
2. Öffne eine beliebige Datei und prüfe mit:
   - `:swapname`

   wo VIM die zugehörige Swap-Datei ablegt. Notiere dir den Pfad und überprüfe im Dateisystem, dass die Datei dort tatsächlich existiert.
3. Betrachte die Hilfe zu Swap-Warnungen mit:
   - `:help E325`

   und verschaffe dir einen Überblick über die angebotenen Optionen („(R)ecover“, „(E)dit anyway“, „(D)elete it“ usw.). Ziel: Verstehen, wie du im Ernstfall bewusst entscheidest, ob du Daten retten oder eine alte Swap-Datei ignorieren möchtest.

12.3.5. Buffer schließen ohne VIM zu verlassen

VIM selbst bleibt aktiv, während nur der *aktuelle* Buffer aus der Bufferliste entfernt wird. Das ist sinnvoll, wenn du eine Datei nicht mehr brauchst, aber die Sitzung, Fensteraufteilung oder andere Dateien weiter nutzen möchtest.

- `:bd` schließt den aktuellen Buffer. Voraussetzung: Änderungen müssen vorher gespeichert sein (`:write`), sonst verweigert VIM den Befehl.
- `:bd!` schließt den aktuellen Buffer und verwirft alle ungespeicherten Änderungen. Dies entspricht einem „hartes Schließen“ dieses Buffers.

Wichtig ist die Trennung:

- `:q` / `:quit` schließt ein *Fenster* (das Layout-Element).
- `:bd` / `:bdelete` entfernt einen *Buffer* (die geladene Datei).

Ein geschlossenes Fenster lässt den Buffer normalerweise weiter existieren (z. B. sichtbar in `:ls`). Ein gelöschter Buffer ist hingegen nicht mehr unter seiner Buffer-Nummer verfügbar, bis die Datei erneut mit `:edit` geladen wird.

Typische Anwendungsfälle

- temporäre Dateien oder Scratch-Buffer schließen, ohne VIM zu beenden,
- aufgeräumte Bufferliste herstellen, wenn viele Dateien nur kurz geöffnet wurden,
- ein Fensterlayout beibehalten, aber die Zuordnung zu einer Datei ändern (z. B. `:bd` und danach `:edit andere\_datei`).

Übungen

1. Öffne nacheinander drei Dateien (z. B. `a.txt`, `b.txt`, `c.txt`) und prüfe mit:
   - `:ls`

   die Bufferliste. Schließe dann den aktuellen Buffer mit `:bd` und kontrolliere erneut mit `:ls`, welche Buffer noch vorhanden sind.
2. Nimm in einer Testdatei eine Änderung vor (ohne zu speichern) und versuche `:bd` auszuführen. Beobachte die Fehlermeldung und wiederhole den Vorgang mit:
   - `:bd!`

   um die Änderung gezielt zu verwerfen.
3. Öffne eine Datei in zwei Fenstern:
   - `:split`

   Schließe im oberen Fenster den Buffer mit `:bd`. Beobachte, dass das Fenster zwar erhalten bleibt, aber nun keinen Inhalt mehr hat bzw. auf einen anderen Buffer wechselt. Ziel: Den Unterschied zwischen „Fenster schließen“ (`:q`) und „Buffer entfernen“ (`:bd`) bewusst nachvollziehen.

12.3.6. Session speichern und später beenden

Sessions erlauben es, einen kompletten Arbeitszustand von VIM zu sichern: geöffnete Dateien (Buffer), Fensteraufteilung (Splits), Tabs, aktuelles Verzeichnis und diverse Optionen. Dieser Zustand kann später in einer neuen VIM-Instanz wiederhergestellt werden.

Die wichtigsten Befehle:

- `:mksession` speichert die aktuelle Sitzung in der Datei `Session.vim` im aktuellen Arbeitsverzeichnis. Existiert diese Datei bereits, wird sie ohne Rückfrage überschrieben.
- `:mksession projekt.vim` speichert die Sitzung explizit in `projekt.vim`. So können mehrere Projekte eigene Session-Dateien besitzen.
- `:source Session.vim` liest die Session-Datei ein und stellt Fenster, Buffer und Layout so weit wie möglich wieder her. Alternativ kann VIM direkt mit:

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

  gestartet werden.

Sessions sind ideal, wenn du:

- an einem Projekt über mehrere Tage arbeitest,
- komplexe Fenster- und Tab-Layouts aufgebaut hast,
- viele Dateien gleichzeitig geöffnet hast und diesen Zustand reproduzierbar wieder laden willst.

Feineres Tuning über `sessionoptions` Was genau in einer Session gespeichert wird, steuert die Option `sessionoptions`:

- `:set sessionoptions?` zeigt die aktuelle Einstellung.
- Typische Einträge sind z. B. `buffers`, `curdir`, `folds`, `help`, `options`, `tabpages`.

Für ein projektorientiertes Arbeiten ist es oft sinnvoll, sicherzustellen, dass mindestens `buffers`, `curdir` und `tabpages` enthalten sind, damit alle Dateien, das Arbeitsverzeichnis und die Tab-Layouts wiederkommen.

Übungen

1. Öffne mehrere Dateien, richte ein paar Splits ein (z. B. Code links, Tests rechts, Terminal unten) und speichere dann eine Session:
   - `:mksession`

   Beende VIM anschließend mit `:qall`. Starte danach eine neue VIM-Instanz im gleichen Verzeichnis und führe:
   - `:source Session.vim`

aus. Ziel: Überprüfen, welche Teile des Layouts exakt wiederhergestellt werden.

2. Erzeuge eine projektspezifische Session:
   - `:mksession meinprojekt.vim`

   Beende VIM und starte es später mit:

   ```
   vim -S meinprojekt.vim
   ```

   Beobachte den Unterschied zu einer leeren VIM-Instanz.
3. Lass dir den aktuellen Wert von `sessionoptions` anzeigen:
   - `:set sessionoptions?`

   Notiere dir die aufgelisteten Flags und überlege, ob du z. B. Faltungen (`folds`) oder globale Optionen (`options`) dauerhaft mit sichern möchtest. Ändere testweise:
   - `:set sessionoptions+=folds`

   und speichere eine neue Session. Prüfe beim Laden, ob deine Faltungen wiederhergestellt werden.

12.3.7. Terminal-Fenster beenden

Bei Terminal-Splits (`:terminal`) gibt es drei getrennte Ebenen:

1. den *Terminal-Modus* in VIM,
2. den eigentlichen Shell-Prozess (z. B. `bash`, `zsh`),
3. das Fenster bzw. den Buffer in VIM.

Je nachdem, was genau beendet werden soll, kommen unterschiedliche Schritte zum Einsatz.

- **Ctrl**-**** **Ctrl**-**n** — Terminal-Modus verlassen Wechselt nur vom Terminal-Insert-Modus zurück in den Normalmodus von VIM. Der Shell-Prozess läuft weiter, das Fenster bleibt offen.
- `exit` im Terminal Beendet den Shell-Prozess (z. B. `bash`) innerhalb des Terminal-Buffers. Danach zeigt VIM nur noch die Beendigungsmeldung; nun kann das Fenster problemlos mit `:q` oder `:bd` geschlossen werden.
- `:q` im Normalmodus Schließt das *Fenster*, in dem der Terminal-Buffer angezeigt wird. Wenn noch ein laufender Job existiert, warnt VIM ggf. und verweigert das Schließen, bis der Prozess beendet ist (oder erzwungen wird).

- `:bd` / `:bd!` `:bd` schließt den Terminal-Buffer (Fenster bleibt ggf. leer oder zeigt einen anderen Buffer). Mit `:bd!` kann das Schließen erzwungen werden, auch wenn der Terminal-Job noch läuft (der Prozess wird dabei beendet).

Praktische Faustregel:

1. **Ctrl**-\ **Ctrl**-**n** — aus dem Terminal-Modus *raus* in Normalmodus.
2. `exit` — Shell-Prozess wirklich beenden.
3. `:q` oder `:bd` — Fenster/Buffer in VIM aufräumen.

Übungen

1. Öffne ein Terminal-Fenster mit:
   - `:terminal`

   Starte darin einen langen Befehl, z. B. `ping 127.0.0.1`. Wechsle mit **Ctrl**-\ **Ctrl**-**n** in den Normalmodus und navigiere mit **k**/**j** im Scrollback. Kehre dann mit **i** in den Terminal-Modus zurück und beende `ping` (z. B. `<C-c>`).
2. Wieder im Terminal-Insert-Modus: gib `exit` ein und drücke **Enter**. Beobachte, dass der Shell-Prozess beendet ist. Wechsle mit **Ctrl**-\ **Ctrl**-**n** in den Normalmodus und schließe das Fenster mit:
   - `:q`
3. Öffne erneut ein Terminal mit `:terminal`, wechsle mit **Ctrl**-\ **Ctrl**-**n** in den Normalmodus und schließe den Buffer diesmal direkt mit:
   - `:bd`

   Beobachte den Unterschied zu `:q`: `:q` schließt nur das Fenster, `:bd` entfernt den Buffer aus VIM.

12.4. Buffer-Management

VIM arbeitet nicht dateibasiert, sondern *bufferbasiert*. Jede geöffnete Datei wird als Buffer geführt, unabhängig davon, ob sie in einem Fenster angezeigt wird. Dadurch lassen sich viele Dateien parallel bearbeiten, ohne dass Tabs oder Fenster überladen werden. Ein Buffer ist damit die eigentliche Arbeitseinheit – Fenster und Tabs sind nur unterschiedliche Sichten darauf.

12.4.1. Buffer anzeigen

Die aktuelle Buffer-Situation lässt sich über mehrere Ex-Befehle prüfen:

- `:ls` — Liste aller Buffer mit kompaktem Status
- `:buffers` — ausführlichere Buffer-Liste (Alias von `:ls` mit leicht anderer Darstellung)
- `:files` — alternative Anzeige (historische Variante, teils andere Spalten)

Typische Ausgabe von `:ls` (schematisch):

```
1 %a   "main.c"        line 120
2  a   "util.c"        line  45
3  h   "README.md"     line  10
```

Die wichtigsten Statuskennzeichen:

- **%** — aktueller Buffer (der gerade im Fokus ist)
- **#** — alternativer Buffer (der zuvor aktive Buffer, Umschalten mit **Ctrl** - **6**)
- **+** — Änderungen vorhanden, noch nicht gespeichert
- **a** — aktuell in mindestens einem Fenster angezeigt („active“)
- **h** — versteckt („hidden“), nicht in einem Fenster sichtbar, aber weiterhin im Speicher

Ob „hidden“ erlaubt ist, hängt von der Option `:set hidden` ab:

- ohne `:set hidden` zwingt VIM oft zum Speichern, bevor der Buffer verlassen wird,
- mit `:set hidden` dürfen geänderte Buffer unsichtbar im Hintergrund weiterexistieren.

Übungen

1. Öffne drei Dateien (z. B. `a.txt`, `b.txt`, `c.txt`) und wechsle zwischen ihnen mit:
   - `:e a.txt`, `:e b.txt`, `:e c.txt`

   Führe dann `:ls` aus und notiere dir, welcher Buffer mit % und welcher mit # markiert ist.
2. Aktiviere versteckte Buffer mit:
   - `:set hidden`

   Ändere eine Zeile in `a.txt`, wechsle mit `:e b.txt` zu einer anderen Datei, ohne zu speichern, und rufe `:ls` auf. Beobachte, dass `a.txt` als geänderter, aber „hidden“er Buffer (+h) in der Liste bleibt.

3. Öffne mehrere Dateien und nutze:
   - `:buffers`

   Wechsle anschließend gezielt per Nummer mit:
   - `:buffer 1`, `:buffer 2`, `:buffer 3`

   Ziel: Ein Gefühl dafür bekommen, dass `:ls` / `:buffers` die *Buffer-Welt* zeigen – unabhängig davon, wie viele Fenster oder Tabs geöffnet sind.

12.4.2. Zwischen Buffern wechseln

Hat man mehrere Dateien als Buffer geöffnet, wird das schnelle Umschalten zwischen ihnen zur Kernfunktion. VIM bietet dafür sowohl nummernbasierte als auch relative Befehle.

- `:buffer <Nummer>` wechselt direkt zu dem Buffer mit der angegebenen Nummer (wie sie in `:ls` / `:buffers` angezeigt wird).
- `:b <Nummer>` Kurzform von `:buffer <Nummer>`.
- `:bnext` zum *nächsten* Buffer in der internen Liste wechseln (Kurzform: `:bn`).
- `:bprevious` zum *vorherigen* Buffer wechseln (Kurzform: `:bp`).
- `:bfirst` / `:blast` zum ersten bzw. letzten Buffer der Liste springen.
- **Ctrl**-**6** zwischen aktuellem und alternativem Buffer (#-Markierung in `:ls`) hin- und herschalten. Praktisch wie „ALT-TAB“ zwischen zwei zuletzt verwendeten Dateien.

Wichtig: **Ctrl**-**6** ist kein Ex-Befehl, sondern eine reine Tastenkombination im Normal-Mode. Sie nutzt die interne „alternate file“-Mechanik (#), die auch in vielen anderen VIM-Kommandos vorkommt (z. B. `:e \#`).

Übungen

1. Öffne nacheinander drei Dateien, etwa:
   - `:e a.txt`
   - `:e b.txt`
   - `:e c.txt`

   Führe dann `:ls` aus und notiere dir die Buffer-Nummern. Wechsle gezielt mit:
   - `:buffer 1`, `:buffer 2`, `:buffer 3`

   und beobachte, wie sich % (aktuell) und # (alternativ) in der Liste verändern.

2. Öffne zwei Dateien, z. B. `main.c` und `util.c`. Springe einige Male zwischen ihnen hin und her:
   - **Ctrl** - **6**

   Ziel: **Ctrl** - **6** soll sich wie ein reflexartiger „ALT-TAB“ zwischen zwei Dateien anfühlen.
3. Öffne vier bis fünf Dateien und experimentiere mit:
   - `:bnext`, `:bprevious`, `:bfirst`, `:blast`

   Überlege dir anschließend, in welchen Situationen du lieber mit *Nummern* (`:buffer 3`) und wann du lieber mit *relativen* Befehlen (`:bnext`/`:bprevious`) arbeiten würdest.

12.4.3. Buffer schließen

Buffer werden unabhängig von Fenstern verwaltet. Ein Fenster kann geschlossen werden, während der zugehörige Buffer weiter existiert – oder der Buffer selbst wird aus der Liste entfernt. Für letzteres sind die `:bd`-Kommandos zuständig:

- `:bd` schließt den *aktuellen* Buffer. Das Fenster bleibt bestehen und zeigt danach einen anderen verfügbaren Buffer (oder wird ggf. leer / unlisted).
- `:bd <Nummer>` schließt explizit den Buffer mit der angegebenen Nummer, wie sie in `:ls` / `:buffers` angezeigt wird.
- `:bd!` erzwingt das Schließen, selbst wenn ungespeicherte Änderungen vorhanden sind. Achtung: Änderungen gehen dabei unwiderruflich verloren.

Wichtig: `:q` schließt nur das *Fenster*, nicht den Buffer. Der Buffer bleibt so lange in der Liste, bis `:bd` (oder eine Variante) ausgeführt wird. Das erklärt, warum `:ls` oft mehr Buffer zeigt, als Fenster geöffnet sind.

Übungen

1. Öffne drei Dateien (z. B. `a.txt`, `b.txt`, `c.txt`) und verschaffe dir mit `:ls` einen Überblick über die Buffer-Liste. Schließe dann nacheinander:
   - den aktuellen Buffer mit `:bd`
   - einen konkreten Buffer mit `:bd 1`

   Beobachte jeweils, wie sich die Markierungen „%“, „#“ und die Flags „a“, „h“ in der Liste verändern.
2. Erzeuge eine testweise Änderung in einem Buffer und versuche, ihn mit `:bd` zu schließen. Notiere dir die Fehlermeldung und wiederhole den Versuch mit

`:bd!`. Ziel: Klar verstehen, wann VIM Änderungen schützt – und wann du bewusst darüber hinweggehst.

3. Öffne zwei Splits, die denselben Buffer zeigen, und führe in einem Fenster `:bd` aus. Beobachte:
   - was mit dem zweiten Fenster passiert,
   - welchen Buffer es danach anzeigt (z. B. „[No Name]“ oder einen anderen aktiven Buffer).

   So wird deutlich, dass Fenster und Buffer getrennte Konzepte sind.

12.4.4. Neuen Buffer erzeugen

Nicht jeder Buffer entsteht durch `:edit` einer bestehenden Datei. Häufig möchte man *zuerst* einen leeren Arbeitsbereich haben und erst später entscheiden, ob und wo gespeichert wird. Dafür stellt VIM spezielle Befehle bereit:

- `:enew` öffnet einen neuen, leeren Buffer im aktuellen Fenster. Der Buffer hat zunächst keinen Dateinamen („[No Name]“).
- `:new` erzeugt einen horizontalen Split mit einem *neuen* leeren Buffer. Oben bleibt der bisherige Inhalt, unten erscheint der neue Buffer.
- `:vnew` erzeugt einen vertikalen Split mit einem leeren Buffer im neuen Fenster (links oder rechts, je nach Konfiguration).

Alle drei Varianten erzeugen einen *Buffer ohne Dateibindung*. Erst durch `:w <Datei>` oder `:saveas <Datei>` entsteht eine Datei auf der Platte und der Buffer erhält einen Dateinamen.

Typische Einsatzszenarien

- spontane Notizzettel, Skizzen, Scratchpads (`:enew`)
- temporäre Zwischenablagen für Refactorings
- parallel ein Konzept (oben) und eine Umsetzung (unten/rechts) bearbeiten, ohne sofort Dateien anzulegen

Übungen

1. Führe nacheinander im selben VIM:
   - `:enew`
   - `:ls`

Beobachte, welcher neue Eintrag in der Bufferliste erscheint und wie er gekennzeichnet ist (Name, Flags).

2. Erzeuge einen horizontalen Split mit `:new` und tippe in den neuen Buffer einige Zeilen Text. Benenne den Buffer dann mit:
   - `:file scratch.txt`
   - `:w`

   Prüfe mit `:ls`, wie sich der Name in der Bufferliste geändert hat.
3. Öffne mit `:vnew` einen vertikalen Split, schreibe etwas Text hinein und schließe den Buffer anschließend mit `:bd`. Beobachte:
   - welches Fenster danach aktiv ist,
   - welche Buffer in `:ls` noch vorhanden sind.

   Ziel: Verstehen, dass „neue Buffer“ losgelöst von Dateien existieren und erst durch Speichern dauerhaft werden.

12.4.5. Buffer benennen

Ungespeicherte Buffer haben zunächst keinen Dateinamen (VIM zeigt z.B. „[No Name]“ in der Statuszeile). Bevor man sie speichert, kann (und sollte) man ihnen einen sinnvollen Namen geben.

- `:file <Name>` setzt den Dateinamen des aktuellen Buffers auf `<Name>`, *ohne* sofort zu speichern.
- `:file` zeigt den aktuellen Dateinamen des Buffers an. Praktisch, um zu kontrollieren, in welcher Datei man gerade arbeitet.

Wichtig: `:file <Name>` ändert nur die *Zuordnung* des Buffers zu einem Dateinamen. Erst ein anschließendes `:w` (oder `:w <Name>`) schreibt den Inhalt wirklich auf die Platte. Das Verzeichnis muss existieren; VIM legt keine Verzeichnisse automatisch an.

Typische Szenarien:

- „Scratch“-Buffer (`:enew`) im Nachhinein sinnvoll benennen.
- Temporär erstellte Notizen später als reguläre Datei sichern.
- Einen Namen korrigieren, bevor die Datei das erste Mal geschrieben wird.

Übungen

1. Erzeuge mit `:enew` einen neuen leeren Buffer und schreibe ein paar Zeilen Text hinein. Gib dann nacheinander ein:
   - `:file`

   - `:file scratch.txt`
   - `:file`

   Beobachte, wie sich die Anzeige des Buffernamens ändert und dass `:file` *nicht* automatisch speichert.
2. Benenne denselben Buffer nun explizit und speichere ihn:
   - `:file notes/todo.txt`
   - `:w`

   Prüfe anschließend im Dateisystem, ob die Datei angelegt wurde. Falls das Verzeichnis `notes/` nicht existiert, lege es vorher an (z.B. `:!mkdir -p notes`) und wiederhole den Versuch.
3. Öffne eine bestehende Datei, ändere sie leicht und führe:
   - `:file backup.txt`
   - `:w`

   Überlege:
   - Was ist mit der ursprünglichen Datei passiert?
   - Auf welchen Namen zeigt der Buffer jetzt?

   Ziel: verstehen, dass `:file` die *Zuordnung* des Buffers ändert und damit steuert, wohin `:w` den Inhalt schreibt.

12.4.6. Buffer speichern

Das Speichern eines Buffers ist eng mit seinem aktuellen Dateinamen verknüpft. VIM unterscheidet dabei klar zwischen „unter bekanntem Namen schreiben“ und „unter neuem Namen ablegen“.

- `:w` speichert den aktuellen Buffer unter seinem bestehenden Dateinamen. Existiert die Datei noch nicht, wird sie neu angelegt.
- `:w <Datei>` schreibt den Inhalt des aktuellen Buffers in die angegebene `<Datei>`, *ändert aber den Buffernamen nicht.* Der Buffer bleibt also weiterhin mit dem alten Dateinamen verknüpft. Praktisch als „Sicherungskopie“ oder Export.
- `:saveas <Datei>` speichert den Buffer unter `<Datei>` *und* setzt den Buffernamen dauerhaft auf diesen neuen Pfad. Weitere `:w`-Befehle schreiben dann in Zukunft nach `<Datei>`.

Faustregel:

- `:w` — normaler Save.
- `:w <Datei>` — einmalige Kopie / „Schreib nach dort, aber bleib hier“.
- `:saveas <Datei>` — „ab jetzt heiße ich so“.

Übungen

1. Öffne eine bestehende Datei, ändere etwas Text und führe:
   - `:w`

   aus. Prüfe im Dateisystem, dass die ursprüngliche Datei aktualisiert wurde. Überlege: Was passiert, wenn du `:q!` statt `:q` eingegeben hättest?
2. Öffne eine Datei `config.yaml`, ändere sie und führe:
   - `:w config.backup.yaml`
   - `:file`

   aus. Beantworte für dich:
   - Unter welchem Namen ist der Buffer weiterhin bekannt?
   - Welche Dateien existieren jetzt auf der Platte?
3. Erzeuge mit `:enew` einen neuen Buffer, schreibe ein paar Zeilen Notizen hinein und führe dann:
   - `:saveas notes/vim-notizen.txt`
   - `:file`

   Beobachte, dass der Buffer nun dauerhaft mit `notes/vim-notizen.txt` verknüpft ist und zukünftige `:w`-Befehle genau dorthin schreiben. Ziel: den Unterschied zwischen `:w <Datei>` und `:saveas <Datei>` im Gefühl haben.

12.4.7. Versteckte Buffer (hidden buffers)

Standardmäßig verweigert VIM das Wechseln, wenn in einem Buffer ungespeicherte Änderungen vorhanden sind. Typische Meldung:

```
E37: No write since last change
```

Das ist der „sichere“ Modus: Bevor du zu einem anderen Buffer oder einer anderen Datei wechselst, sollst du dich bewusst entscheiden, ob du speichern willst oder nicht.

Mit:

- `:set hidden`

dürfen Buffer im Hintergrund existieren, ohne gespeichert oder geschlossen zu werden. Du kannst also:

- den Buffer verlassen, obwohl er ungespeichert ist,
- zu einem anderen Buffer springen (`:bnext`, `:buffer~<Nummer>`),
- andere Dateien öffnen (`:edit~<Datei>`),

und VIM hält den alten Buffer mit allen Änderungen im Speicher. In der `:ls`-Liste erkennst du solche Buffer an den Markierungen `h` (hidden) und + (geändert).

Vor- und Nachteile

- Vorteil: Du kannst gedanklich „Tabs“ wechseln, ohne ständig speichern zu müssen. Besonders angenehm bei vielen Files oder Experimenten.
- Nachteil: Es ist leichter, geänderte Buffer zu vergessen. In großen Sitzungen kann es passieren, dass mehrere nicht gespeicherte Stände „herumliegen“, ohne dass du es sofort bemerkst.

Praktische Ergänzungen:

- `:set hidden?` — zeigt, ob `hidden` derzeit aktiv ist.
- `:set nohidden` — kehrt zum „strengen“ Standardverhalten zurück.

Übungen

1. Öffne zwei Dateien, z. B.:
   - `:edit a.txt`
   - `:split b.txt`

   Ändere in `a.txt` eine Zeile und versuche dann ohne Speichern mit `:bnext` oder `:edit c.txt` zu wechseln. Beobachte die Fehlermeldung.
2. Aktiviere nun:
   - `:set hidden`

   Wiederhole das Szenario: Ändere `a.txt`, wechsle mit `:bnext` oder `:edit c.txt` zu einer anderen Datei und rufe anschließend `:ls` auf. Achte auf die Kennzeichnungen + und `h`.
3. Arbeite eine Weile mit `:set hidden` und nutze zwischendurch:
   - `:wa`

   um alle geänderten Buffer zu speichern. Ziel: Ein Gefühl dafür bekommen, wie viele geänderte, aber ungespeicherte Buffer sich ansammeln können — und wie `:ls` und `:wa` dir helfen, den Überblick zu behalten.

12.4.8. Buffer gleichzeitig anzeigen

Buffer können in mehreren Fenstern gleichzeitig sichtbar sein. Dabei zeigt jedes Fenster *denselben* Buffer, aber an einer eigenen Position (eigener Cursor, eigener Scroll-Offset).

Typische Befehle:

- `:split <Datei>` — horizontales Split mit der angegebenen Datei (neuer Buffer oder bereits bestehender Buffer, falls schon geöffnet).
- `:vsplit <Datei>` — vertikales Split mit der Datei rechts vom aktuellen Fenster.
- `:sbuffer <Nummer>` — zeigt einen bereits existierenden Buffer mit der angegebenen Nummer im neuen Split an, ohne den alten Fensterinhalt zu ersetzen.
- `:vert sbuffer <Nummer>` — wie `:sbuffer`, aber direkt als vertikaler Split.

Typische Szenarien:

- oben: Funktionsdefinition, unten: deren Aufrufstellen im selben Buffer,
- links: aktueller Arbeitsbereich, rechts: Referenz-Stelle oder Kommentar,
- zwei Ausschnitte derselben Log-Datei an verschiedenen Stellen (Anfang/Ende).

Wichtig ist dabei:

- Änderungen im einen Fenster sind *sofort* im anderen sichtbar (es ist derselbe Buffer, kein Duplikat),
- jeder Split hat seinen eigenen Cursor und eigenen Viewport,
- alle Schreib-/Speicherbefehle (`:w`, `:wa`, `:x`) wirken immer auf den aktuellen Buffer, egal in welchem Fenster er gezeigt wird.

Übungen

1. Öffne eine Datei, z. B. `main.c`, und teile das Fenster:
   - `:edit main.c`
   - `:split`

   Scrolle im oberen Fenster zum Dateianfang, im unteren zum Dateiende. Beobachte, dass beide Fenster denselben Buffer anzeigen, aber an unterschiedlichen Positionen.
2. Zeige einen zweiten Buffer parallel an:
   - `:ls` — Buffernummern merken
   - `:vsplit`
   - `:sbuffer <Nummer>` — wähle einen anderen Buffer

   Prüfe, wie sich nun drei Fenster verhalten (zwei mit Buffer A, eines mit Buffer B).
3. Öffne wiederholt denselben Buffer in mehreren Splits (z. B. mit `:vsplit` und anschließend `:buffer <Nummer>`). Ändere eine Zeile in einem Fenster

und beobachte *live*, wie die Änderung beim Neuzeichnen in allen anderen Fenstern sichtbar wird. Ziel: Verinnerlichen, dass Fenster nur „Ansichten“ sind, während der Buffer der eigentliche Datenträger ist.

12.4.9. Buffer löschen ohne Projektverlust

Beim Schließen eines Buffers mit `:bd` prüft VIM, ob noch ungespeicherte Änderungen vorhanden sind. Ist das der Fall, erscheint häufig:

```
E89: No write since last change
```

Diese Meldung bedeutet: VIM verweigert das Löschen des Buffers, um keinen Datenverlust zu riskieren.

In solchen Situationen gibt es drei saubere Wege:

- Änderungen bewusst verwerfen:
  - `:bd!` — Buffer schließen und Änderungen verwerfen
- Vorher speichern und dann löschen:
  - `:w` — Buffer speichern
  - `:bd` — danach gefahrlos schließen
- Speichern *und* sofort löschen in einem Schritt:
  - `:w \textbar bd` — schreibt die Datei und löscht danach den Buffer

Damit gilt: Ein „Projektverlust“ findet nur statt, wenn man sich bewusst für `:bd!` entscheidet. Alle anderen Varianten sorgen dafür, dass die Inhalte sicher auf Disk landen, bevor der Buffer aus der Liste entfernt wird.

Übungen

1. Öffne eine Datei, ändere eine Zeile und gib `:bd` ein. Beobachte die Fehlermeldung `E89: No write since last change`. Speichere anschließend mit `:w` und wiederhole `:bd`.
2. Wiederhole das Szenario, nutze diesmal aber direkt:
   - `:w \textbar bd`

   Prüfe mit `:ls`, dass der Buffer verschwunden ist und die Datei auf Disk aktualisiert wurde.
3. Öffne zwei Fenster mit derselben Datei (z. B. `:vsplit`). Schließe in einem Fenster nur das Fenster mit `:q` und im anderen explizit den Buffer mit `:bd`. Ziel: Den Unterschied zwischen „Fenster zu“ und „Buffer weg“ verstehen.

12.5. Tabs (optional)

VIM unterstützt neben Fenstern und Splits auch Tabs. Im Gegensatz zu Browser-Tabs handelt es sich hierbei *nicht* um isolierte Arbeitsbereiche, sondern um Sammlungen von Fenstern. Jeder Tab kann beliebig viele Splits enthalten, und jeder Split zeigt wiederum einen eigenen Buffer.

Tabs sind also eher *Layout-Container* als „Dokumenten-Tabs“ im klassischen Sinne: Die Buffer bleiben global, nur die Anordnung der Fenster unterscheidet sich von Tab zu Tab.

12.5.1. Tabs erstellen

Es gibt mehrere Wege, neue Tabs anzulegen:

- `:tabnew` öffnet einen neuen Tab mit einem *leeren* Buffer.
- `:tabnew <Datei>` öffnet einen neuen Tab und lädt die angegebene Datei in diesem Tab.
- `:tabedit <Datei>` öffnet die Datei direkt in einem neuen Tab (ggf. bestehenden Buffer wiederverwenden).
- `:tabe <Datei>` Kurzform von `:tabedit <Datei>`.

Wichtig:

- Tabs sind *zusätzliche Ansichten* auf die ohnehin vorhandenen Buffer. Eine Datei kann gleichzeitig in mehreren Tabs (und dort in mehreren Splits) geöffnet sein.
- Splits werden *innerhalb* eines Tabs erzeugt. Jeder Tab hat also sein eigenes Fensterlayout.

Typische Einsatzszenarien:

- Tab 1: Projektübersicht, z. B. Dateibrowser und Hauptkonfigurationsdatei.
- Tab 2: Quelltext links, Tests rechts (vertikale Splits).
- Tab 3: Log-Datei oben, Terminal unten.

Übungen

1. Öffne eine Datei `main.c` und erzeuge anschließend:
   - einen neuen Tab mit `:tabnew`
   - danach einen weiteren Tab mit `:tabnew main.c`

   Beobachte mit `:ls`, dass `main.c` nur *ein Buffer* ist, aber in mehreren Tabs angezeigt werden kann.

2. Erzeuge nacheinander drei Tabs mit verschiedenen Dateien (z. B. `config.yaml`, `app.log`, `README.md`) über `:tabedit <Datei>`. Navigiere anschließend mit `:tabnext` / `:tabprev` bzw. **g** **t** / **g** **T** zwischen den Tabs und beobachte, wie sich das Fensterlayout pro Tab unterscheidet.
3. Öffne im ersten Tab einen vertikalen Split (`:vsplit`) und im zweiten Tab einen horizontalen Split (`:split`). Wechsle mehrfach zwischen den Tabs und stelle fest, dass jeder Tab sein *eigenes* Split-Layout beibehält, obwohl teilweise dieselben Buffer angezeigt werden.

12.5.2. Tabs wechseln

Sind mehrere Tabs geöffnet, wird der schnelle Wechsel zwischen ihnen wichtig. Grundsätzlich gibt es zwei Wege: Ex-Befehle und Normal-Mode-Kommandos.

- `:tabnext` zeigt den *nächsten* Tab an (zyklisch, d. h. vom letzten wieder zum ersten).
- `:tabprev` zeigt den *vorherigen* Tab an.
- `:tabfirst` springt direkt zum ersten Tab.
- `:tablast` springt direkt zum letzten Tab.
- **g** **t** wechselt im Normalmodus *vorwärts* zum nächsten Tab (entspricht `:tabnext`).
- **g** **T** wechselt im Normalmodus *rückwärts* zum vorherigen Tab (entspricht `:tabprev`).

Im Alltag sind **g** **t** und **g** **T** oft die schnellste Variante, weil sie ohne Doppelpunkt und ohne Kommandozeile auskommen.

Praktische Ergänzungen, die viele Nutzer in ihrer `vimrc` pflegen:

```
nnoremap <Leader>1 1gt
nnoremap <Leader>2 2gt
nnoremap <Leader>3 3gt
```

Damit wird z. B. <leader>1 zum direkten Sprung auf Tab 1.

Übungen

1. Öffne drei Dateien in separaten Tabs, z. B.:
   - `:tabedit main.c`

   - `:tabedit config.yaml`
   - `:tabedit README.md`

   Wechsle anschließend mehrfach mit g t und g T durch die Tabs und beobachte die Tab-Reihenfolge in der Statuszeile oder GUI-Leiste.
2. Nutze bewusst die Ex-Befehle:
   - `:tabfirst`
   - `:tablast`
   - `:tabnext`
   - `:tabprev`

   Ziel: Verstehen, wie sich die zyklische Navigation verhält, wenn nur zwei, drei oder viele Tabs geöffnet sind.
3. Falls du eine `vimrc` pflegst, richte testweise ein einfaches Mapping ein:

```
nnoremap <Leader>1 1gt
```

   Starte VIM neu, öffne mehrere Tabs und prüfe, ob du mit <leader>1 zuverlässig auf den ersten Tab springen kannst.

12.5.3. Tabs schließen

Tabs lassen sich unabhängig von Buffern schließen – das Layout verschwindet, die zugrunde liegenden Dateien (Buffer) bleiben erhalten.

- `:tabclose` schließt den *aktuellen* Tab. Alle darin geöffneten Fenster (Splits) werden geschlossen, die zugehörigen Buffer bleiben jedoch im Hintergrund erhalten und können später wieder angezeigt werden.
- `:tabclose 3` schließt explizit Tab Nummer 3 (Tab-Indizes beginnen bei 1).
- `:tabonly` schließt *alle anderen* Tabs und lässt nur den aktuellen Tab bestehen. Praktisch, wenn man viele Tabs geöffnet hat und wieder zu einem einzigen Arbeitsbereich zurückkehren möchte.

Wichtig: Tab-Schließen wirkt sich nicht auf die dahinterliegenden Buffer aus – die Dateien bleiben geöffnet und können z. B. mit `:buffer`, `:bnext` oder über neue Tabs/Fenster jederzeit wieder sichtbar gemacht werden.

Übungen

1. Öffne drei Dateien in separaten Tabs:

   - `:tabedit a.txt`
   - `:tabedit b.txt`
   - `:tabedit c.txt`

   Schließe anschließend den mittleren Tab mit `:tabclose 2` und prüfe mit `:ls`, dass alle drei Buffer weiterhin existieren.
2. Öffne erneut mehrere Tabs und führe dann:
   - `:tabonly`

   aus. Beobachte, dass nur der aktuelle Tab bleibt, die Buffer der anderen Tabs aber weiterhin über `:buffer` erreichbar sind.
3. Kombiniere Tabs und Buffer-Navigation: Erzeuge zwei Tabs mit jeweils mehreren Buffern (z. B. durch `:split` und `:edit`). Schließe einen Tab mit `:tabclose` und hole anschließend einen der darin angezeigten Buffer im verbleibenden Tab mit `:buffer <Nummer>` zurück.

12.5.4. Tabs reorganisieren

VIM erlaubt es, Tabs flexibel umzunummerieren und ihre Reihenfolge anzupassen. Die Position eines Tabs wird über `:tabmove` gesteuert:

- `:tabmove` verschiebt den aktuellen Tab an das *Ende* der Tab-Liste.
- `:tabmove 0` verschiebt den aktuellen Tab an den *Anfang* (erste Position).
- `:tabmove 3` verschiebt den aktuellen Tab auf Position 3. Die Zählung beginnt bei 0, d. h.:
  - 0 = erster Tab
  - 1 = zweiter Tab
  - 2 = dritter Tab
  - 3 = vierter Tab

Praktischer Merksatz: `:tabmove N` positioniert den aktuellen Tab *vor* dem Tab mit Index N. Ohne Argument wird er ans Ende verschoben.

Übungen

1. Öffne drei Tabs:
   - `:tabnew a.txt`
   - `:tabnew b.txt`
   - `:tabnew c.txt`

   und beobachte die Reihenfolge in der Tab-Leiste. Verschiebe dann den mittleren Tab mit:

   - `:tabmove 0`

   und überprüfe, dass dieser Tab nun ganz links steht.
2. Erzeuge vier Tabs (z. B. `a.txt` bis `d.txt`) und führe nacheinander aus:
   - `:tabmove`
   - `:tabmove 1`

   Notiere dir vor jedem Kommando die Reihenfolge der Tabs und vergleiche sie anschließend.
3. Nutze `:tabmove` in Kombination mit `:tabnext` / `:tabprev`, um dir einen „Arbeits-Tab“ immer auf Position 0 zu holen und weniger wichtige Tabs ans Ende zu verschieben. Ziel: Eine Tab-Ordnung finden, in der du wichtige Dateien schnell wiederfindest.

12.5.5. Tabs und Fensterlayouts

Tabs entfalten ihre Stärke vor allem dann, wenn sie als „Layout-Container“ für unterschiedliche Fensteranordnungen genutzt werden. Ein typisches Setup könnte so aussehen:

Tab 1: Projektübersicht (`:Ex`)
Tab 2: Code links, Tests rechts (Split)
Tab 3: Terminal unten (`:split \textbar {} terminal`)

Dabei gilt:

- Jeder Tab kann ein eigenes Fensterlayout (Splits) enthalten.
- In jedem Fenster kann ein anderer (oder derselbe) Buffer angezeigt werden.
- Tabs trennen die Layouts logisch voneinander, ohne die Buffer selbst zu duplizieren.

Praktische Einsatzszenarien:

- **Projekt-Tab:** oben Dateiübersicht mit `:Ex` oder einem Dateibrowser-Plugin, darunter zentrale Konfigurations- oder Einstiegspunkte des Projekts.
- **Code-und-Test-Tab:** links der Produktionscode (z. B. `main.c`), rechts die zugehörigen Tests oder Logs. Durch Splits kann man Änderungen und Auswirkungen direkt beobachten.
- **Terminal-Tab:** unten ein Terminal-Split für `:make`, `:pytest`, `:npm test` o. Ä., oben der relevante Code. So bleiben Build-Ausgaben getrennt vom eigentlichen Editier-Layout.

Wichtig ist die Trennung der Rollen:

- **Buffer** = Inhalte (Dateien, Terminal-Sessions, Scratch-Buffers)
- **Fenster** = Sicht auf einen Buffer (inkl. Cursorposition, Scrollzustand)
- **Tabs** = Sammlung von Fenstern (Layout-Einheiten)

Tabs sind ideal, wenn komplexe Arbeitsbereiche getrennt, aber schnell erreichbar sein sollen – zum Beispiel unterschiedliche Projektteile, verschiedene Git-Branches oder getrennte Kontexte (Code / Doku / Tools).

Übungen

1. Richte ein dreiteiliges Layout ein:
   - Tab 1: `:tabnew` und dann `:Ex` für die Projektübersicht.
   - Tab 2: `:tabnew main.c`, anschließend `:vsplit test\_main.c`.
   - Tab 3: `:tabnew main.c`, dann `:split` und im unteren Fenster `:terminal` starten.

   Wechsle mit `:gt` und `:gT` zwischen den Tabs und prüfe, dass *jedes* Tab sein eigenes Fensterlayout behält.
2. Wähle im „Code-und-Test"-Tab unterschiedliche Cursorpositionen in linkem und rechtem Fenster. Wechsle mehrfach zwischen Tabs hin und her. Beobachte, dass:
   - jede Fensterkombination ihre eigene Cursorposition behält,
   - derselbe Buffer in verschiedenen Tabs unterschiedliche „Sichtfenster" haben kann.
3. Experimentiere mit einem „Doku-Tab": Lege einen Tab mit ausschließlich Markdown- oder LaTeX-Dateien an, in dem du Notizen zum Projekt führst. Nutze in einem anderen Tab ausschließlich Code + Terminal. Ziel: Ein Gefühl dafür entwickeln, wie Tabs helfen, mentale Kontexte sauber zu trennen.

12.5.6. Tabs automatisch öffnen

Für Workflows mit festen Startdateien oder wiederkehrenden Projekten ist es praktisch, Tabs gleich beim Start oder per Ex-Befehl automatisch anzulegen.

- `:tab all` Öffnet alle Dateien aus der aktuellen Argumentliste jeweils in einem eigenen Tab. Typischer Ablauf:
  - `:args *.c`
  - `:tab all`

  Danach liegt jede `.c`-Datei in einem eigenen Tab.

- Shell: `vim -p *.txt` Startet VIM im „Parallel-Tab-Modus“: jede übergebene Datei wird sofort in einem eigenen Tab geöffnet.

Der Parameter `-p` weist VIM an, für jedes übergebene Argument einen Tab anzulegen, statt alles in einem einzigen Fenster/Tab zu öffnen. Analog dazu existiert `-o` für horizontale Splits (statt Tabs).

Übungen

1. Lege im Terminal ein Testverzeichnis mit mehreren Dateien an (z. B. `a.txt`, `b.txt`, `c.txt`) und starte:

   ```
   vim -p *.txt
   ```

   Prüfe in VIM mit `:tabs`, wie viele Tabs geöffnet wurden, und wechsle mit `:gt` und `:gT` zwischen ihnen.
2. Öffne VIM zunächst nur mit einer Datei, z. B.:

   ```
   vim main.c
   ```

   Richte dann eine Argumentliste ein und verteile diese Dateien auf Tabs:
   - `:args *.c`
   - `:tab all`

   Kontrolliere mit `:args` und `:tabs`, wie Argumentliste und Tabs zusammenhängen.
3. Vergleiche Tabs mit Splits: Starte einmal mit:

   ```
   vim -p main.c util.c config.c
   ```

   und ein anderes Mal mit:

   ```
   vim -o main.c util.c config.c
   ```

   Beobachte die Unterschiede:
   - Wie verändert sich das Layout?
   - Wann sind Tabs angenehmer, wann Splits?

   Ziel: Ein Gefühl entwickeln, wann `-p` (Tabs) und wann `-o` (Splits) besser zum eigenen Workflow passen.

13. Konfiguration und vimrc

VIM besteht grob aus zwei Schichten: dem *Programm* selbst und der *Konfiguration*, die beim Start eingelesen wird. Die Binärdatei (das eigentliche VIM) bringt eine Menge Default-Verhalten mit, aber erst durch Optionen, Mappings, Autocommands und Plugins entsteht dein persönlicher Editor.

Dieses Kapitel erklärt systematisch:

- was VIM beim Start in welcher Reihenfolge lädt,
- wie die „klassische“ Konfigurationsdatei `vimrc` aufgebaut sein kann,
- wie Optionen, Mappings, Autocommands, Dateityp-spezifische Einstellungen und Plugins zusammenspielen,
- wie man eine eigene Konfiguration strukturiert, debuggt und pflegt.

Ziel ist nicht eine „magische“ Copy -& Paste-`vimrc`, sondern ein Verständnis dafür, *warum* VIM sich so verhält, wie es sich verhält – und wo du gezielt eingreifen kannst.

13.1. Grundlagen der VIM-Konfiguration

VIM ist kein Monolith mit fest verdrahtetem Verhalten, sondern eine Laufzeitumgebung, die beim Start eine Reihe von Skripten und Konfigurationsdateien ausführt. Wichtige Punkte:

- Das Programm `vim` ist nur der Kern (Befehle, Modi, Ex-Engine).
- Alles Konfigurierbare (Optionen, Mappings, Plugins, Syntax-Regeln) wird zur Laufzeit über Skripte geladen.
- Die Reihenfolge, in der diese Skripte ausgeführt werden, entscheidet, welche Einstellung am Ende „gewinnt“.

Daraus folgt: Wer VIM verstehen will, muss wissen, *was* beim Start geladen wird und *woher* bestimmte Einstellungen kommen. Genau das leistet die `vimrc` in Kombination mit der VIM-Runtime.

13.1.1. Was VIM beim Start lädt

Beim Start läuft VIM eine feste Abfolge von Konfigurationsquellen ab. Vereinfacht kann man sich das so merken:

1. **Programm-Defaults** Alles, was beim Compilieren festgelegt wurde (Features, Standardwerte vieler Optionen).
2. **Systemweite VIM-Konfiguration** Eine Datei wie `$VIM/vimrc` oder `/etc/vim/vimrc` legt globale Voreinstellungen für alle Benutzer fest. Typisch sind hier Dinge wie:
   - Aktivierung von `:syntax on`
   - globale `:set`-Optionen
   - systemweite Plugins oder Mappings
3. **Benutzerspezifische** `vimrc` Anschließend liest VIM die persönliche Konfigurationsdatei, z. B.:
   - `$HOME/.vimrc` (klassisch)
   - oder Plattformvarianten wie `%USERPROFILE%\_vimrc` unter Windows

   Alles, was hier steht, *überschreibt* systemweite Defaults.
4. **Optionale GUI-Konfiguration (`gvimrc`)** Wenn eine grafische Version (GVim) läuft, wird zusätzlich eine `gvimrc` ausgewertet, z. B. für Menüs, Fonts, GUI-Farben.
5. **Runtime-Pfad und Skripte** Danach arbeitet VIM den `runtimepath` ab:
   - `plugin/*.vim` – globale Plugins beim Start
   - `ftdetect/*.vim` – Erkennung von Dateitypen
   - `ftplugin/*.vim` – dateitypspezifische Einstellungen
   - `indent/*.vim` – Einrückungslogik pro Dateityp
   - `after/`-Verzeichnisse – „späte“ Overrides

   Hier greifen auch Plugin-Manager und zusätzliche Konfigurationsdateien.
6. **Dateityp, Syntax, Farben** Auf Basis von Dateinamenerweiterung, Shebang-Zeile oder Dateiinhalten setzt VIM den `filetype`, lädt passende Syntax-Definitionen und ggf. dein `:colorscheme`.

Zwei Diagnosewerkzeuge sind hier extrem hilfreich:

- `:version` zeigt u. a. an, wo VIM die systemweite `vimrc` und die benutzerspezifische `vimrc` erwartet (Zeilen wie „system vimrc file“ / „user vimrc file“).

- `:scriptnames` listet alle bisher geladenen Skripte in genau der Reihenfolge, in der sie ausgeführt wurden – inklusive deiner `vimrc`, Plugins und ftplugin-Dateien.

Damit kannst du im Problemfall nachvollziehen:

- *woher* eine bestimmte Einstellung kommen könnte,
- *ob* deine `vimrc` überhaupt gelesen wurde,
- *welche* Plugins oder Skripte bereits aktiv sind.

Übungen

1. Starte VIM ohne zusätzliche Argumente und führe nacheinander aus:
   - `:version`
   - `:echo \$MYVIMRC`

   Notiere dir:
   - welchen Pfad VIM als „user vimrc file“ anzeigt,
   - ob `$MYVIMRC` gesetzt ist und wohin es zeigt.

   Ziel: Du weißt, *welche* Datei deine persönliche Konfiguration ist und wo sie liegt.
2. Führe `:scriptnames` aus und scrolle mit **j**/**k** durch die Liste. Beantworte für dich:
   - Wird deine `vimrc` überhaupt geladen?
   - Welche `plugin/`-, `ftplugin/`- oder `indent/`-Dateien siehst du?

   Ziel: Ein Gefühl für die tatsächliche Lade-Reihenfolge bekommen.
3. Starte VIM einmal mit und einmal ohne Konfiguration:
   - `vim` (normal)
   - `vim -u NONE -U NONE` (ohne `vimrc/gvimrc`)

   Öffne dieselbe Datei und vergleiche:
   - Welche Optionen (z. B. Zeilennummern, Farben, Tabbreite) sind nur mit deiner `vimrc` aktiv?
   - Welche Verhaltensweisen stammen also eindeutig aus deiner eigenen Konfiguration?

   Ziel: Klar trennen können, was „VIM pur“ ist und was deine „Persönlichkeits-Schicht“ der Konfiguration oben drauf setzt.

13.1.2. vimrc: Speicherort und Varianten

Die zentrale Konfigurationsdatei von VIM wird traditionell `vimrc` genannt. Je nach Plattform und Version unterscheidet sich der genaue Speicherort – und bei Neovim heißt die Hauptdatei sogar anders. Wichtig ist daher, die *Varianten* zu kennen und zu wissen, welche Datei in deinem Setup tatsächlich benutzt wird.

Klassischer VIM (`.vimrc`)

Unter Unix-ähnlichen Systemen (Linux, BSD, macOS im Terminal) sucht VIM standardmäßig im Home-Verzeichnis nach:

- `$HOME/.vimrc`

Auf Windows heißen die entsprechenden Dateien typischerweise:

- `%USERPROFILE%\_vimrc`
- oder `%VIM%\_vimrc`

Die `.vimrc` bzw. `_vimrc` ist die *persönliche* Benutzerkonfiguration. Alles, was hier steht, hat Vorrang vor systemweiten Defaults und Voreinstellungen aus `/etc/vim/vimrc` oder `$VIM/vimrc`.

Zur Kontrolle kannst du in VIM prüfen:

- `:echo \$MYVIMRC` – zeigt die aktuell verwendete `vimrc`
- `:version` – listet u. a. „user vimrc file“ und „system vimrc file“ auf

Neovim: `init.vim` und `init.lua`

Neovim verwendet eine andere Konfigurationsstruktur. Die Standarddatei heißt nicht `.vimrc`, sondern:

- `$HOME/.config/nvim/init.vim` (klassischer VIM-Stil)
- oder `$HOME/.config/nvim/init.lua` (Lua-basierte Konfiguration)

Wichtig:

- Neovim liest `.vimrc` *nicht automatisch* ein.
- Viele VIM-Einstellungen funktionieren in `init.vim` unverändert, aber Pfade, Plugin-Manager und manche Optionen unterscheiden sich.

Für dieses Handbuch nehmen wir `VIM` im klassischen Sinn als Referenz. Die meisten Beispiele lassen sich auf Neovim übertragen, wenn man den anderen Konfigurationspfad beachtet.

GUI-spezifische Konfiguration: `gvimrc`

Für grafische Varianten (GVim, MacVim etc.) gibt es eine zusätzliche GUI-Konfiguration:

- `$HOME/.gvimrc` (Unix/macOS)
- `%USERPROFILE%\_gvimrc` (Windows)

Dort landen typischerweise Einstellungen wie:

- Fenstergröße, Fonts, GUI-Farben,
- Menü-/Toolbar-Anpassungen,
- Mausverhalten in der GUI.

Die `gvimrc` wird *zusätzlich* zur normalen `vimrc` geladen. Allgemeine Einstellungen (Mappings, Optionen, Plugins) gehören in die `vimrc`, reine GUI-Details in die `gvimrc`.

Systemweite vs. benutzerspezifische Konfiguration

Neben der persönlichen `vimrc` existieren oft systemweite Dateien, z. B.:

- `/etc/vim/vimrc`
- `$VIM/vimrc`

Typische Verwendung:

- Systemweite Defaults (z. B. `:syntax on`, Tabs vs. Spaces)
- Distribution-spezifische Anpassungen (z. B. Debian-/Ubuntu-Defaults)

Die Reihenfolge ist grob:

1. Programm-Defaults
2. systemweite `vimrc`
3. benutzerspezifische `vimrc`
4. optional: `gvimrc`

Alles, was später geladen wird, kann frühere Einstellungen überschreiben.

Übungen

1. Finde heraus, welche `vimrc` dein VIM tatsächlich verwendet:
   - Starte VIM und führe `:echo \$MYVIMRC` aus.
   - Öffne die angezeigte Datei mit `:edit \$MYVIMRC`.

   Ziel: Du kennst *genau* den Pfad deiner persönlichen Konfigurationsdatei.
2. Ermittle die systemweite Konfiguration:
   - Führe `:version` aus und suche nach den Zeilen „system vimrc file“ und „user vimrc file“.

- Öffne die systemweite Datei (falls vorhanden) mit `:edit` und schau nach, welche Voreinstellungen dort gesetzt sind.

Ziel: Verstehen, was vom System vorgegeben ist und was du selbst überschreiben kannst.

3. Falls du Neovim installiert hast:
   - Starte `nvim` und führe `:echo stdpath('config')` aus.
   - Öffne `init.vim` oder `init.lua` in diesem Verzeichnis.

   Ziel: Den Unterschied zwischen VIM-`vimrc` und Neovim-`init.*` praktisch sehen und sauber gedanklich trennen.

13.1.3. Konfigurationsprinzipien

Eine „funktionierende“ `vimrc` ist schnell gebaut – eine *lesbare* und *wartbare* `vimrc` dagegen erfordert etwas Disziplin. Ziel sollte immer sein, dass du deine Konfiguration auch in ein paar Monaten noch verstehst und gezielt erweitern oder entschlacken kannst.

Lesbarkeit statt Einzeiler-Chaos

Technisch ist es möglich, sehr viel in eine einzige Zeile zu packen:

```
set nu et sts=2 sw=2 ts=2|nnoremap <Space> :noh<CR>
```

Lesbar ist das nicht. Besser ist eine klare, zeilenweise Struktur:

```
" Anzeige
set number

" Einrückung
set expandtab
set shiftwidth=2
set softtabstop=2
set tabstop=2

" Tastenkürzel
nnoremap <Space> :nohlsearch<CR>
```

Grundprinzipien:

- *Eine* Einstellung pro Zeile.
- Aussagekräftige Kommentare vor oder über Blöcken.
- Keine „Magie“, die du dir nicht selbst erklären kannst.

Logische Blöcke bilden

Statt eine wachsende Liste von `:set`-Befehlen zu haben, lohnt sich eine Gliederung in logische Abschnitte, z. B.:

- Benutzeroberfläche (Nummern, Statuszeile, Farbthema)
- Editierverhalten (Einrückung, Tabs vs. Spaces, Backspace-Regeln)
- Suche und Ersetzen (`:hlsearch`, `:ignorecase`, `:smartcase`)
- Navigation (Mappings für <leader>, Tag-Sprünge, Folds)
- Plugins (Plugin-Manager, Plugin-spezifische Konfiguration)
- Sprache/Encoding (UTF-8, Fileformats)

Typischer Aufbau in der `vimrc`:

```
" ==========================
" UI / Darstellung
" ==========================
set number
set showcmd
set laststatus=2

" ==========================
" Einrückung / Tabs
" ==========================
set expandtab
set shiftwidth=4
set softtabstop=4

" ==========================
" Suche
" ==========================
set ignorecase
set smartcase
set hlsearch
```

Wichtig ist nicht das *eine richtige* Schema, sondern dass du *konsistent* bleibst.

Kommentieren und dokumentieren

Kommentare sind in der `vimrc` extrem billig – und später Gold wert. Üblich sind Anführungszeichen am Zeilenanfang:

```
" Suchverhalten: case-insensitive, außer bei Großbuchstaben
set ignorecase
set smartcase
```

Bewährt haben sich:

- kurze Kopfzeilen für Blöcke,
- knappe Begründungen bei „ungewöhnlichen“ Einstellungen,
- Datum oder Kontext bei temporären Hacks.

Alles, was du beim Lesen denkst: „Warum ist das hier so?“, gehört eigentlich als Kommentar in die Datei.

Versionskontrolle und Backup

Eine `vimrc` wächst schnell – entsprechend wichtig ist es, den Verlauf nachvollziehen zu können:

- Lege dein `$HOME/.vim/` und deine `vimrc` in ein Git-Repository (`git init`, `git commit`).
- Nutze sinnvolle Commit-Messages („Folding-Einstellungen aufgeräumt“ statt „update“).
- Halte mindestens eine Kopie außerhalb des Systems bereit (Backup, dotfile-Repo, Cloud-Sync etc.).

So kannst du jederzeit:

- auf einen älteren Stand zurückgehen, wenn etwas „kaputt konfiguriert“ wurde,
- deine Konfiguration auf anderen Maschinen klonen,
- experimentelle Branches für neue Ideen anlegen.

Übungen

1. Öffne deine aktuelle `vimrc` und gruppiere alle `:set`-Befehle in mindestens drei Blöcke (z. B. „UI“, „Editieren“, „Suche“). Füge jeweils eine kurze Kopfzeile als Kommentar ein.
2. Suche nach Zeilen, in denen mehrere Einstellungen oder Kommandos mit `|` oder Semikolon verkettet sind. Teile diese bewusst auf mehrere Zeilen auf und kommentiere jede Gruppe kurz.

3. Lege ein Git-Repository für deine VIM-Konfiguration an:
   - Erstelle bei Bedarf ein Verzeichnis `$HOME/.vim/` (falls noch nicht vorhanden).
   - Initialisiere dort `git` und füge `vimrc` sowie ggf. `after/`, `ftplugin/` etc. hinzu.
   - Mache einen ersten Commit mit einer sinnvollen Nachricht.

   Ziel: Deine `vimrc` ist ab jetzt versioniert und du kannst jede spätere Änderung nachvollziehen.

13.2. Aufbau einer strukturierten vimrc

13.2.1. Header und Grundsetup

Der Kopf deiner `vimrc` legt die *Grundregeln* für alle weiteren Einstellungen fest. Was hier steht, beeinflusst:

- ob VIM sich „modern“ oder wie altes `vi` verhält,
- welches Encoding und Zeilenformat verwendet werden,
- ob Einstellungen aus Dateien („modelines“) ausgewertet werden,
- und welche Basisoptionen in *jedem* Projekt gelten.

Ein typischer, schlanker Header könnte so aussehen:

```
" ==========================
" Basis-Setup
" ==========================
set nocompatible
set encoding=utf-8
set fileencoding=utf-8
set fileformats=unix,dos
set nomodeline
```

Die folgenden Abschnitte erläutern diese Bausteine.

`compatible` vs. `nocompatible`

VIM kennt einen Kompatibilitätsmodus zu klassischem `vi`. Ist `:compatible` aktiv, werden viele moderne VIM-Funktionen deaktiviert oder eingeschränkt:

- weniger Undo-Möglichkeiten,

- eingeschränkte Mappings,
- teilweise andere Default-Werte.

Daher sollte *am Anfang* der `vimrc` fast immer stehen:

```
set nocompatible
```

Wichtig:

- `:set nocompatible` *muss* vor den meisten anderen Einstellungen kommen.
- Viele Distributionen setzen das bereits in der Systemkonfiguration, aber im eigenen Header ist es explizit und nachvollziehbar.

Encoding: UTF-8 als Standard

Ohne explizite Vorgabe hängt das Text-Encoding oft von Umgebung und System ab. Für moderne Setups ist UTF-8 die sinnvolle Voreinstellung:

```
set encoding=utf-8
set fileencoding=utf-8
set fileencodings=utf-8,latin1
```

Grobe Bedeutung:

- `:encoding` – internes Encoding von VIM (Speicher, Register, Messages).
- `:fileencoding` – bevorzugtes Encoding beim Speichern.
- `:fileencodings` – Liste, in der VIM beim Öffnen zu raten versucht (Reihenfolge ist wichtig).

Praxis-Tipp:

- Halte `:encoding` und `:fileencoding` in der Regel identisch.
- `:fileencodings` ergänzt du nur, wenn du tatsächlich mit alten Encodings (z. B. `latin1`) arbeiten musst.

Zeilenenden: `fileformat`

Unterschiedliche Betriebssysteme nutzen unterschiedliche Zeilenenden:

- `:fileformat=unix` – „`\n`“, Standard unter Unix/Linux/macOS.
- `:fileformat=dos` – „`\r\n`“, typisch für Windows.

Sinnvolle Basiseinstellung im Header:

```
set fileformat=unix
set fileformats=unix,dos
```

Damit:

- speichert VIM standardmäßig im Unix-Format,
- kann aber beim Öffnen auch DOS-Dateien erkennen und korrekt behandeln.

Nur wenn du regelmäßig Dateien *für* Windows-Werkzeuge erzeugst, stellst du das Format gezielt um, z. B.:

- `:set fileformat=dos`
- `:w`

`modeline`: Komfort vs. Sicherheit

`:modeline` erlaubt es, Optionen direkt in Dateien zu hinterlegen, etwa:

```
/* vim: set ts=4 sw=4 et : */
```

oder:

```
# vim: set filetype=python :
```

Das ist praktisch, hat aber zwei Haken:

- Sicherheit: Fremde Dateien können so (in Grenzen) dein Verhalten im Editor beeinflussen.
- Nachvollziehbarkeit: Einstellungen stehen dann *verteilt* in Dateien statt zentral in der `vimrc`.

Darum wählen viele heute bewusst:

```
set nomodeline
```

Wenn du `:modeline` gezielt einsetzen willst, solltest du zusätzlich `:modelines` (Anzahl der eingelesenen Zeilen am Anfang/Ende der Datei) prüfen und ggf. einschränken.

Basisoptionen, die „überall“ sinnvoll sind

In den Header gehören auch Einstellungen, die fast nie schaden und praktisch in jedem Projekt nützlich sind. Beispiele:

```
" Basis-UI
set number            " Zeilennummern
set showcmd           " letzte Eingabe in der Statuszeile an
set ruler             " Cursorposition in der Statuszeile ze

" Editierverhalten
set backspace=indent,eol,start
set autoindent
set smartindent

" Suche
set ignorecase
set smartcase
set hlsearch
set incsearch
```

Kriterien für „Header-tauglich“:

- Option ist in 95 % der Fälle sinnvoll.
- Du willst sie *nicht* pro Projekt umschalten.
- Sie hat keine seiteneffekte, die bestimmte Workflows kaputt machen.

Feinere Projekt- oder Dateityp-spezifische Einstellungen gehören später in „lokale“ Konfigurationsblöcke (z. B. `ftplugin`, `:autocmd`-Gruppen).

Übungen

1. Öffne deine `vimrc` und ergänze am Anfang einen sauberen Header-Block mit:
   - `:set nocompatible`
   - einem klar kommentierten Abschnitt „Basis-Setup“
   - expliziten Einstellungen für `:encoding`, `:fileencoding`, `:fileformats`.

   Ziel: Der Kopf der `vimrc` erklärt auf einen Blick, in welchem „Modus“ VIM laufen soll.
2. Füge eine Zeile mit `:set nomodeline` ein und öffne danach eine Datei, die eine „vim:“-Zeile enthält (z. B. aus einem Fremdprojekt). Prüfe mit `:set` (`:set ts?`, `:set ft?`), ob diese Zeile noch Einfluss hat oder nicht.

3. Lege eine Testdatei mit Windows-Zeilenenden an (z. B. über ein externes Tool oder `:set fileformat=dos` + `:w`). Öffne sie in VIM und beobachte:
   - Wie verhält sich `:fileformat`, wenn deine `vimrc` den Header aus diesem Abschnitt nutzt?
   - Was passiert, wenn du danach explizit wieder `:set fileformat=unix` setzt und speicherst?

   Ziel: Gefühl für das Zusammenspiel von `:fileformat` und `:fileformats` bekommen.

13.2.2. Konfigurationsblöcke

Eine *lesbare* `vimrc` besteht nicht aus einer endlosen Liste einzelner `:set`-Zeilen, sondern aus klar getrennten Konfigurationsblöcken. Ziel ist, dass du auch nach Monaten auf einen Blick erkennst:

- *wo* eine Einstellung hingehört,
- *warum* sie existiert,
- und *welche* Bereiche du gefahrlos umschalten oder löschen kannst.

Ein typisches Layout gliedert sich in:

1. globale Optionen (Basisverhalten),
2. Mappings (Tastenkürzel),
3. Autocommands (automatisches Verhalten),
4. Plugin-Konfigurationen.

Layout mit Kommentaren

Lesbarkeit steht an erster Stelle. Bewährt haben sich klar kommentierte Abschnitte, z. B.:

```
" ==========================
" Basis-Optionen
" ==========================
set nocompatible
set encoding=utf-8
set number
set relativenumber
set ignorecase
set smartcase
```

```
" ==========================
" Tastenkürzel (Mappings)
" ==========================
nnoremap <leader>w :w<CR>
nnoremap <leader>q :q<CR>

" ==========================
" Autocommands
" ==========================
augroup my_autocmds
  autocmd!
  autocmd BufReadPost * normal! zv
augroup END

" ==========================
" Plugins
" ==========================
" Plug 'tpope/vim-fugitive'
" Plug 'junegunn/fzf.vim'
```

Wichtige Punkte:

- Überschriften mit auffälligen Kommentarzeilen erleichtern das spätere Navigieren.
- Innerhalb eines Blocks sollten die Einstellungen thematisch zusammenpassen (z. B. alle Suchoptionen beieinander).
- Kommentiere *warum* du etwas setzt, nicht nur *was* es macht, insbesondere bei ungewöhnlichen Optionen.

Reihenfolge: Optionen → Mappings → Autocommands → Plugins

Konfiguration wird in VIM streng von oben nach unten abgearbeitet. Deshalb ist die Reihenfolge nicht nur kosmetisch, sondern funktional:

1. **Optionen** Zuerst Grundverhalten definieren:
    - Anzeige (`:number`, `:relativenumber`, `:cursorline`)
    - Editierlogik (`:expandtab`, `:shiftwidth`, `:tabstop`)
    - Suche (`:ignorecase`, `:smartcase`, `:hlsearch`)

Mappings und Autocommands sollen auf diesem stabilen Fundament aufbauen.

2. **Mappings** Danach allgemeine Tastenkürzel:
   - Leader-Mappings (`:let mapleader = "\textbackslash"`)
   - Navigation (`:nnoremap <leader>n :bnext<CR>`)
   - Komfortbefehle (`:nnoremap <leader>s :w<CR>`)

   Wichtig: „noremap"-Varianten (`:nnoremap`, `:vnoremap`, `:inoremap`) verwenden, damit sich Mappings nicht gegenseitig unerwartet umbiegen.
3. **Autocommands** Erst hier Verhalten definieren, das auf Ereignisse reagiert:
   - `:BufRead`, `:BufWritePost`, `:InsertEnter`, ...
   - immer in `:augroup`-Blöcken mit `:autocmd!`, um Mehrfachregistrierungen zu vermeiden.
4. **Plugins** Plugin-Manager und Plugin-spezifische Einstellungen:
   - Definition der Plugins (z. B. `plug#begin`, `call dein#begin`).
   - pro Plugin möglichst ein eigener, klar kommentierter Block.

Grundregel: Was *viele* andere Dinge beeinflusst, steht oben (Optionen). Was von vielem *abhängt*, steht unten (Plugins, plugin-spezifische Mappings).

Allgemein vs. projekt- und dateitypspezifisch

Nicht jede Einstellung gehört in die globale `vimrc`. Eine saubere Trennung vermeidet Chaos:

- **Allgemein** (in `vimrc`):
  - Basis-UI (`:number`, `:showcmd`, `:ruler`)
  - allgemeine Such- und Editieroptionen
  - universelle Mappings („funktioniert in jedem Projekt")
- **Dateitypspezifisch**:
  - `:ftplugin` oder `after/ftplugin` für Sprache-spezifische Einstellungen (z. B. `python.vim`, `markdown.vim`).
  - Autocommands der Form `:FileType python setlocal ...` statt globale `:set`.
- **Projektbezogen**:
  - lokale `vimrc`-Varianten (`.vim/local.vim`, `.nvimrc`) per Plugin oder Skript einbinden.
  - nur dort Dinge definieren, die *dieses* Projekt betreffen (z. B. `:makeprg`, Linting-Regeln).

Heuristik: Wenn du eine Option regelmäßig „für dieses eine Projekt“ anders brauchst, gehört sie nicht in den globalen Block, sondern in einen dateityp- oder projektbezogenen Abschnitt.

Übungen

1. Öffne deine bestehende `vimrc` und markiere mit Kommentaren vier Hauptblöcke:
   - Basis-Optionen
   - Mappings
   - Autocommands
   - Plugins

   Verschiebe einzelne Einstellungen so, dass sie in den passenden Block fallen (z. B. alle `:nnoremap`-Zeilen in den Mapping-Abschnitt).
2. Suche in deiner `vimrc` nach Einstellungen, die offensichtlich nur für einen bestimmten Dateityp gedacht sind (z. B. `:set tabstop=4` *nur* für Python). Überlege, ob diese in einen `:FileType`-Autocommand oder eine `ftplugin`-Datei ausgelagert werden sollten.
3. Erstelle im Kopf deiner `vimrc` einen kurzen Kommentarblock mit einer „Legende“, z. B.:

   ```
   " Struktur:
   "   1) Basis-Optionen
   "   2) Mappings
   "   3) Autocommands
   "   4) Plugins
   ```

   Ziel: Deine eigene `vimrc` soll in einem Jahr für dich genauso verständlich sein wie heute – oder besser.

13.2.3. Externe Dateien einbinden

Eine einzelne, immer länger werdende `vimrc` wird schnell unübersichtlich. VIM erlaubt es, Konfiguration auf mehrere Dateien aufzuteilen und bei Bedarf einzulesen. Zentrales Werkzeug dafür ist der Ex-Befehl `:source`.

`:source`: Konfigurationsdateien laden

Mit `:source` kann jede VIM-Skriptdatei eingelesen und ausgeführt werden:

- `:source ~/.vim/mappings.vim`
- `:source ~/.vim/plugins.vim`
- `:source ~/.vim/filetypes.vim`

Typische Anwendungsfälle:

- Aufteilen einer großen `vimrc` in mehrere thematische Dateien (z. B. `options.vim`, `mappings.vim`, `plugins.vim`).
- Projekt- oder kundenbezogene Zusatzkonfigurationen (`proj_xyz.vim`).
- Gemeinsame Konfiguration für mehrere Maschinen (z. B. via Git-Repository), die lokal nur noch per `:source` eingebunden wird.

Beispiel: modulare `vimrc` Im Kopf der `vimrc` können externe Bausteine geladen werden:

```
" ==========================
" Basis-Setup
" ==========================
set nocompatible
set encoding=utf-8

" ==========================
" Externe Konfigbausteine
" ==========================
if filereadable(expand("~/.vim/options.vim"))
  source ~/.vim/options.vim
endif

if filereadable(expand("~/.vim/mappings.vim"))
  source ~/.vim/mappings.vim
endif

if filereadable(expand("~/.vim/plugins.vim"))
  source ~/.vim/plugins.vim
endif
```

Vorteil: Jeder Block kann separat gepflegt, versioniert und bei Bedarf auch auf anderen Systemen wiederverwendet werden.

Eigene Dateien für Mappings, Plugins, Dateiarten

Bewährt haben sich u. a. folgende Aufteilungen:

- `options.vim` Grundoptionen, Anzeige, Verhalten, Suche, Einrückung.
- `mappings.vim` alle globalen `:noremap`-Definitionen, Leader-Mappings, Komfortbefehle.
- `plugins.vim` Plugin-Manager + Plugin-Liste, grundlegende Plugin-Optionen.
- `filetypes.vim` einfache `:autocmd FileType`-Regeln, falls kein eigenes `ftplugin`-Verzeichnis genutzt wird.
- projektspezifische Skripte, z. B. `proj_kunde_A.vim`, die nur bei Bedarf mit `:source` geladen werden.

Wichtig ist eine klare, selbsterklärende Benennung: Schon am Dateinamen sollte erkennbar sein, *worum* es in der Datei geht.

Neovim: `init.vim` und Lua-Konfiguration

Für Neovim gilt das gleiche Prinzip. Typisches Layout:

```
" ~/.config/nvim/init.vim
source ~/.config/nvim/options.vim
source ~/.config/nvim/mappings.vim
source ~/.config/nvim/plugins.vim
```

In Lua-basierter Konfiguration kann analog mit `require` gearbeitet werden (z. B. `lua/options.lua`, `lua/mappings.lua`). Das Konzept bleibt identisch: kleine, klar fokussierte Konfigbausteine.

Vorteile kleiner, wartbarer Konfigbausteine

Das Aufteilen einer großen `vimrc` in mehrere Dateien hat praktische Effekte:

- Änderungen sind lokal begrenzt: ein Fehler in `mappings.vim` zerstört nicht die gesamte Grundkonfiguration.
- Bestimmte Bausteine können temporär deaktiviert werden, indem die zugehörige `:source`-Zeile auskommentiert wird.
- Einzelne Konfigteile (z. B. `plugins.vim`) lassen sich leicht auf anderen Systemen wiederverwenden.
- Versionierung (z. B. mit Git) wird übersichtlicher, weil Diffs pro Themenbereich kleiner bleiben.

Übungen

1. Erstelle im Verzeichnis `~/.vim` (bzw. `~/.config/nvim`) eine Datei `options.vim` mit einigen Basisoptionen, z. B.:

   ```
   " Basis-Optionen
   set number
   set relativenumber
   set ignorecase
   set smartcase
   ```

   Binde sie in deiner `vimrc` über `:source` ein und überprüfe mit `:set number?`, dass die Optionen übernommen wurden.
2. Lagere deine globalen Mappings in eine eigene Datei `mappings.vim` aus. Entferne die Mappings aus der `vimrc` und ersetze sie dort durch eine einzelne Zeile:

   ```
   source ~/.vim/mappings.vim
   ```

   Starte VIM neu und teste, ob alle Tastenkürzel weiterhin funktionieren.
3. Erzeuge eine kleine projektbezogene Konfigurationsdatei, z. B. `proj_demo.vim`, in der du `:setlocal tabstop`, `:shiftwidth` und `:expandtab` definierst. Lade diese Datei nur im entsprechenden Projektverzeichnis per `:source` und beobachte, wie sich das Verhalten von VIM dort gezielt unterscheidet. Ziel: ein Gefühl für modulare, kontextabhängige Konfiguration entwickeln.

13.3. Optionen gezielt konfigurieren

13.3.1. Anzeige und Benutzeroberfläche

Viele VIM-Optionen beeinflussen nicht den Inhalt der Datei, sondern *nur* die Art, wie sie dargestellt wird. Eine sinnvolle Grundeinstellung der Anzeigeoptionen entscheidet darüber, ob man früh Fehler erkennt oder sich in unübersichtlichen Buffern verliert.

Im Folgenden ein praxisnaher Überblick über typische UI-Optionen.

Zeilennummern

Mit `number` und `relativenumber` werden Zeilennummern eingeblendet:

- `:set number` zeigt absolute Zeilennummern an (1, 2, 3, ...).
- `:set relativenumber` zeigt relative Nummern zur Cursorzeile (0, ±1, ±2, ...).
- Kombination beider Optionen: `:set number relativenumber` Die aktuelle Zeile zeigt die absolute Nummer, alle anderen Zeilen relative Abstände. Ideal für Befehle wie `:5j` oder `:3dd`.

Tipp: Wer sich mit relativen Nummern nicht anfreunden kann, beginnt einfach mit `:set number` und schaltet `:set relativenumber` später dazu.

Cursorzeile hervorheben

Die Option `cursorline` hebt die aktuelle Zeile optisch hervor:

- `:set cursorline` aktiviert die Hervorhebung der Cursorzeile.
- `:set nocursorline` deaktiviert sie wieder.

Gerade in breiten Splits oder auf großen Monitoren verhindert `cursorline`, dass der Blick „verloren geht“. Optional kann man auch `cursorcolumn` nutzen, was eine Spalte markiert — in vielen Farbschemata aber schnell überladen wirkt.

Unsichtbare Zeichen sichtbar machen

Mit `list` und `listchars` kontrollierst du, welche „unsichtbaren“ Zeichen sichtbar gemacht werden:

- `:set list` schaltet die Anzeige laut `listchars` ein.
- `:set nolist` schaltet sie wieder aus.

Typische Konfiguration:

- `:set list`
- `:set listchars=tab:>-,trail:· ,extends:>,precedes:<,nbsp`

Damit werden u. a.:

- Tabs als `>-` angezeigt,
- nachgestellte Leerzeichen als Punkt (·),
- nicht umbrechbare Leerzeichen (`nbsp`) als `_`.

Das hilft enorm dabei, *unnötige* oder *gefährliche* Whitespace zu sehen (z. B. Tabs in YAML, Spaces am Zeilenende, harte Umbrüche).

Statuszeile und Kommandorückmeldung

Die Statuszeile und einige kleine Helferoptionen geben Rückmeldung über „wo bin ich gerade?“ und „was tippe ich?“:

- `:set ruler` zeigt Position (Zeile, Spalte, Prozent) in der Statuszeile.
- `:set showcmd` zeigt unvollständige Befehlsfolgen in der unteren Zeile an, z. B. `2d` während der Eingabe.
- `:set showmode` zeigt Modusmeldungen wie „– INSERT –“ unten an.

Bei vielen modernen Statusline-Plugins (Neovim, Lua-Statuszeilen) wird `showmode` bewusst deaktiviert, weil der Modus bereits in der Statuszeile steht. In einer klassischen VIM-Umgebung ist `showmode` dagegen oft hilfreich.

Farben und Farbschema

Die grundsätzliche Farbgebung wird über `:colorscheme` und `:set background` gesteuert:

- `:colorscheme desert` wählt das Schema `desert` (nur Beispiel — abhängig von der Installation stehen unterschiedliche Schemata zur Verfügung).
- `:set background=dark` optimiert die Farbauswahl für dunkle Hintergründe (Terminal mit dunklem Theme).
- `:set background=light` optimiert für helle Hintergründe.

Praxisregel: Zuerst das Terminal- oder GUI-Theme festlegen, dann `:set background` passend setzen, erst danach ein `:colorscheme` wählen. Viele Schemata sehen sonst „verwaschen“ aus, weil sie für den jeweils anderen Hintergrund gedacht sind.

Beispiel: Minimaler UI-Block für die `vimrc`

Ein kleiner, alltagstauglicher Block könnte so aussehen:

```
" Anzeige & UI
set number
set relativenumber
set cursorline

set list
set listchars=tab:>-,trail:· ,extends:>,precedes:<,nbsp:_

set ruler
set showcmd
```

```
set showmode

set background=dark
colorscheme default
```

Dies ist ein solider Startpunkt, den man schrittweise anpassen kann.

Übungen

1. Aktiviere nacheinander:
   - `:set number`
   - `:set relativenumber`

   und beobachte, wie sich die Anzeige der Zeilennummern ändert. Entscheide bewusst, ob du nur absolute oder absolute plus relative Nummern verwenden möchtest, und trage die gewählte Kombination in deine `vimrc` ein.
2. Aktiviere `:set list` und konfiguriere `listchars` wie oben gezeigt. Erzeuge eine Testdatei mit:
   - Tabs am Zeilenanfang,
   - nachgestellten Leerzeichen,
   - Leerzeilen, die wirklich leer sein sollen.

   Passe `listchars` so an, dass du diese Fälle sofort visuell erkennst.
3. Spiele mit `:set background=dark` und `:set background=light` und wechsle dazwischen, während ein Farbschema aktiv ist. Beobachte, welche Kombination für dein Terminal oder deine GUI am besten lesbar ist, und trage deine bevorzugte Einstellung in die `vimrc` ein.
4. Deaktiviere testweise `:set showcmd` und `:set showmode` und arbeite einige Minuten in VIM (z. B. mit Operatoren wie `:3dw`, `:2dd`). Schalte die Optionen dann wieder ein und entscheide, ob dir die zusätzliche Rückmeldung bei der Befehlseingabe hilft oder eher stört.

13.3.2. Einrückung und Tabs

Einrückung entscheidet in vielen Sprachen über Lesbarkeit, in einigen (Fallstricke: Python, YAML, Makefiles) sogar über die *Semantik*. VIM trennt hier drei Dinge:

- wie viele Zeichen ein sichtbares „Tab“ *repräsentiert*,
- wie weit strukturierende Befehle (z. B. `:>>`) einrücken,

- ob echte Tab-Zeichen oder Leerzeichen geschrieben werden.

Die wichtigsten Optionen dafür sind `tabstop`, `shiftwidth`, `softtabstop`, `expandtab`, `autoindent` und `smartindent`.

Grundparameter für Tabs und Einrückung

Die folgenden Optionen wirken zusammen:

- `tabstop` (`ts`) Steuert, wie viele Spalten ein *Tab-Zeichen* im Text *anzeigeseitig* einnimmt.
  - `:set tabstop=4` — Tabs erscheinen optisch als 4 Spalten.
  - Die Datei kann trotzdem echte Tabs *oder* Leerzeichen enthalten.
- `shiftwidth` (`sw`) Bestimmt, wie viele Spalten strukturierende Befehle einrücken:
  - `:set shiftwidth=4` — `:>>`, `:<<`, `:==` etc. arbeiten in 4er-Schritten.
  - Üblich: `shiftwidth` = `tabstop`.
- `softtabstop` (`sts`) Steuert, wie sich **<Tab>** und **<BS>** im Insert-Mode verhalten:
  - `:set softtabstop=4` — **<Tab>** fügt (logisch) 4 Spalten hinzu, **<BS>** löscht sie wieder als Einheit.
  - `:set softtabstop=0` — orientiert sich an `tabstop`.
- `expandtab` (`et`) Entscheidet, ob **<Tab>** echte Tab-Zeichen *oder* Leerzeichen erzeugt:
  - `:set expandtab` — Tabs werden beim Schreiben in Leerzeichen umgesetzt.
  - `:set noexpandtab` — echte `\t`-Zeichen bleiben erhalten.

Typische Kombination:

```
" 4-Leerzeichen-Einrückung (z.B. für viele Codebasen)
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab
```

Damit verhält sich **<Tab>** wie ein Block aus 4 Leerzeichen, und Strukturbefehle (`:>>`) arbeiten im selben Raster.

Automatische Einrückung

Neben der reinen Breite kann VIM Einrückungen automatisch fortführen:

- `autoindent` (`ai`) Kopiert die Einrückung der vorherigen Zeile:
  - `:set autoindent` – neue Zeile beginnt mit der gleichen Einrückung wie die vorherige.
- `smartindent` Ergänzt `autoindent` um einfache C-ähnliche Regeln:
  - `:set smartindent` – nach { wird tiefer eingerückt, nach } wieder zurück.
  - Funktioniert gut für C, C++, einfache Sprachen mit Blockklammern, weniger für exotische Syntax.

Für viele Setups ist zusätzlich sinnvoll:

- `" Einrückung vom Filetype abhängig machen (falls unterstützt)`
- `:filetype plugin indent on`

Damit verwenden viele Sprachen ihre eigenen Einrückungsregeln (z. B. spezielle Regeln für Python, XML, LaTeX, ...).

Typische Setups: Code vs. Konfiguration

Je nach Dateityp haben sich unterschiedliche Konventionen etabliert:

Allgemeiner Code (z. B. C, Python, Lua)

```
" 4-Spalten-Einrückung mit Spaces
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab
set autoindent
set smartindent
```

- Lesbare 4er-Einrückung,
- keine echten Tab-Zeichen (kompatibel mit verschiedensten Tools),
- automatische Fortsetzung der Einrückung in Code-Blöcken.

Konfigurationsdateien (YAML, TOML, viele INI-Varianten)

```
" 2-Spalten-Einrückung für kompakte Konfigurationen
set tabstop=2
set shiftwidth=2
set softtabstop=2
set expandtab
set autoindent
```

- YAML und Co. sind extrem empfindlich bei Tabs,
- `:set expandtab` ist hier praktisch Pflicht,
- 2 Leerzeichen machen verschachtelte Strukturen kompakt und leserlich.

Tab-pflichtige Dateien (z. B. Makefiles)

```
" Makefiles: echte Tabs notwendig
set noexpandtab
set tabstop=8
set shiftwidth=8
set softtabstop=0
```

- Make erwartet *echte Tabs* vor Rezeptzeilen,
- `:set noexpandtab` verhindert, dass VIM sie in Spaces umwandelt.

Wichtige Praxisregel: Nie innerhalb eines Projekts zwischen „Tabs“ und „Spaces“ mischen, ohne dass alle Beteiligten die gleichen Einstellungen verwenden. `:set list` und `listchars` (siehe vorheriger Abschnitt) helfen, unerwünschte Tabs oder Spaces sichtbar zu machen.

Übungen

1. Erzeuge eine Testdatei `indent-test.txt` und setze:
   - `:set tabstop=4 shiftwidth=4 softtabstop=4 expandtab`

   Schreibe mehrere Code-ähnliche Zeilen und nutze:
   - **<Tab>** im Insert-Mode,
   - `:>>` und `:<<` im Normal-Mode.

   Aktiviere anschließend `:set list` und prüfe, ob nur Leerzeichen, aber keine echten Tabs verwendet wurden.
2. Ändere die Einstellungen auf:

- `:set noexpandtab tabstop=4 shiftwidth=4 softtabstop=4`

Wiederhole die Einrückungstests. Vergleiche mit `::set list`, wo jetzt reale Tab-Zeichen entstehen, und beobachte die Unterschiede im Verhalten von **<BS>** im Insert-Mode.

3. Erzeuge eine Beispiel-`Makefile` mit einer einfachen Regel (z. B. `all:` und einer `echo`-Zeile darunter). Stelle sicher, dass `:set noexpandtab` aktiv ist und die Rezeptzeile wirklich mit einem Tab beginnt. Prüfe mit `:set list`, dass dort ein echtes Tab-Zeichen steht, und führe `::!make` aus, um zu sehen, dass Make akzeptiert.
4. Baue in deiner `vimrc` einen kleinen Block, der anhand des Dateityps unterschiedliche Einrückung setzt, z. B.:

```
augroup indent_by_filetype
  autocmd!
  autocmd FileType yaml,yml,toml
        \ setlocal tabstop=2 shiftwidth=2 softtabstop=2
  autocmd FileType make
        \ setlocal noexpandtab tabstop=8 shiftwidth=8 s
augroup END
```

Öffne anschließend eine YAML-Datei und ein Makefile und kontrolliere mit `::setlocal` und `:set list`, dass jeweils die erwarteten Einrückungsregeln aktiv sind.

13.3.3. Suche und Ersetzen

Die Optionen für Suche und Ersetzen legen fest, wie VIM auf „finden“ und „ersetzen“ reagiert – unabhängig davon, ob du mit **/**, **?** oder mit `:substitute` arbeitest. In der `vimrc` werden damit vor allem drei Dinge konfiguriert:

- Groß-/Kleinschreibung der Suche,
- visuelles Feedback (Highlighting, inkrementelle Suche),
- die Semantik von regulären Ausdrücken („magic“ vs. „nomagic“).

Alle diese Optionen wirken gleichzeitig auf:

- vorwärts-Suche (**/**),
- rückwärts-Suche (**?**),
- Ersetzungen mit `:s/\dots /\dots /` und `:substitute`.

Groß-/Kleinschreibung: `ignorecase` und `smartcase`

Zwei Optionen steuern Case-Sensitivity global:

- `ignorecase` (`ic`) `:set ignorecase` sorgt dafür, dass Suchmuster standardmäßig *nicht* zwischen Groß- und Kleinschreibung unterscheiden.
  - [/] `foo` findet `foo`, `Foo`, `FOO` usw.
  - `:\%s/foo/bar/g` ersetzt alle Varianten, solange das Muster keine Großbuchstaben enthält.
- `smartcase` `:set smartcase` macht `ignorecase` „intelligent“: enthält das Suchmuster *mindestens einen Großbuchstaben*, wird die Suche automatisch case-sensitiv.
  - [/] `foo` (alles klein) ⟶ case-insensitive.
  - [/] `Foo` (mit Großbuchstabe) ⟶ case-sensitiv, findet nur `Foo`, nicht `foo`.

Ein typischer Block in der `vimrc`:

```
" Intelligente Groß-/Kleinschreibung bei der Suche
set ignorecase
set smartcase
```

Damit musst du nicht ständig zwischen „sensitiv“ und „insensitiv“ umschalten: Klein geschrieben = locker, gemischt geschrieben = streng.

Visuelles Feedback: `hlsearch` und `incsearch`

Für das unmittelbare Feedback beim Suchen sind zwei Optionen wichtig:

- `hlsearch` `:set hlsearch` markiert *alle* Vorkommen des letzten Suchmusters im aktuellen Buffer. Typisches Verhalten:
  - Muster mit [/] oder [?] suchen,
  - alle Treffer bleiben hervorgehoben,
  - mit `:nohlsearch` bzw. `:noh` die Markierung temporär wieder abschalten.
- `incsearch` `:set incsearch` schaltet „inkrementelle Suche“ ein: während du das Suchmuster tippst, springt der Cursor schon zu potenziellen Treffern und zeigt dir live, was passieren würde.
  - [/] tippen, dann Buchstabe für Buchstabe schreiben,
  - der Cursor wandert mit und zeigt „live“, was gefunden wird.

Empfohlener Block in der `vimrc`:

```
" Such-Highlighting und inkrementelle Suche
set hlsearch
set incsearch
```

Wer die „Dauerbeleuchtung“ nicht mag, kann in der Praxis z.B. eine Tastenbelegung ergänzen:

```
" Such-Highlighting per Taste löschen
nnoremap <leader>h :nohlsearch<CR>
```

`'magic'` vs. `'nomagic'`

VIM kennt historisch zwei Regelsätze für reguläre Ausdrücke:

- `nomagic` (Voreinstellung in klassischem VIM) Nur wenige Zeichen sind „magisch“ (z.B. `.`, `^`, `$`), viele Metazeichen müssen mit Backslash eingeleitet werden (z.B. `\+`, `\?`, `\|`).
- `magic` Mehr Zeichen sind standardmäßig „besonders“; dadurch werden Muster kürzer, aber auch „regex-lastiger“.

Statt die globale Option hart umzukrempeln, arbeitet man in der Praxis häufig mit *Präfixen* im Suchmuster:

- `\v` — „very magic“: fast alle Regex-Zeichen sind magisch Beispiel: `/ \vfoo.+bar/`
- `\V` — „very nomagic“: fast alles ist Literal Beispiel: `/ \Vfoo.+bar/` sucht *genau* `foo.+bar`.
- `\m` / `\M` — „magic“ bzw. „nomagic“ schalten auf die entsprechenden Modi um.

Typische Empfehlung in der `vimrc`:

```
" Standardmäßig etwas "regex-freundlicher"
set magic
```

und dann bei komplizierten Mustern bewusst mit `\v` oder `\V` arbeiten:

```
" Beispiel: Wort 'foo' gefolgt von beliebigem Text bis 'bar
/\v\<foo\>.*bar

" Beispiel: Literalzeichen-Kette 'foo.*bar' suchen
/\Vfoo.*bar
```

Wichtig: Diese Regeln gelten einheitlich für:

- `/` und `?` (normale Suche),
- `:s/\dots /\dots /` (Substitute),
- viele Plugin-Kommandos, die auf VIMs Regex-Engine aufsetzen.

Übungen

1. Aktiviere in einer Test-`vimrc` zunächst:

   ```
   set ignorecase
   set smartcase
   set hlsearch
   set incsearch
   ```

   Erzeuge eine Datei mit den Wörtern `foo`, `Foo`, `FOO` in verschiedenen Zeilen. Probiere nacheinander:
   - `/`foo, `n`, `N`
   - `/`Foo, `n`, `N`

   Beobachte:
   - wann die Suche case-insensitiv,
   - und wann sie case-sensitiv wird.
2. Füge in deiner `vimrc` folgenden Block ein:

   ```
   set hlsearch
   set incsearch
   nnoremap <leader>h :nohlsearch<CR>
   ```

   Suche in einer größeren Datei nach einem Begriff deiner Wahl und nutze danach <leader>`h`, um das Highlighting gezielt auszuschalten. Ziel: „Such-Spotlight“ bewusst ein- und ausblenden.
3. Teste den Unterschied zwischen „magic“ und „very magic“:
   a) Schreibe einige Zeilen mit Mustern wie `foo123bar`.
   b) Suche zuerst mit einem *klassischen* Muster:

      ```
      /foo[0-9]\+bar
      ```

   c) Suche dann mit `\v`:

      ```
      /\vfoo\d+bar
      ```

   Vergleiche, welches Muster du intuitiver findest, und überlege, ob `\v` als Standardstil für komplexe Regexe in deinem Workflow sinnvoll ist.

4. Ergänze in deiner `vimrc` einen Kommentarblock, der in eigenen Worten festhält, wie `ignorecase`, `smartcase`, `hlsearch`, `incsearch` und `magic` zusammenspielen. Ziel: deine persönliche „Mini-Doku“ direkt neben den Optionen – damit du in ein paar Monaten noch nachvollziehen kannst, *warum* du diese Kombination gewählt hast.

13.3.4. Backup, Swap und Undo

VIM schützt Inhalte auf mehreren Ebenen gegen Datenverlust. In der `vimrc` lassen sich diese Mechanismen gezielt steuern:

- „klassische“ Backups (`backup`, `writebackup`),
- Swap-Dateien (`swapfile`),
- persistente Undo-Historie (`undofile`).

Ziel ist ein Setup, das Abstürze, Stromausfälle und Fehlbedienungen möglichst gut abfedert, ohne das Dateisystem mit temporären Dateien zu zumüllen.

Backup-Dateien: `backup`, `writebackup`, `backupdir`

VIM kann vor dem Überschreiben einer Datei eine Kopie anlegen:

- `backup` `:set backup` sorgt dafür, dass beim Speichern eine Backup-Datei angelegt wird (typisch: -Suffix).
- `writebackup` `:set writebackup` erstellt nur während des Schreibvorgangs ein Backup – danach wird dieses wieder entfernt. Dadurch sinkt die Gefahr von Dateikorruptionen, ohne dauerhaft viele Backup-Dateien zu hinterlassen.

Mit `backupdir` lässt sich der Speicherort zentral steuern:

```
" Backup-Dateien zentral in einem Verzeichnis sammeln
set backup
set writebackup
set backupdir^=~/.vim/backup//
```

Wichtig dabei:

- Das Verzeichnis `/~.vim/backup` muss existieren.
- Der doppelte Slash `//` am Ende sorgt dafür, dass VIM bei Namenskollisionen Unterverzeichnisse bildet und Pfadanteile kodiert, statt Dateien zu überschreiben.

Typische Minimal-Variante (wenn man nur „Crash-Schutz“ will):

```
set writebackup
set nobackup
```

Hier wird eine Schutzkopie nur während des Schreibens angelegt, danach aber wieder entfernt.

Swap-Dateien: `swapfile`, `directory`

Swap-Dateien (`.swp`) dienen dazu, den aktuellen Bearbeitungsstand auch bei hartem Absturz wiederherstellen zu können. Sie werden beim Öffnen einer Datei erzeugt und beim sauberen Beenden wieder gelöscht.

Zentrale Optionen:

- `swapfile` `:set swapfile` aktiviert Swap-Dateien (Standard). Mit `:set noswapfile` lässt sich dies global abschalten – was allerdings bei Stromausfällen ein Risiko bedeutet.
- `directory` legt fest, wo Swap-Dateien abgelegt werden. Beispiel:

  ```
  " Swap-Dateien nicht im Projektverzeichnis, sondern zentral
  set directory^=~/.vim/swap//
  ```

Empfohlenes Muster:

1. Verzeichnis anlegen, z. B. `/~.vim/swap`.
2. `directory` in der `vimrc` darauf zeigen lassen.

Damit bleiben Projektverzeichnisse sauber, und dennoch können Crash-Wiederherstellungen über `:recover` genutzt werden.

Persistente Undo-Historie: `undofile`, `undodir`

Standardmäßig ist Undo nur so lange verfügbar, wie der Buffer im Speicher ist. Nach dem Schließen der Datei ist die Undo-Historie verloren.

Mit *persistent undo* lässt sich das ändern:

```
" Persistentes Undo für alle Dateien
set undofile
set undodir^=~/.vim/undo//
```

Wirkung:

- VIM legt beim Bearbeiten zu jeder Datei eine Undo-Datei an.

- Beim erneuten Öffnen der Datei steht die komplette Undo-Historie wieder zur Verfügung.
- Aktionen wie **u** (Undo) und **Ctrl**-**r** (Redo) funktionieren auch über Sitzungen hinweg.

Voraussetzungen:

- Das Verzeichnis `/~.vim/undo` muss existieren.
- `undofile` darf nicht lokal ausgeschaltet werden (z. B. durch Plugins).

Persistentes Undo ist besonders wertvoll bei:

- großen Projekten mit langlaufenden Refactorings,
- Konfigurationsdateien, die man selten, aber riskant anfasst,
- „Fehlersuchen“, bei denen man Schritt für Schritt Änderungsketten zurückgehen möchte.

Typische Kombination in der `vimrc`

Ein praxistauglicher Block (Unix/Neovim/VIM):

```
" Verzeichnisse für Backup, Swap und Undo (vorher anlegen!)
set backup
set writebackup
set swapfile
set undofile

set backupdir^=~/.vim/backup//
set directory^=~/.vim/swap//
set undodir^=~/.vim/undo//
```

Optional können bestimmte Pfade von Backups/Swaps ausgenommen werden (z. B. `/tmp`, Remote-Mounts), typischerweise über autocommands oder spezifische `:setlocal`-Blöcke.

Übungen

1. Lege die Verzeichnisse `/~.vim/backup`, `/~.vim/swap` und `/~.vim/undo` an und füge den obigen Konfigurationsblock in deine `vimrc` ein. Starte VIM neu und beobachte beim Bearbeiten einer Testdatei, welche Dateien in den drei Verzeichnissen angelegt werden.
2. Erzeuge absichtlich einen „unsauberen“ Editor-Abbruch:

   a) Datei öffnen, Inhalt ändern.
   b) VIM *nicht* regulär beenden (z. B. Terminal-Fenster schließen).
   c) Datei erneut mit VIM öffnen.

   Beobachte die Meldung zur bestehenden Swap-Datei und führe `:recover` aus. Ziel: den Wiederherstellungs-Workflow einmal bewusst erlebt haben.
3. Aktiviere `undofile` und bearbeite eine Datei über mehrere Sitzungen hinweg (z. B. an zwei aufeinanderfolgenden Tagen). Nutze in der zweiten Sitzung **u** und **Ctrl**-**r**, um *über den Sitzungswechsel hinaus* Schritte rückgängig zu machen. Prüfe mit `:ls` und im `undodir`, welche Undo-Dateien erzeugt werden.
4. Erstelle in deiner `vimrc` einen eigenen Kommentarkomplex („Backup/Swap/Undo-Policy“), in dem du kurz festhältst:
   - ob du dauerhafte Backups willst (`backup`) oder nur temporäre (`writebackup`),
   - wo Swap-Dateien liegen sollen,
   - ob persistentes Undo für alle Dateien oder nur für bestimmte Verzeichnisse gelten soll.

   Ziel: eine klar dokumentierte Entscheidung, die du später leicht anpassen kannst.

13.4. Mappings und Abkürzungen

13.4.1. Grundlagen von Mappings

Mappings sind das zentrale Werkzeug, um VIM an die eigenen Arbeitsweisen anzupassen. Statt jeden Befehl mühsam neu zu tippen, können Tastenfolgen „umgebogen“ werden: Eine Taste (oder Kombination) löst dann eine längere Kommando-Sequenz aus.

Formal besteht ein Mapping immer aus:

- einem *linken Teil* („lhs“ – left hand side): die Tastenfolge, die du drückst, z. B. **,** **s** oder <leader>f.
- einem *rechten Teil* („rhs“ – right hand side): die Aktion, die ausgeführt wird, z. B. `:write` oder `:nohlsearch`.

Definiert werden Mappings über Ex-Kommandos wie `:nmap`, `:nnoremap`, `:imap` usw. – und *nicht* durch das normale Tippen im Text.

`map` vs. `noremap`: rekursiv oder nicht

Der wichtigste Unterschied bei Mappings ist: „rekursiv“ vs. „nicht rekursiv“:

- `map`, `nmap`, `vmap`, `imap`, ...sind *rekursiv*. Der rechte Teil kann selbst wieder andere Mappings auslösen.
- `noremap`, `nnoremap`, `vnoremap`, `inoremap`, ...sind *nicht rekursiv*. Der rechte Teil wird so ausgeführt, als gäbe es keine weiteren Mappings.

Typische Schreibweise (Normal-Mode, nicht rekursiv):

```
nnoremap <leader>w :write<CR>
```

Hier passiert beim Drücken von <leader>w genau das, was rechts steht – unabhängig davon, ob [:] oder [Enter] vielleicht ihrerseits gemappt sind.

Gefährlich wird es bei rekursiven Mappings, zum Beispiel:

```
nmap j gj
nmap gj j
```

Solche Konstruktionen können sich gegenseitig aufrufen oder unerwartete Kettenreaktionen auslösen. Deshalb gilt als Faustregel:

> **Eigenes Mapping: immer die `*noremap`-Variante.**

Modusspezifische Varianten

Mappings gelten immer nur für bestimmte Modi. Die wichtigsten Unterkommandos:

- **Normal-Mode**:
  - `nmap` / `nnoremap` – nur im Normal-Mode
- **Insert-Mode**:
  - `imap` / `inoremap` – nur im Insert-Mode
- **Visual-Mode**:
  - `vmap` / `vnoremap` – nur im Visual-Mode
  - `xmap` / `xnoremap` – „visual“ im engeren Sinne (inkl. Block-Visual)
- **Operator-Pending-Mode**:
  - `omap` / `onoremap` – für Kombinationen wie [d] gefolgt von einem Textobjekt oder einer Bewegung

Beispiele:

```
" Normal-Mode: Suchhighlight entfernen
nnoremap <leader>n :nohlsearch<CR>

" Insert-Mode: 'jk' als ESC-Ersatz
inoremap jk <Esc>

" Visual-Mode: Auswahl nach rechts einrücken
vnoremap > >gv
```

Man erkennt das Muster:

- erster Buchstabe = Modus (`n`, `i`, `v`, `x`, `o`),
- `noremap` = nicht rekursiv.

Best Practices für Mappings

Einige einfache Regeln verhindern Chaos in der eigenen `vimrc`:

- **Immer `*noremap` verwenden** Eigene Mappings praktisch nie rekursiv anlegen. Rekursion ist nur in sehr speziellen Fällen sinnvoll.
- **Modus bewusst wählen** Keine `map` ohne Präfix (sie wirkt in mehreren Modi gleichzeitig). Stattdessen: `nnoremap`, `inoremap`, `vnoremap` etc. verwenden.
- **Prefix-Tasten nutzen** Eindeutige Einstiegstasten wie <leader> oder `<localleader>` vermeiden Kollisionen mit eingebauten Shortcuts:

  ```
  let mapleader = " "
  nnoremap <leader>f :Files<CR>
  ```

- **Basisbewegungen nicht überschreiben** Tasten wie h, j, k, l, w, b möglichst unverändert lassen – sonst wird jede fremde VIM-Instanz zur Hürde.
- **Gruppieren und kommentieren** Mappings in der `vimrc` thematisch bündeln („Navigation“, „Fenster“, „Projekt“) und kurz kommentieren, wozu sie dienen.
- **Diagnose-Kommandos kennen** `:map`, `:nmap`, `:imap`, `:vmap` zeigen an, welche Mappings aktuell aktiv sind. Mit `:verbose nmap <lhs>` wird zusätzlich angezeigt, *wo* ein Mapping definiert wurde.

Übungen

1. Lege in deiner `vimrc` ein einfaches Normal-Mode-Mapping an, das <leader>s auf `:write` legt:

   ```
   nnoremap <leader>s :write<CR>
   ```

   Starte VIM neu, ändere eine Testdatei und speichere sie mit <leader>s. Kontrolliere das Mapping mit `:nmap <leader>s`.
2. Erzeuge bewusst ein rekursives Mapping und beobachte den Effekt:

   ```
   nmap j gj
   nmap gj j
   ```

   Navigiere mit **j** und **g** **j** durch eine Datei und achte darauf, wie sich der Cursor verhält. Stelle anschließend auf nicht rekursiv um:

   ```
   nnoremap j gj
   nnoremap k gk
   ```

   und vergleiche das Verhalten.
3. Lege ein Insert-Mode-Mapping an, das **j** **j** in **Esc** umwandelt, und teste es in einer Testdatei. Prüfe mit `:imap jj`, ob das Mapping korrekt registriert wurde. Deaktiviere es temporär mit:

   ```
   iunmap jj
   ```

   und überprüfe den Unterschied.
4. Nutze `:verbose nmap <leader>s` (oder ein anderes Mapping deiner Wahl), um herauszufinden, in welcher Datei und Zeile das Mapping definiert ist. Ziel: Sicher mit Diagnose-Kommandos umgehen, wenn sich zwei Mappings gegenseitig überlagern.

13.4.2. Leader und LocalLeader

Um eigene Shortcuts sauber von VIMs Standardtasten zu trennen, bietet VIM zwei spezielle „Prefix-Tasten“:

- `mapleader` (<leader>) für *globale* Mappings,
- `maplocalleader` (`<localleader>`) für *dateityp- oder projektbezogene* Mappings.

Die Idee: Statt normale Tasten wie **f**, **t** oder **q** zu überladen, definiert man einen klaren Einstieg wie **<leader>** und hängt daran seine eigenen Kommandos auf.

`mapleader` definieren

Standardmäßig ist `mapleader` das Backslash-Zeichen \. Viele Nutzer ändern es auf eine leichter erreichbare Taste, z. B. die Leertaste oder das Komma.

Typische Variante (in der `vimrc`, weit oben):

```
" Leader auf Leertaste legen
let mapleader = " "
```

Wichtig:

- **Immer vor allen Leader-Mappings setzen**. Alles, was <leader> benutzt, liest diesen Wert ein.
- Nur ein einzelnes Zeichen verwenden (oder eine einfache Schlüsselnotation wie "
  ").

Beispiele für globale Leader-Mappings:

```
" Speichern
nnoremap <leader>w :write<CR>

" Suchhighlight löschen
nnoremap <leader>n :nohlsearch<CR>

" Schnell zum nächsten Buffer
nnoremap <leader>b :bnext<CR>
```

Im Kopf kann man sich **<leader>** als „mein persönlicher Shortcut-Namensraum“ merken.

`maplocalleader` für lokale Mappings

`maplocalleader` funktioniert genauso wie `mapleader`, gilt aber primär für *buffer-lokale* oder *dateitypspezifische* Mappings – also typischerweise in `ftplugin`-Dateien oder projektbezogenen Konfigurationsschnipseln.

Definition (ebenfalls in der `vimrc`):

```
let maplocalleader = ","
```

Ein „lokales“ Mapping in einem Datei-Typ-Plugin, z. B. `ftplugin/python.vim`:

```
" In Python-Dateien: aktuellen Buffer formatieren
nnoremap <buffer> <localleader>f :Black<CR>

" Tests im aktuellen File ausführen
nnoremap <buffer> <localleader>t :!pytest %<CR>
```

So bleiben globale Leader-Mappings *überall* gleich, während `<localleader>` pro Dateityp unterschiedliche Funktionen ansteuern kann.

Konventionen für eigene Shortcuts

Ein paar bewährte Regeln, damit der eigene Shortcut-Dschungel nicht ausufert:

- **Leader für „globale“ Aktionen** Dinge, die in *allen* Dateien sinnvoll sind, an **<leader>** hängen, z. B. Speichern, Fensterverwaltung, Projekt-Suche.
- **LocalLeader für Spezialfälle** Dateityp- oder projektbezogene Kommandos unter `<localleader>` halten (Tests, Linter, Formatter einer bestimmten Sprache).
- **Keine Standardtasten zerstören** Basisbewegungen und „muscle memory“ von VIM unangetastet lassen (**h**, **j**, **k**, **l**, **w**, **b**, **y** **y**, **d** **d**, ...).
- **Leader früh definieren** `let mapleader = ...` und `let maplocalleader = ...` ganz am Anfang der `vimrc` setzen, vor Plugin- und Mapping-Blöcken.
- **Sinnvolle Gruppen bilden** Beispiel: **<leader>** **f**... = „find“ (FZF, Grep), **<leader>** **g**... = Git-Kommandos, **<leader>** **w**... = Window/Fenster-Operationen.

Übungen

1. Setze in deiner `vimrc`:

   ```
   let mapleader = " "
   ```

   und ergänze zwei einfache Mappings:

```
nnoremap <leader>w :write<CR>
nnoremap <leader>q :quit<CR>
```

Starte VIM neu und teste **<leader>** **w** sowie **<leader>** **q** in einer Testdatei. Prüfe mit `:echo mapleader`, ob VIM die Einstellung korrekt übernommen hat.

2. Definiere zusätzlich:

```
let maplocalleader = ","
```

Erstelle eine Datei `ftplugin/markdown.vim` (Pfad gemäß deiner VIM-Installation) mit folgendem Inhalt:

```
nnoremap <buffer> <localleader>p :!pandoc % -
o out.pdf<CR>
```

Öffne eine `.md`-Datei, ändere etwas und teste `<localleader>` **p**. Ziel: Spüren, dass dieses Mapping *nur* in Markdown-Dateien aktiv ist.

3. Führe `:nmap <leader>` und `:nmap <localleader>` aus und inspiziere, welche Mappings bereits an deinen Leader und LocalLeader gebunden sind. Überlege, ob du für bestimmte Gruppen (z. B. Git, Tests, Navigation) konsistente Präfixe eingeführt hast oder ob du deine Belegung noch aufräumen möchtest.

13.4.3. Praxisbeispiele für Mappings

Sind die Grundkonzepte von Mappings klar, stellt sich sofort die Frage: „Was lohnt sich in der Praxis wirklich zu belegen?“ Dieser Abschnitt zeigt typische Muster, die den Alltag spürbar beschleunigen.

Die Beispiele gehen davon aus, dass `mapleader` bereits gesetzt ist (z. B. auf die Leertaste) und nutzen **<leader>** als Einstieg für eigene Kommandos.

Schnelles Speichern, Beenden und Neu-Laden

Statt ständig `:w`, `:q` und `:e` zu tippen, sind kompakte Leader-Mappings praktisch:

```
" Speichern, Beenden, Neu-Laden
nnoremap <leader>w :write<CR>
nnoremap <leader>q :quit<CR>
```

```
nnoremap <leader>Q :qall!<CR>
nnoremap <leader>e :edit<CR>
```

Typisches Muster:

- **<leader>** **w** – aktuellen Buffer speichern,
- **<leader>** **q** – aktuelles Fenster schließen,
- **<leader>** **Q** – alles schließen (`:qall!`),
- **<leader>** **e** – Datei neu einlesen (z. B. nach externen Änderungen).

Build, Quickfix und Suche

Klassischer Workflow: „Speichern → Bauen → Fehler durchgehen“. Statt alles einzeln zu tippen:

```
" Build-Workflow
nnoremap <leader>m :write<CR>:make<CR>
nnoremap <leader>c :copen<CR>
nnoremap <leader>n :cnext<CR>
nnoremap <leader>p :cprev<CR>

" Projektweite Suche (Beispiel mit :vimgrep)
nnoremap <leader>/ :vimgrep /TODO/ **/*.c<CR>:copen<CR>
```

Damit wird **<leader>** **m** zur „Build-Taste“: erst speichern, dann `:make` starten, danach mit **<leader>** **n** / **<leader>** **p** durch die Fehlereinträge springen.

Fenster-, Buffer- und Tab-Steuerung

Grundidee: Häufige Fenster- und Bufferaktionen auf einen konsistenten Leader-Namensraum legen.

Fenster-Navigation:

```
" Zwischen Splits wechseln
nnoremap <leader>h <C-w>h
nnoremap <leader>j <C-w>j
nnoremap <leader>k <C-w>k
nnoremap <leader>l <C-w>l
```

```
" Fenstergrößen schnell anpassen
nnoremap <leader>+ <C-w>+
nnoremap <leader>- <C-w>-
nnoremap <leader>= <C-w>=
```

Buffer-Wechsel:

```
" Buffer-Navigation
nnoremap <leader>bn :bnext<CR>
nnoremap <leader>bp :bprevious<CR>
nnoremap <leader>bb :buffer #<CR>
```

Tab-Steuerung:

```
" Tabs
nnoremap <leader>tn :tabnew<CR>
nnoremap <leader>to :tabonly<CR>
nnoremap <leader>tl :tabnext<CR>
nnoremap <leader>th :tabprevious<CR>
```

So bleiben die Standardbefehle (`:bnext`, `:tabnext`, `:tabnew`) erreichbar, aber die häufigsten Varianten hängen bequem an **<leader>**.

Kleine Editier-Helfer

Viele „Mini-Aktionen“ im Alltag lassen sich in ein, zwei Tasten packen:

```
" Nach dem Speichern automatisch Whitespace bereinigen
nnoremap <leader>s :write<CR>:%s/\s\+$//e<CR>

" Aktuelle Zeile nach oben/unten verschieben
nnoremap <leader>k :m .-2<CR>==
nnoremap <leader>j :m .+1<CR>==

" Suchhighlight löschen
nnoremap <leader>n :nohlsearch<CR>
```

Hier zeigt sich ein typisches Muster:

- Mappings kapseln Ex-Befehle, die man sonst nie vollständig im Kopf behalten würde,
- der eigentliche „Zauber“ steckt in `:m`, `:s`, `:nohlsearch`, das Mapping macht ihn nur abrufbar.

Übungen

1. Ergänze deine `vimrc` um folgende Mappings:

   ```
   nnoremap <leader>w :write<CR>
   nnoremap <leader>q :quit<CR>
   nnoremap <leader>m :write<CR>:make<CR>
   ```

   Öffne ein kleines Projekt, ändere eine Datei und teste den Ablauf [<leader>][w] → [<leader>][m]. Ziel: Speichern und Build sollen sich wie ein einziger zusammengehöriger Workflow anfühlen.
2. Füge Fenster- und Buffer-Mappings hinzu:

   ```
   nnoremap <leader>h <C-w>h
   nnoremap <leader>l <C-w>l
   nnoremap <leader>bn :bnext<CR>
   nnoremap <leader>bp :bprevious<CR>
   ```

   Öffne mehrere Dateien und Splits, wechsle bewusst für ein paar Minuten *nur* mit diesen Mappings. Beobachte, ob du noch [Ctrl]-[w] oder `:bnext` tippst – wenn ja, fehlt dem Mapping vielleicht noch eine intuitivere Belegung.
3. Definiere ein eigenes „Qualitäts-Mapping“, z. B.:

   ```
   nnoremap <leader>t :write<CR>:!pytest %<CR>
   ```

   (oder ein anderes Test-/Lint-Tool deiner Wahl.) Ziel: Ein einziger Shortcut soll immer *deine* Standardprüfung für die aktuelle Datei starten. Überlege anschließend, ob du für alle Sprachen denselben Präfix ([<leader>][t] für Tests) beibehalten kannst, damit deine Mappings langfristig konsistent bleiben.

13.4.4. Abkürzungen (abbreviations)

Abkürzungen sind das Pendant zu Mappings auf Textebene: statt Tastenfolgen in andere Kommandos umzulenken, werden hier *Wörter* beim Tippen automatisch zu längeren Texten erweitert.

Wichtige Unterschiede zu Mappings:

- Abkürzungen greifen beim *Schreiben* von Text (*Insert-Mode*) oder in der *Command-Line* (`:`-Zeile),
- sie werden nur an „Wortgrenzen“ ausgelöst (z. B. nach **<Space>**, **Enter**, **.**, **,** ...),
- sie ersetzen Text durch Text, *nicht* Tastenfolgen.

Insert-Mode-Abkürzungen (iabbrev)

Insert-Abkürzungen werden mit `:iabbrev` definiert. Typische Einsätze sind Standardfloskeln, E-Mail-Adressen oder wiederkehrende Code-Snippets.

```
" Einfache Textabkürzungen
iabbrev mfg Mit freundlichen Grüßen
iabbrev vg  Viele Grüße
iabbrev @jb jens.buttenschoen@example.org
```

Verhalten:

- Du tippst `mfg`, dann **<Space>** → VIM ersetzt `mfg` automatisch durch `Mit freundlichen Grüßen`.
- Die Expansion erfolgt erst, *wenn* die Abkürzung als abgeschlossenes Wort erkannt wird (Wortgrenze).

Abkürzungen auflisten und löschen:

- `:iabbrev` zeigt alle Insert-Abkürzungen.
- `:iunabbrev mfg` löscht genau diese Abkürzung.
- `:abclear` / `:iabclear` löscht alle (Insert-)Abkürzungen.

Command-Line-Abkürzungen (cabbrev)

Command-Line-Abkürzungen wirken in der `:`-Zeile, also bei Ex-Befehlen.

```
" häufige Vertipper abfangen
cnoreabbrev W  w
cnoreabbrev Q  q
cnoreabbrev WQ wq
cnoreabbrev Wq wq

" eigenes Kurz-Kommando
cnoreabbrev CLog copen
```

Typische Anwendungsfälle:

- Schutz vor Groß-/Kleinschreibfehlern (`:W` statt `:w`),
- eigene „Kurznamen“ für längere Kommandos,
- schnelle Sprünge in das Quickfix-Fenster, zum Dateibrowser, etc.

Empfehlung: Nutze konsequent die *nichtrekursiven* Varianten `:cnoreabbrev`, `:inoreabbrev` statt `:cabbrev`/`:iabbrev`, um unerwartete Selbstreferenzen zu vermeiden – analog zu `:noremap` bei Mappings.

Stolperfallen und Grenzen

Abkürzungen wirken nur dann stabil, wenn sie sauber gewählt sind:

- **Keine zu kurzen Trigger** `in`, `an`, `if` sind schlechte Kandidaten – sie kommen in vielen normalen Wörtern vor. Besser: `;mfg`, `;sig`, `todo;`.
- **Wortgrenzen beachten** VIM nutzt die Einstellung `iskeyword`, um zu entscheiden, was ein „Wort“ ist. Abkürzungen werden nur ersetzt, wenn links und rechts eine Wortgrenze vorliegt.
- **Nicht für alles verwenden** Für komplexe Aktionen mit Bewegungen, Fenstern oder Quickfix sind Mappings (`:nnoremap`, `:vnoremap`, ...) das robustere Werkzeug.

Abkürzungen sind am stärksten, wenn sie:

- seltene, aber immer gleich formulierte Textfragmente ersetzen,
- eindeutig erkennbaren Trigger haben,
- im Zweifel „zu lang“ statt zu kurz gewählt sind.

Übungen

1. Füge deiner `vimrc` zwei Insert-Abkürzungen hinzu:

   ```
   iabbrev ;mfg Mit freundlichen Grüßen
   iabbrev ;sig Jens Buttenschön
   ```

   Öffne eine neue Datei, schreibe Text und tippe am Zeilenende `;mfg` **<Space>**. Beobachte genau, wann die Abkürzung ausgelöst wird und wann nicht (z. B. mitten im Wort vs. am Wortende).
2. Definiere Command-Line-Abkürzungen für häufige Vertipper:

   ```
   cnoreabbrev W  w
   cnoreabbrev Q  q
   cnoreabbrev WQ wq
   ```

Tippe bewusst `:W`, `:Q` und `:WQ` und prüfe mit `:cnoreabbrev`, wie VIM sie umsetzt. Ziel: Du solltest „falsch“ tippen können, ohne dass es dich aus dem Flow bringt.

3. Erzeuge eine „Projekt-Kurzform“, z. B.:

   ```
   iabbrev ;todo TODO(jb):
   ```

   Nutze diese Abkürzung in mehreren Dateien und prüfe mit `:vimgrep /TODO(jb)/ **/*`, ob du alle Stellen wiederfindest. Überlege danach, ob du weitere, ähnlich strukturierte Projekt-Snippets als `:iabbrev` hinterlegen möchtest (z. B. Bug-Referenzen, Ticket-IDs, Standardkommentare).

13.5. Autocommands und augroups

13.5.1. Autocommand-Grundlagen

Autocommands („autocmds“) lassen VIM auf Ereignisse reagieren: statt selbst einen Befehl auszuführen, definierst du eine Regel nach dem Muster „Wenn X passiert und die Datei zu Y passt, führe Z aus“. Damit wird VIM konfigurierbar wie ein kleines Ereignis-System.

Grundform:

```
:autocmd \emph {Event} \emph {Muster} \emph
{Befehl}
```

Dabei gilt:

- *Event* – wann soll etwas passieren? (z. B. beim Öffnen, Speichern, Moduswechsel)
- *Muster* – auf welche Dateien/Bedingungen soll es wirken? (z. B. `*.py`, `*.tex`)
- *Befehl* – was soll ausgeführt werden? (Ex-Befehl oder Befehlskette)

Beispiele für typische Ereignisse:

- **BufRead**, **BufReadPost** beim Laden einer Datei in einen Buffer (vor/nach dem Einlesen).
- **BufWritePre**, **BufWritePost** direkt vor bzw. nach dem Speichern.
- **InsertEnter**, **InsertLeave** beim Wechsel in den Insert-Mode bzw. zurück in den Normal-Mode.
- **VimEnter**, **VimLeave** beim Starten bzw. Beenden von VIM.

- **WinEnter**, **WinLeave** beim Wechseln zwischen Fenstern.
- **FileType** sobald der Dateityp erkannt gesetzt wurde (z. B. `python`, `markdown`).

Muster: Dateinamen und Dateitypen

Das *Muster* ist meistens ein Dateiname mit Wildcards:

```
" eigene Formatierung nur für Python
autocmd BufWritePre *.py :%s/\s\+$//e

" andere Farbeinstellungen für Log-Dateien
autocmd BufReadPost *.log setlocal nowrap
```

Hinweis:

- `*` – steht für beliebige Zeichen
- `?` – genau ein Zeichen
- `**` – rekursive Pfade (je nach VIM/Neovim-Version)

Beim Event **FileType** bezieht sich das Muster *nicht* auf den Dateinamen, sondern auf den Wert der Option `filetype`:

```
" gilt für alle Python-Dateien, egal unter welchem Namen
autocmd FileType python setlocal tabstop=4 shiftwidth=4
```

Für Ereignisse, bei denen der Dateiname keine Rolle spielt (z. B. **VimEnter**), wird üblicherweise `*` als Muster verwendet:

```
" beim Start von Vim: letzte Position wiederherstellen
autocmd VimEnter * silent! normal! g`"
```

Diagnosewerkzeuge

Um herauszufinden, welche Autocommands aktiv sind:

- `:autocmd` zeigt alle definierten Autocommands.
- `:autocmd BufWritePost` nur Autocommands für dieses Event anzeigen.
- `:autocmd FileType python` Autocommands für `FileType` „python“ anzeigen.

Dies ist besonders wichtig, wenn sich Konfigurationen aus mehreren Dateien (Plugins, eigene Snippets) gegenseitig beeinflussen.

Typische Stolperfallen

Ohne Struktur kommt es schnell zu Problemen:

- **Doppelte Autocommands** Wenn du deine `vimrc` mehrmals mit `:source \~/.vimrc` lädst, werden dieselben `:autocmd`-Zeilen jedes Mal erneut hinzugefügt. Folge: Aktionen laufen mehrfach (z. B. mehrfaches Ausführen eines Formatierungsbefehls beim Speichern).
- **Zu breite Muster** Ein `:autocmd BufWritePre * \dots` trifft *alle* Dateien – auch temporäre, Log-Files oder Fremd-Projekte. Besser sind gezielte Muster wie `*.c`, `*.tex`, `*.md` oder Pfadfilter.
- **Nebeneffekte durch Plugins** Viele Plugins definieren eigene Autocommands. Wenn du dieselben Events noch einmal „global" belegst, kann es zu unerwarteten Kombinationen kommen (z. B. zwei Formatter, die sich gegenseitig überschreiben).

Die saubere Lösung gegen doppelte Autocommands sind `augroups` mit `:autocmd!` zum vorherigen Löschen — das behandeln wir im nächsten Abschnitt im Detail.

Übungen

1. Erzeuge in deiner `vimrc` ein einfaches Autocommand, das beim Speichern von `*.txt`-Dateien eine kurze Meldung ausgibt:

   ```
   autocmd BufWritePost *.txt echo "TXT-
   Datei gespeichert"
   ```

   Speichere deine `vimrc`, lade sie mit `:source \~/.vimrc` neu und teste das Verhalten an einer `.txt`-Datei. Achte darauf, ob die Meldung *einmal* oder mehrfach erscheint.
2. Definiere eine Datei-typabhängige Einstellung:

   ```
   autocmd FileType markdown setlocal spell
   ```

   Öffne eine Markdown-Datei und prüfe mit `:setlocal spell?`, ob Rechtschreibprüfung nur dort aktiv ist. Öffne danach eine `.c`- oder `.py`-Datei und verifiziere, dass `spell` dort *nicht* gesetzt ist.
3. Untersuche alle aktuell gesetzten Autocommands:
   - `:autocmd BufWritePre`
   - `:autocmd FileType`

Notiere dir, welche davon aus deiner eigenen `vimrc` stammen und welche vermutlich von Plugins kommen. Ziel: Ein Gefühl dafür bekommen, dass die Startkonfiguration aus vielen Quellen besteht – und warum saubere Strukturierung mit `augroups` wichtig ist.

13.5.2. augroup sauber nutzen

Ohne Struktur sammeln sich Autocommands schnell zu einem unübersichtlichen Haufen an. Spätestens, wenn die `vimrc` mehrfach mit `:source` neu geladen wird, entstehen dadurch doppelte oder dreifache Ausführungen derselben `:autocmd`-Zeilen. `augroup` ist das zentrale Werkzeug, um Autocommands sauber zu gruppieren und vor dem Neuladen aufzuräumen.

Grundmuster

Ein typischer Block sieht so aus:

```
augroup FormatOnSave
  autocmd!
  autocmd BufWritePre *.py  %s/\s\+$//e
  autocmd BufWritePre *.js  %s/\s\+$//e
augroup END
```

Wichtig sind dabei drei Dinge:

- `augroup Name` legt eine logisch benannte Gruppe an (hier: `FormatOnSave`).
- `autocmd!` *innerhalb* der Gruppe löscht *alle* bisherigen Autocommands, die zu dieser Gruppe gehören. Das verhindert Dubletten, wenn die `vimrc` mehrfach geladen wird.
- `augroup END` schließt die Gruppe sauber ab.

Das Muster ist immer gleich:

```
augroup MeinThema
  autocmd!
  " ... alle autocmds zu diesem Thema ...
augroup END
```

Aufräumen beim Neuladen der vimrc

Der kritische Punkt ist `autocmd!`: Ohne diesen Befehl würden bei jedem `:source \~/.vimrc` alle Autocommands erneut registriert. Mit `autocmd!` sorgst du dafür, dass die Gruppe vor dem Neuanlegen vollständig geleert wird.

Typischer Ablauf beim Neuladen:

1. `augroup FormatOnSave` – Gruppe auswählen/erstellen.
2. `autocmd!` – alle bisherigen Einträge dieser Gruppe löschen.
3. neue `:autocmd`-Zeilen definieren.
4. `augroup END` – Gruppe schließen.

Dadurch ist garantiert, dass es pro Gruppe und Ereignis nur genau die Autocommands gibt, die aktuell in der `vimrc` stehen – weder mehr noch weniger.

Strukturierung nach Themen

Statt alle Autocommands in einen einzigen Block zu werfen, ist eine thematische Aufteilung deutlich wartbarer. Beispiele:

```
" Formatierung vor dem Speichern
augroup FormatOnSave
  autocmd!
  autocmd BufWritePre *.py  %s/\s\+$//e
  autocmd BufWritePre *.c   %s/\s\+$//e
augroup END

" Linting und Checks
augroup Linting
  autocmd!
  autocmd BufWritePost *.py  !flake8 %
  autocmd BufWritePost *.sh  !shellcheck %
augroup END

" Faltung und Ansicht
augroup Folding
  autocmd!
  autocmd FileType vim  setlocal foldmethod=marker
  autocmd FileType lua  setlocal foldmethod=marker
  autocmd BufWinEnter * silent! normal! zR
```

```
augroup END
```

Vorteile dieser Aufteilung:

- Du kannst gezielt nur einen Block kommentieren oder ändern, ohne andere Bereiche zu berühren.
- Beim Debuggen ist sofort sichtbar, welche Autocommands zu welchem Thema gehören (Formatierung, Linting, Folds, UI-Anpassungen, ...).
- Bei Problemen lässt sich testweise eine ganze Gruppe deaktivieren, indem du den entsprechenden Block vorübergehend auskommentierst.

Typische Muster je Gruppe

Ein paar realistische Muster für Gruppen:

- **Formatierung**

```
augroup FormatOnSave
  autocmd!
  autocmd BufWritePre *.go   lua vim.lsp.buf.format()
  autocmd BufWritePre *.lua  lua vim.lsp.buf.format()
augroup END
```

- **Whitespace-Bereinigung**

```
augroup TrimWhitespace
  autocmd!
  autocmd BufWritePre * %s/\s\+$//e
augroup END
```

- **Faltung und Ansicht**

```
augroup Folding
  autocmd!
  autocmd FileType vim,lua setlocal foldmethod=marker
  autocmd BufWinEnter *     silent! normal! zR
augroup END
```

- **Projekt-/Pfadabhängige Einstellungen**

```
augroup ProjectConfig
  autocmd!
  autocmd BufRead,BufNewFile */myproj/* setlocal tabsto
augroup END
```

Übungen

1. Ersetze in deiner bestehenden `vimrc` alle `:autocmd`-Blöcke, die lose im Dokument verteilt sind, durch logisch benannte Gruppen. Vorschlag:
   - `FormatOnSave`
   - `UISettings`
   - `FiletypeSpecific`

   Füge in jeder Gruppe als erste Zeile `autocmd!` ein. Lade deine `vimrc` mehrfach mit `:source \~/.vimrc` und prüfe, ob Aktionen *nur einmal* ausgeführt werden.
2. Erzeuge eine eigene Gruppe `TrimTrailingSpace`, die beim Speichern *aller* Dateien trailing whitespace entfernt, aber nur für bestimmte Dateitypen:

   ```
   augroup TrimTrailingSpace
     autocmd!
     " TODO: nur für *.c, *.h, *.py
   augroup END
   ```

   Ergänze die passende `:autocmd`-Zeile und teste an mehreren Dateitypen, ob das Verhalten deinen Erwartungen entspricht.
3. Lege eine Gruppe `ProjectMyApp` an, die nur für Dateien unter einem bestimmten Pfad (z. B. `*/myapp/*`) spezielle Optionen setzt (z. B. `shiftwidth`, `tabstop`, `makeprg`). Öffne eine Datei innerhalb und außerhalb dieses Pfads und vergleiche mit `:setlocal` die jeweils aktiven Optionen. Ziel: Verstehen, wie `augroup` + `autocmd`-Muster eine projektbezogene Konfiguration ermöglichen, ohne die globale `vimrc` zu verschmutzen.

13.5.3. Typische Anwendungsfälle

Autocommands entfalten ihre Stärke vor allem in wiederkehrenden Situationen: Formatierung beim Speichern, Aufräumen von Whitespace und das Setzen dateitypspezifischer Optionen. Ziel ist, manuelle Routinehandlungen zu automatisieren – ohne unkontrollierbare Nebenwirkungen.

Automatische Formatierung beim Speichern

Ein klassisches Muster ist das automatische Formatieren vor dem Speichern:

```
augroup FormatOnSave
  autocmd!
  " Beispiel: Trailing Whitespace vor dem Speichern entfern
  autocmd BufWritePre * %s/\s\+$//e
augroup END
```

- `BufWritePre` – Ereignis kurz vor dem Schreiben der Datei.
- `*` – gilt für alle Dateinamen.
- `%s/\s\+$//e` – entfernt Leerraum am Zeilenende („trailing whitespace“).

Für Sprachen mit Language-Server oder Format-Tool:

```
augroup LspFormatOnSave
  autocmd!
  autocmd BufWritePre *.go,*.lua  lua vim.lsp.buf.format()
augroup END
```

So wird nur in ausgewählten Dateitypen formatiert.

Trailing Whitespace markieren oder löschen

Statt Whitespace immer zu löschen, kann er auch nur optisch hervorgehoben werden:

```
augroup HighlightTrailingWhitespace
  autocmd!
  autocmd ColorScheme * highlight ExtraWhitespace ctermbg=d
  autocmd BufWinEnter,BufNewFile * match ExtraWhitespace /\
augroup END
```

- Eigene Highlight-Gruppe `ExtraWhitespace`.
- `match` markiert Leerraum am Zeilenende.
- `ColorScheme`-Autocommand sorgt dafür, dass die Farbe auch nach einem `:colorscheme`-Wechsel erhalten bleibt.

Wer beides kombinieren will (erst sehen, dann bei Bedarf löschen), kann separate Gruppen für Markierung und Entfernung anlegen und die Lösch-Gruppe bei Bedarf temporär deaktivieren.

Dateitypspezifische Optionen

Typisch ist das automatische Setzen von Optionen pro Dateityp:

```
augroup FiletypeSettings
  autocmd!
  " Python: 4er Indent, Soft-Tabs
  autocmd FileType python setlocal tabstop=4 shiftwidth=4 expandta

  " Markdown: Textbreite und automatische Zeilenumbrüche
  autocmd FileType markdown setlocal textwidth=80 formatoptions+=t

  " Makefiles: echte Tabs nötig
  autocmd FileType make setlocal noexpandtab tabstop=8 shiftwidth=
augroup END
```

Weitere Varianten:

- Pfadabhängig (Projektkonfiguration):

  ```
  augroup ProjectMyApp
    autocmd!
    autocmd BufRead,BufNewFile */myapp/* setlocal tabstop=2 shif
  augroup END
  ```

- Logdateien immer am Ende öffnen:

  ```
  augroup Logs
    autocmd!
    autocmd BufReadPost *.log normal! G
  augroup END
  ```

Übungen

1. Lege eine Gruppe `TrimOnSave` an, die bei *allen* Dateien trailing whitespace entfernt, *außer* in Markdown-Dateien. Teste das Verhalten in verschiedenen Dateitypen und beobachte mit `:setlocal`, welche Optionen/Befehle aktiv sind.

2. Erzeuge eine Gruppe `VisualWhitespace`, die trailing whitespace nur markiert, aber nicht löscht. Schalte die Markierung testweise durch Auskommentieren der `match`-Zeile aus und wieder ein, um den Unterschied zwischen „sehen“ und „automatisch aufräumen“ bewusst wahrzunehmen.
3. Richte eine Gruppe `ProjectConfig` ein, die für Dateien unter `*/projektx/*` eigene Einstellungen setzt (`tabstop`, `shiftwidth`, `textwidth`). Öffne Dateien innerhalb und außerhalb des Pfades und vergleiche die Optionen mit `:setlocal`. Ziel: Verstehen, wie sich projekt- oder dateitypspezifisches Verhalten sauber über Autocommands kapseln lässt.

13.6. Dateityp-spezifische Konfiguration

13.6.1. filetype plugin indent on

VIM besitzt ein eigenes Dateityp-System, das anhand von Dateiendungen, Shebang-Zeilen oder Inhalt einen *filetype* zuweist (z. B. `python`, `markdown`, `sh`). Erst wenn dieses System aktiviert ist, können dateitypspezifische Plugins und Einrückungsregeln voll genutzt werden.

Die zentrale Direktive dafür lautet:

```
filetype plugin indent on
```

Sie wird üblicherweise *einmalig* in der vimrc gesetzt und hat drei Effekte:

- **filetype**: Dateityperkennung wird eingeschaltet (`:filetype on`).
- **plugin**: dateitypspezifische Plugins unter `ftplugin/` werden geladen (`:filetype plugin on`).
- **indent**: dateitypspezifische Einrückungsskripte unter `indent/` werden aktiv (`:filetype indent on`).

Damit stehen z. B. für `python`, `c`, `markdown` oder `vim` sofort passende Einstellungen für Einrückung, Formatierung oder Hilfsfunktionen zur Verfügung, ohne dass man alles selbst setzen muss.

Unterschied zu `:syntax on`

- `:syntax on` steuert ausschließlich das Syntax-Highlighting (Farben, Gruppen, Schlüsselwörter).

- `:filetype plugin indent on` steuert Dateityp-Erkennung sowie das Laden von Plugins und Indent-Skripten, also *Verhalten*, nicht nur Farbe.

In einer gut gepflegten vimrc stehen daher meist beide Zeilen:

```
syntax on
filetype plugin indent on
```

Diagnose: Welcher Dateityp, welche Skripte?

Zur Kontrolle des Dateityps:

- `:filetype`

VIM meldet dann z. B.:

```
filetype=python  filetype detection:ON  plugin:ON  indent:ON
```

Damit lässt sich schnell erkennen, ob Erkennung, Plugins und Indent wirklich aktiv sind.

Welche Skripte konkret geladen wurden, zeigt:

- `:scriptnames`

Dort tauchen u. a. Dateien wie:

```
... indent/python.vim
... ftplugin/python.vim
```

auf. So lässt sich nachvollziehen, welche Konfigbausteine den aktuellen Buffer beeinflussen.

Typische Stolperfallen

- **Reihenfolge in der vimrc**: `filetype plugin indent on` sollte nicht in einer bedingten Umgebung versteckt sein, die nur manchmal ausgeführt wird. Am besten früh, aber nach grundlegenden Optionen platzieren.
- **Lokale Overrides**: Eigene `ftplugin`- oder `indent`-Dateien im Benutzerverzeichnis können das Verhalten modifizieren oder überschreiben. Bei merkwürdigem Verhalten lohnt ein Blick in `:scriptnames`.
- **Unbekannte Dateiendung**: Wenn VIM keinen Dateityp erkennt, bleibt `filetype` leer. In solchen Fällen kann man testweise manuell setzen:
  - `:set filetype=python`

  und anschließend prüfen, welche Plugins/Indent-Regeln greifen.

Übungen

1. Öffne eine Datei `test.py` und stelle sicher, dass in deiner vimrc die Zeile

   ```
   filetype plugin indent on
   ```

   gesetzt ist. Prüfe mit:
   - `:filetype`

   ob `filetype=python`, `plugin:ON` und `indent:ON` gemeldet werden.
2. Öffne nacheinander Dateien mit unterschiedlichen Endungen (z. B. `.c`, `.md`, `.vim`) und rufe jeweils:
   - `:filetype`
   - `:setlocal shiftwidth? tabstop? expandtab?`

   auf. Ziel: Ein Gefühl dafür bekommen, welche dateitypspezifischen Einstellungen automatisch gesetzt werden.
3. Führe `:scriptnames` aus und filtere in deinem Kopf nach Zeilen, die `ftplugin` oder `indent` enthalten. Überlege für einen deiner meistgenutzten Dateitypen, welche Skripte tatsächlich geladen werden und ob du bei Bedarf ein eigenes `ftplugin/<typ>.vim` ergänzen möchtest.

13.6.2. `ftplugin`-Dateien

Statt die komplette Logik in einer monolithischen `vimrc` zu bündeln, bietet VIM für jeden Dateityp eigene Konfigurationsdateien unterhalb von `ftplugin/` an. Diese Dateien werden automatisch geladen, sobald ein Buffer mit passendem `filetype` geöffnet wird.

Typischer Speicherort (klassisches VIM):

```
~/.vim/ftplugin/<filetype>.vim
```

Bei Neovim:

```
~/.config/nvim/ftplugin/<filetype>.vim
```

Beispiele:

- `/.vim/ftplugin/markdown.vim` Einstellungen nur für Markdown-Dateien.

- `/.vim/ftplugin/python.vim` Einrückung, Falten, Linting nur für Python.
- `/.vim/ftplugin/vim.vim` Komfortoptionen, sobald VIM-Skripte bearbeitet werden.
- `/.vim/ftplugin/tex.vim` LaTeX-spezifische Optionen (`textwidth`, Formatierung etc.).

Ein einfaches `ftplugin`-Beispiel

Beispiel: eigene Einstellungen für Markdown-Dateien:

```
" ~/.vim/ftplugin/markdown.vim
setlocal spell
setlocal wrap
setlocal linebreak
setlocal textwidth=80
```

- `:setlocal` stellt sicher, dass die Optionen nur für diesen Buffer gelten.
- Andere Dateitypen bleiben von diesen Einstellungen unberührt.

Für Python könnte eine Datei `python.vim` so aussehen:

```
" ~/.vim/ftplugin/python.vim
setlocal expandtab
setlocal tabstop=4
setlocal shiftwidth=4
setlocal softtabstop=4
```

Sobald eine `.py`-Datei mit `:edit` geöffnet wird und der `filetype` korrekt erkannt ist, lädt VIM diese Datei automatisch.

Vorteile gegenüber „`if &filetype == ...`“ in der vimrc

Natürlich könnte man auch in der `vimrc` Konstrukte wie:

```
if &filetype ==# 'python'
  setlocal expandtab
  setlocal shiftwidth=4
endif
```

verwenden. In der Praxis ist das aus mehreren Gründen die schlechtere Variante:

- **Übersichtlichkeit** Die `vimrc` bleibt frei von langen `if`-Blöcken für jeden Dateityp. Stattdessen liegt die Logik dort, wo sie hingehört: `ftplugin/<filetype>.vim`.
- **Wiederverwendbarkeit** `ftplugin`-Dateien lassen sich leicht in andere Setups kopieren oder per Versionskontrolle teilen, ohne die komplette `vimrc` zu verschleppen.
- **Automatische Ladereihenfolge** VIM kümmert sich darum, dass `ftplugin`-Dateien genau dann geladen werden, wenn der `filetype` gesetzt wird – auch beim nachträglichen Ändern von `filetype`.
- **Lokale Overrides** Neben globalen `ftplugin`-Dateien können projekt- oder nutzerspezifische Varianten (`after/ftplugin`) ergänzt werden, ohne den Kernbestand anzufassen.

Kurz gesagt: `ftplugin`-Dateien sind modular, wartbar und passen sauber in das Dateityp-System von VIM. Die Konstruktion `if &filetype == ...` in der `vimrc` ist nur für kleine Schnell-Hacks sinnvoll, nicht für eine langfristig gepflegte Konfiguration.

Übungen

1. Prüfe zunächst deinen aktuellen `filetype` in einer geöffneten Datei mit:
   - `:filetype`

   Merke dir einen Dateityp, den du häufig verwendest (z. B. `markdown`, `python`, `tex`).
2. Lege für diesen Dateityp eine eigene `ftplugin`-Datei an, z. B.:

   ```
   ~/.vim/ftplugin/markdown.vim
   ```

   und füge dort mindestens drei `:setlocal`-Optionen ein, die nur für diesen Dateityp gelten sollen.
3. Öffne anschließend mehrere Dateien *unterschiedlicher* Dateitypen und vergleiche mit:
   - `:setlocal` (ohne Argument)

   welche Optionen jeweils aktiv sind. Ziel: Klar erkennen, dass deine `ftplugin`-Konfiguration wirklich nur den gewünschten Dateityp beeinflusst.
4. Entferne testweise den Dateityp, indem du:
   - `:set filetype=`

ausführst, und setze ihn dann wieder:

- `:set filetype=<deintyp>`

Beobachte, dass VIM beim Setzen des `filetype` deine `ftplugin`-Datei erneut lädt (z. B. anhand von `:setlocal textwidth?`).

13.6.3. Indent- und Syntax-Anpassungen

Neben `ftplugin`-Dateien kennt VIM zwei weitere Bausteine für dateitypspezifisches Verhalten: *Indent*- und *Syntax*-Dateien. Sie steuern, wie VIM Einrückungen berechnet und wie Code farblich hervorgehoben wird.

Indent-Dateien

Indent-Dateien definieren, wie VIM neue Zeilen einrückt und bestehende Zeilen nachformatiert. Typische Speicherorte:

```
~/.vim/indent/<filetype>.vim
~/.config/nvim/indent/<filetype>.vim
```

Beispiele:

- `indent/python.vim` — Python-spezifische Einrückregeln
- `indent/c.vim` — Einrückung für C/CPP
- `indent/tex.vim` — Einrückung für LaTeX-Umgebungen

Damit diese Dateien überhaupt genutzt werden, sollte in der vimrc grundsätzlich das Dateityp-System aktiviert sein:

```
filetype indent on
```

In vielen Fällen reichen die mitgelieferten Indent-Skripte aus. Eigene Indent-Dateien lohnen sich vor allem, wenn:

- eine Sprache / DSL gar keinen Indent-Support hat,
- das Projekt sehr spezielle Einrückungsregeln verlangt,
- Autoformatierung (`:=` / `:gg=G`) exakt zum Coding-Style passen soll.

Syntax-Dateien

Syntax-Dateien definieren, wie VIM Schlüsselwörter, Strings, Kommentare oder spezielle Sprachkonstrukte erkennt und highlightet. Typische Pfadstruktur:

```
~/.vim/syntax/<filetype>.vim
~/.config/nvim/syntax/<filetype>.vim
```

Aktiviert wird das System global mit:

```
syntax on
filetype plugin indent on
```

Eigene Syntax-Dateien sind sinnvoll, wenn:

- eine Sprache nicht unterstützt wird (interne DSL, Konfigsprache),
- Standard-Syntax-Dateien wichtige Konstrukte ignorieren,
- bestimmte Tokens (z. B. TODO, FIXME, eigene Keywords) besonders hervorgehoben werden sollen.

Für kleinere Ergänzungen ist es meist nicht nötig, die komplette Syntax-Datei neu zu schreiben. Stattdessen bietet sich das `after`-Verzeichnis an.

`after/`-Verzeichnis für sanfte Anpassungen

Statt Original-Dateien zu überschreiben (riskant bei Updates), können eigene Ergänzungen in `after/`-Verzeichnissen abgelegt werden:

```
~/.vim/after/indent/<filetype>.vim
~/.vim/after/syntax/<filetype>.vim

~/.config/nvim/after/indent/<filetype>.vim
~/.config/nvim/after/syntax/<filetype>.vim
```

VIM lädt zunächst die normalen Indent-/Syntax-Dateien und danach alles unterhalb von `after/`. Typische Anwendungsfälle:

- zusätzliche Keywords für eine bestehende Sprache definieren,
- Farbgruppen feintunen, ohne das Original anzufassen,
- einzelne Indent-Regeln leicht anpassen (z. B. Sonderfälle in Projekten).

Wann lohnt sich eigener Syntax-/Indent-Code?

Nicht jeder Wunsch rechtfertigt eine eigene Datei. Ein grober Daumen:

- **Lohnt sich:** Regelmäßige Arbeit an derselben Sprache / demselben Format, bei der Einrückung und Syntax-Hervorhebung ein echter Produktivitätsfaktor sind (z. B. Haus-DSL, zentraler Config-Stack, täglicher Projektcode).

- **Selten sinnvoll:** Einmalige Quick-Hacks für eine Konfig-Datei, die man nie wieder sieht. Hier ist es oft schneller, kurz manuell zu formatieren.
- **Warnsignal:** Wenn die Pflege der Syntax-/Indent-Dateien mehr Zeit frisst als das tatsächliche Editieren des Formats, ist die Balance falsch.

Übungen

1. Öffne eine Datei eines bekannten Typs (z. B. `.py`, `.c`, `.tex`) und prüfe:
   - `:set filetype?`
   - `:filetype`

   Stelle sicher, dass VIM den Dateityp korrekt erkannt hat.
2. Aktiviere explizit:
   - `:syntax on`
   - `:filetype plugin indent on`

   Öffne die Datei erneut und beobachte Einrückung (`:=`) und Syntax-Highlighting. Nutze `:scriptnames`, um zu sehen, welche `syntax`/`indent`-Dateien geladen wurden.
3. Lege eine minimale `after`-Syntax-Datei an, z. B.:

   ```
   ~/.vim/after/syntax/markdown.vim
   ```

   und füge dort z. B. ein zusätzliches Keyword oder eine Hervorhebung hinzu (zur Not zunächst nur als Kommentar mit `" TODO: hier später erweitern`). Ziel: Verstehen, wo eigene Anpassungen landen würden.
4. Dokumentiere anschließend im Kommentarbereich deiner vimrc kurz, für welche Dateitypen du *persönlich* irgendwann Syntax-/Indent-Anpassungen planst und für welche nicht. Ziel: Klarheit darüber, wo sich weiterer Invest tatsächlich lohnt.

13.7. Plugin-Management (optional)

13.7.1. Plugins ohne Plugin-Manager

VIM (ab Version 8) und Neovim bringen einen eingebauten Mechanismus für Plugins mit, ganz ohne zusätzlichen Plugin-Manager. Die zentrale Idee ist das *Paket-System* über `packpath` und `:packadd`: Plugins werden in einem definierten Verzeichnisbaum abgelegt und von dort entweder automatisch oder auf Befehl geladen.

Grundprinzip: packpath und Paket-Verzeichnisse

VIM durchsucht beim Start alle Verzeichnisse unterhalb von packpath. Typische Strukturen sind:

```
~/.vim/pack/<gruppe>/start/<pluginname>/
~/.vim/pack/<gruppe>/opt/<pluginname>/
```

unter Neovim z. B.:

```
~/.local/share/nvim/site/pack/<gruppe>/start/<pluginname>/
~/.local/share/nvim/site/pack/<gruppe>/opt/<pluginname>/
```

Wichtig sind dabei vor allem zwei Ebenen:

- start/ Alles, was hier liegt, wird beim Start automatisch geladen.
- opt/ „Optionale“ Plugins: Sie werden nur geladen, wenn du sie explizit mit :packadd anforderst.

Der Name <gruppe> ist frei wählbar („vendor“, „jb“, „thirdparty“, …); er dient nur der Strukturierung.

Plugin ohne Manager installieren

Statt einen Plugin-Manager zu benutzen, kannst du ein Git-Repository direkt in den Paketbaum klonen, z. B.:

```
mkdir -p ~/.vim/pack/jb/start
cd ~/.vim/pack/jb/start
git clone https://github.com/tpope/vim-sensible.git
```

Beim nächsten Start von VIM wird vim-sensible automatisch geladen, weil es unterhalb von start/ liegt.

Für optionale Plugins:

```
mkdir -p ~/.vim/pack/jb/opt
cd ~/.vim/pack/jb/opt
git clone https://github.com/preservim/nerdtree.git
```

In der vimrc kannst du das Plugin dann gezielt aktivieren:

```
packadd nerdtree
```

oder bedarfsweise im laufenden VIM mit:

- :packadd nerdtree

Vorteile gegenüber klassischen Plugin-Managern

Plugins über `packpath` zu verwalten hat einige klare Vorteile:

- Kein zusätzlicher Plugin-Manager nötig Weniger Abhängigkeiten, weniger Magie, keine Bootstrapping-Skripte.
- Standardmechanismus Funktioniert in allen aktuellen VIM-/Neovim-Versionen gleich und ist im `:help` dokumentiert (`:h packages`, `:h pack-add`).
- Gut kombinierbar mit Git Der gesamte Paketbaum kann einfach in ein persönliches Dotfiles-Repository aufgenommen werden.
- Klare Trennung Durch frei wählbare Gruppen (`jb`, `vendor`, `thirdparty`) lässt sich sauber unterscheiden zwischen eigenen Skripten und Fremd-Plugins.

Nachteile und Grenzen

Dem stehen einige Einschränkungen gegenüber:

- Kein automatisches Installieren/Updaten „`:PlugUpdate`“ oder „`:PackerSync`“ gibt es hier nicht. Updates laufen über `git pull` oder manuelle Downloads.
- Kein eingebaute Lazy-Load-Logik Zwar können Plugins in `opt/` liegen und nur per `:packadd` geladen werden, komplexere Lazy-Loading-Szenarien (z. B. „nur für diesen Dateityp“, „nur bei bestimmtem Befehl“) muss man selbst mit `autocmd` und Mappings bauen.
- Weniger Komfort Dinge wie automatische Snapshot-Verwaltung, Lock-Dateien oder Abhängigkeitsauflösung liefern reine `packpath`-Setups nicht von allein.

Für viele Nutzer reicht der eingebaute Mechanismus trotzdem vollständig aus, insbesondere wenn nur eine überschaubare Anzahl an Plugins genutzt wird und man Updates bewusst kontrolliert.

Übungen

1. Prüfe in deinem VIM/Neovim, wie `packpath` aktuell gesetzt ist:
   - `:set packpath?`

   Notiere dir das Verzeichnis, unter dem deine eigenen Plugin-Pakete sinnvollerweise liegen sollten.

2. Lege eine eigene Paketgruppe „jb“ an und installiere testweise ein kleines Plugin ohne Plugin-Manager, z. B. ein minimalistisches Statusline- oder Kommentar-Plugin:

   ```
   mkdir -p ~/.vim/pack/jb/start
   cd ~/.vim/pack/jb/start
   git clone <plugin-url>
   ```

   Starte VIM neu und überprüfe mit `:scriptnames`, ob das Plugin geladen wurde.
3. Installiere ein zweites Plugin unterhalb von

   ```
   ~/.vim/pack/jb/opt/<pluginname>/
   ```

   und lade es explizit mit:
   - `:packadd <pluginname>`

   Überprüfe im `:help`, welche neuen Befehle oder Mappings es bereitstellt. Ziel: Den Unterschied zwischen `start/` und `opt/` praktisch nachvollziehen.
4. Ergänze in deiner `vimrc` einen kommentierten Abschnitt „Plugins ohne Manager“, in dem du:
   - den Basis-Pfad (`packpath`) dokumentierst,
   - deine Paketgruppen (`vendor`, `jb`, ...) kurz beschreibst,
   - und optional `:packadd`-Zeilen für optionale Plugins einträgst.

   Ziel: Eine nachvollziehbare, wartbare Struktur für zukünftige Plugin-Erweiterungen schaffen.

13.7.2. Plugin-Manager grob einordnen

Neben dem eingebauten Paket-System (`packpath`) existieren eine ganze Reihe spezialisierter Plugin-Manager. Bekannte Beispiele sind `vim-plug`, `dein.vim`, `paq-nvim` oder `lazy.nvim` (hauptsächlich für Neovim). Ziel dieser Werkzeuge ist es, Installation, Update und ggf. Lazy-Loading von Plugins zu vereinfachen.

Deklaration vs. Konfiguration

Ein wichtiges Prinzip bei Plugin-Managern ist die Trennung zwischen:

- **Deklaration** Hier wird festgelegt, *welche* Plugins überhaupt verwendet werden sollen. Typisch ist eine zentrale Liste von Git-URLs oder Paketnamen,

z. B. in einer Datei `plugins.vim` oder in einem speziellen Abschnitt der `vimrc`.
- **Konfiguration** Hier werden die einzelnen Plugins eingerichtet: Optionen, Mappings, Autocommands, Farben, usw. Diese Einstellungen gehören in eigene Blöcke oder Dateien („plugin-config“) und sollten so strukturiert sein, dass sie unabhängig von der Installationsmethode lesbar bleiben.

Die Deklaration antwortet auf die Frage: „Welche Bausteine gehören in meinen Editor?“ Die Konfiguration beantwortet: „Wie verhalten sich diese Bausteine in meinem Setup?“

Typische Aufteilung: „plugins.vim“ vs. „plugin-config.vim“

In der Praxis hat sich folgende Struktur bewährt:
- `plugins.vim` (oder ein dedizierter Abschnitt in der `vimrc`) Enthält *nur* die Deklaration der Plugins. Beim Start sorgt der Plugin-Manager dafür, dass alle dort aufgeführten Repositories vorhanden sind und bei Bedarf geladen werden.
- `plugin-config.vim` (bzw. mehrere thematische Dateien) Enthalten die konkreten Einstellungen für die einzelnen Plugins: Mappings, Befehle, Optionen, Autocommands. Diese Dateien können über `:source` eingebunden werden und sind inhaltlich weitgehend unabhängig davon, ob die Plugins über `packpath` oder einen Manager installiert wurden.

Vorteile dieser Trennung:
- Die Liste der Plugins bleibt übersichtlich und schnell änderbar.
- Konfiguration kann in kleinere, thematische Dateien aufgeteilt werden (z. B. `plugin-config/git.vim`, `plugin-config/statusline.vim`, ...).
- Ein Wechsel des Plugin-Managers oder ein Rückfall auf den eingebauten `packpath`-Mechanismus ist einfacher, weil die eigentliche Konfiguration nicht im Manager-Syntax-Chaos untergeht.

Plugin-Manager: Wann lohnen sie sich?

Ein dedizierter Plugin-Manager lohnt sich typischerweise dann, wenn:
- du deutlich mehr als nur wenige Plugins nutzt,
- du regelmäßig Updates einspielen möchtest („alle Plugins mit einem Kommando aktualisieren“),

- du Lazy-Loading-Features brauchst („nur bei bestimmtem Dateityp“, „nur bei bestimmtem Kommando“),
- du dein Setup auf mehreren Maschinen synchron hältst und eine reproduzierbare Installation wünschst.

Bei sehr kleinen, stabilen Setups kann der eingebaute `packpath` + manuelle Git-Pflege völlig ausreichen. Entscheidend ist, die Verantwortlichkeiten sauber zu trennen:

- „Wie kommt das Plugin auf die Platte?“ (Manager oder `git clone` / Paket-System)
- „Wie wird das Plugin in VIM konfiguriert?“ (Klar strukturierte Konfig-Dateien, unabhängig vom Manager)

Übungen

1. Erstelle in deinem Konfigurationsverzeichnis (z. B. $HOME/`.vim` oder $HOME/`.config/nvim`) zwei Dateien:
   - `plugins.vim`
   - `plugin-config.vim`

   und binde sie in deiner `vimrc` ein, z. B.:

   ```
   " Deklaration
   source ~/.vim/plugins.vim

   " Konfiguration
   source ~/.vim/plugin-config.vim
   ```

   Ziel: Die logische Trennung einmal konkret anlegen, auch wenn du zunächst nur wenige Plugins verwendest.
2. Dokumentiere in `plugins.vim` alle Plugins, die du bereits nutzt – egal, ob sie über `packpath` oder einen Plugin-Manager installiert sind. Nutze Kommentare, um kurz den Zweck jedes Plugins zu beschreiben. Ziel: Eine aktuelle, menschenlesbare Übersicht deiner „Plugin-Landschaft“ erhalten.
3. Verschiebe vorhandene Plugin-spezifische Einstellungen aus der `vimrc` in `plugin-config.vim` (oder mehrere thematische Dateien). Achte darauf, dass jede Datei klar kommentiert ist („Git-Integration“, „Statusline“, „Fuzzy Finder“, ...). Ziel: Deine Konfiguration so zu strukturieren, dass du später problemlos den Installationsmechanismus ändern kannst, ohne die eigentliche Konfiguration neu schreiben zu müssen.

13.7.3. Konfigurationsstruktur für Plugins

Damit Plugin-Konfiguration dauerhaft wartbar bleibt, lohnt sich eine klare Struktur: statt alle Einstellungen in einen einzigen Block der `vimrc` zu kippen, wird jede Erweiterung in einen eigenen Abschnitt oder sogar eine eigene Datei ausgelagert.

Pro Plugin eigener Block / eigene Datei

Zwei bewährte Muster:

- **Eigene Blöcke in einer Gesamtdatei** Zum Beispiel eine Datei `plugin-config.vim` mit sauber kommentierten Abschnitten:

  ```
  " ==================== Git-Integration ====================
  " Plugin: tpope/vim-fugitive
  nnoremap <leader>gs :G<CR>
  nnoremap <leader>gd :Gdiffsplit<CR>

  " ==================== Fuzzy Finder =======================
  " Plugin: nvim-telescope/telescope.nvim
  nnoremap <leader>ff :Telescope find_files<CR>
  nnoremap <leader>fg :Telescope live_grep<CR>
  ```

  Vorteil: alle Plugin-Configs an einem Ort, aber trotzdem klar getrennt und kommentiert.
- **Eigene Datei pro Plugin** Alternativ erhält jedes Plugin eine eigene Datei, z. B.:

  ```
  ~/.vim/plugin-config/fugitive.vim
  ~/.vim/plugin-config/telescope.vim
  ~/.vim/plugin-config/treesitter.vim
  ```

  Die `vimrc` lädt diese Dateien dann explizit:

  ```
  source ~/.vim/plugin-config/fugitive.vim
  source ~/.vim/plugin-config/telescope.vim
  ```

  Vorteil: einzelne Plugins können leicht ein- und ausgeschaltet oder separat versioniert werden.

Wichtig ist weniger die konkrete Struktur als deren *Konsequenz*: Pro Plugin gibt es einen klar abgegrenzten Konfigurationsbereich (Namenskommentar oder eigene Datei), der alle zugehörigen Optionen, Mappings und Autocommands enthält.

Lazy-Loading vs. immer geladen

Je nach Setup werden Plugins entweder:

- **immer geladen** (z. B. im `packpath`-Verzeichnis `start/`), oder
- **lazy geladen**, also nur bei Bedarf (Dateityp, Kommando, Mapping, ...).

Die Konfigurationsstruktur sollte dazu passen:

- **Immer geladene Plugins** Konfiguration kann direkt beim Start ausgeführt werden, z. B.:

```
" ~/.vim/plugin-config/fugitive.vim
" Wird beim Start geladen - Plugin ist bereits verfügba
nnoremap <leader>gs :G<CR>
nnoremap <leader>gc :Gcommit<CR>
```

- **Lazy geladene Plugins** Hier muss sichergestellt sein, dass Konfiguration und Mappings *erst dann* greifen, wenn das Plugin verfügbar ist. Typische Muster:
  - Konfig-Dateien über einen Plugin-Manager erst nach „Load-Events“ sourcen.
  - „Stub-Mappings“ definieren, die beim ersten Aufruf zunächst das Plugin nachladen und dann den eigentlichen Befehl ausführen.

Beispiel für ein defensives Mapping, das nur aktiv wird, wenn das Plugin existiert:

```
" ~/.vim/plugin-config/fugitive.vim
if exists(':G')
  nnoremap <leader>gs :G<CR>
  nnoremap <leader>gd :Gdiffsplit<CR>
endif
```

So verhindert man Fehlermeldungen, falls das Plugin auf einem System noch nicht installiert ist.

Abhängigkeiten an passender Stelle definieren

Viele Plugin-Konfigurationen bestehen aus drei Bausteinen:

- Optionen (`let g:...`, `set ...`)
- Keymaps (`:nnoremap`, `:vnoremap`, ...)
- Autocommands (`:autocmd`, `:augroup`)

Diese sollten *immer* in unmittelbarer Nähe zueinander stehen, idealerweise in der gleichen Datei wie das Plugin selbst:

```
" ~/.vim/plugin-config/telescope.vim

" 1) Optionen / Setup
lua << EOF
require('telescope').setup{
  defaults = {
    layout_strategy = 'horizontal',
  }
}
EOF

" 2) Mappings
nnoremap <leader>ff :Telescope find_files<CR>
nnoremap <leader>fg :Telescope live_grep<CR>

" 3) Autocommands (optional)
augroup telescope_custom
  autocmd!
  autocmd FileType telescope setlocal nonumber norelativenumber
augroup END
```

Vorteile:

- Alle Abhängigkeiten („nutzt `:Telescope`, braucht das Plugin“) sind lokal sichtbar.
- Beim Entfernen eines Plugins kann die gesamte zugehörige Datei gelöscht werden, ohne Reste in der `vimrc`.
- Autocommands, die Plugin-Kommandos aufrufen, liegen unmittelbar beim Plugin selbst und werden nicht versehentlich früher ausgeführt.

Übungen

1. Identifiziere in deiner aktuellen `vimrc` mindestens zwei Plugins, deren Konfiguration über mehrere Stellen verteilt ist (Optionen, Mappings, Autocommands). Fasse diese Konfiguration jeweils in einem eigenen Block zusammen und kommentiere ihn klar mit dem Pluginnamen.

2. Lege ein Verzeichnis `plugin-config/` in deinem VIM-Konfigurationspfad an (z. B. `$HOME/.vim/plugin-config`) und verschiebe die Konfiguration *eines* Plugins in eine eigene Datei, z. B. `plugin-config/git.vim`. Binde diese Datei anschließend explizit über `:source` in deiner `vimrc` ein.
3. Suche ein Plugin, das derzeit immer geladen wird, obwohl du es nur für bestimmte Dateitypen oder Aufgaben benötigst. Überlege, ob sich ein Lazy-Loading-Ansatz lohnt (z. B. Laden nur für `FileType python` oder nur bei bestimmten Kommandos) und notiere dir ein mögliches Konzept dazu in einem Kommentarblock in der entsprechenden Plugin-Konfigurationsdatei. Ziel ist nicht die perfekte Lösung, sondern ein klares Lade-Konzept pro Plugin zu formulieren.

13.8. Performance, Diagnose und Debugging

13.8.1. Startzeit analysieren

Wenn sich VIM „langsam anfühlt“, liegt das oft gar nicht am Programm selbst, sondern an der geladenen Konfiguration: Plugins, Autocommands, Dateityp-Logik und Syntax-Definitionen können den Start deutlich verlangsamen. Ziel dieses Abschnitts ist es, den Startweg so zu zerlegen, dass sich die eigentlichen Bremsklötze identifizieren lassen.

Startzeit messen mit `--startuptime`

VIM (und Neovim) können detailliert protokollieren, was beim Start passiert:

```
vim --startuptime startup.log
```

oder unter Neovim:

```
nvim --startuptime startup.log
```

Danach ganz normal eine Datei öffnen oder direkt `:q` ausführen, um VIM wieder zu beenden. Die Datei `startup.log` enthält nun eine zeitlich sortierte Liste aller Schritte, z. B.:

```
000.012  sourcing /usr/share/vim/vim90/filetype.vim
012.345  sourcing ~/.vim/plugin/fugitive.vim
045.678  sourcing ~/.vim/plugin/treesitter.vim
```

Wichtige Punkte beim Lesen:

- Die erste Spalte zeigt die *vergangene Zeit seit Start* (in Millisekunden).
- Große Sprünge zwischen zwei Zeilen deuten auf „teure“ Skripte hin (z. B. einzelne Plugins oder sehr umfangreiche Syntax-Dateien).
- „sourcing“ zeigt, welche Dateien (vimrc, Plugin-Skripte, ftplugin, Syntax, …) tatsächlich geladen wurden.

Typische Bremsen:

- große Plugin-Sammlungen, die *alle* beim Start geladen werden,
- aufwendige Statusleisten- oder Tree-Plugins,
- komplexe Datei-Typ- oder Syntax-Definitionen,
- „workarounds“, die bei jedem Start teure externe Tools oder Shell-Befehle aufrufen.

Vergleich: volle vs. minimale Konfiguration

Um herauszufinden, ob die eigene `vimrc` (oder Neovim-`init.vim`) der Hauptverursacher ist, kann man mit reduzierter Konfiguration starten.

Ohne Konfiguration starten

```
vim -u NONE -U NONE --noplugin
```

- `-u NONE` — keine `vimrc` laden,
- `-U NONE` — keine `gvimrc` laden (falls GUI),
- `--noplugin` — keine Plugins laden.

Das ergibt eine Art „Werkszustand“. Wenn VIM so sehr schnell startet, das normale Setup aber deutlich langsamer, liegt das Problem fast sicher in der eigenen Konfiguration.

Eigene Minimal-Konfiguration Für gezieltes Debugging ist eine „Minimal-vimrc“ sinnvoll, z. B.:

```
" minimal.vim
set nocompatible
```

```
syntax on
filetype plugin indent on
set number
```

Start mit:

```
vim -u minimal.vim -U NONE --noplugin --
startuptime minimal.log
```

Dadurch kann man schrittweise Optionen, Autocommands und Plugins hinzufügen und beobachten, ab *wann* der Start plötzlich deutlich langsamer wird.

Was verlangsamt den Start typischerweise?

Erfahrungsgemäß sind die größten Kandidaten:

- **Plugin-Ballungen** im `plugin/`-Verzeichnis, die alle sofort geladen werden, statt Lazy-Loading zu nutzen.
- **Schwere Statusleisten / File-Tree-Plugins**, die beim Start aufwändige Initialisierung, Dateiscans oder externe Kommandos ausführen.
- **Mehrfach geladene Konfiguration**, z. B. durch unklare `autocmd`-Definitionen ohne `:autocmd!` in `augroups`.
- **Externe Programme im Startpfad**, etwa `:!`-Aufrufe in der `vimrc`, die jedes Mal Shells, Interpreter oder Linter starten.

Ziel ist es, den Startpfad so aufzuräumen, dass beim Editorstart nur *Basisfunktionalität* und wirklich notwendige Plugins geladen werden; alles andere wird erst beim tatsächlichen Bedarf aktiviert (z. B. per Dateityp, Kommando oder Mapping).

Übungen

1. **Startzeit protokollieren** Erzeuge mit:

   ```
   vim --startuptime full.log
   ```

   einen Startzeit-Log deiner normalen Konfiguration. Öffne `full.log` in VIM und suche nach Zeilen mit deutlich höheren Millisekunden-Werten. Notiere dir mindestens zwei Dateien (Plugins oder Skripte), die überdurchschnittlich lange brauchen.
2. **Vergleich: ohne Konfiguration** Starte VIM mit:

```
vim -u NONE -U NONE --noplugin --
startuptime naked.log
```

und vergleiche `naked.log` mit `full.log`. Bestimme grob:
   - Wie groß ist der Unterschied in der Gesamtzeit?
   - Welche Skripte sind nur in `full.log` zu sehen?

3. **Minimal-vimrc testen** Lege eine Datei `minimal.vim` an, die nur wenige, unverfängliche Optionen setzt (z. B. `nocompatible`, `number`, `syntax on`). Starte:

```
vim -u minimal.vim --noplugin --
startuptime minimal.log
```

und vergleiche die Startzeit mit deiner normalen Umgebung. Ziel: ein Gefühl dafür entwickeln, wie viel „Overhead“ tatsächlich aus deiner gewachsenen Konfiguration stammt.

13.8.2. Konfigurationsfehler finden

Viele merkwürdige Effekte in VIM sind am Ende keine „Bugs“, sondern Konfigurationsfehler: doppelt gesetzte Optionen, kollidierende Mappings, Plugins, die unerwartet Optionen überschreiben, oder Blöcke, die auf einer Plattform gar nicht existieren dürften. Ziel dieses Abschnitts ist es, systematisch herauszufinden, *woher* eine bestimmte Einstellung oder ein bestimmtes Verhalten kommt.

`:messages`: Fehlermeldungen nachlesen

VIM zeigt Fehlermeldungen oft nur kurz in der Statuszeile an. Mit:

- `:messages`

lässt sich der gesamte Meldungspuffer ansehen. Typische Kandidaten:

- `E492: Not an editor command: …` → Tippfehler im Befehl oder Plugin/Mappings, die nicht geladen wurden.
- `E15: Invalid expression` → fehlerhafte `if`-Bedingung, `has('…')`-Abfrage oder ungültige Option.
- `E518: Unknown option: …` → Option existiert in dieser VIM/Neovim-Version oder auf dieser Plattform nicht.

`:messages` ist der erste Blick, wenn beim Start kurz eine rote Meldung auftaucht, die man nicht vollständig lesen konnte.

`:verbose`: Wer setzt diese Option?

Wenn man wissen will, *woher* eine bestimmte Option ihren Wert hat, hilft das `verbose`-Flag:

- `:verbose set number?`

VIM zeigt dann nicht nur den aktuellen Wert von `number`, sondern auch, *welche* Datei ihn zuletzt gesetzt hat, z. B.:

```
number
  Last set from ~/.vim/after/ftplugin/python.vim
```

Das funktioniert mit allen Optionen, etwa:

- `:verbose set tabstop?`
- `:verbose set background?`
- `:verbose set expandtab?`

So lassen sich Konflikte eingrenzen:

- Globale Defaults vs. ftplugin-Einstellungen
- vimrc vs. `after/`-Verzeichnis
- eigene Konfiguration vs. Plugin-Voreinstellungen

`:verbose map`: Woher kommen meine Mappings?

Mit `:verbose map` und Varianten kann man nachvollziehen, welches Skript ein Mapping gesetzt hat:

- `:verbose nnoremap <leader>f`

zeigt das konkrete Mapping und die Datei, aus der es stammt. Weitere nützliche Varianten:

- `:verbose map` — alle rekursiven Mappings
- `:verbose noremap` — alle nicht rekursiven Mappings
- `:verbose imap` — Insert-Mode-Mappings
- `:verbose vnoremap` — Visual-Mode-Mappings

Damit findet man z. B. heraus:

- welches Plugin eine Standard-Tastenkombination „geklaut“ hat,
- warum ein eigenes Mapping nicht greift (überschrieben),
- welche `ftplugin`-Datei dateitypspezifische Mappings gesetzt hat.

`:scriptnames`: Was wurde überhaupt geladen?

Mit:

- `:scriptnames`

zeigt VIM eine nummerierte Liste aller bisher geladenen Skripte:

- vimrc / gvimrc / init.vim
- `plugin/`-Dateien
- `ftplugin/`, `indent/`, `syntax/`
- `after/`-Verzeichnisse

Typische Nutzung:

- Prüfen, ob eine bestimmte Datei *überhaupt* geladen wurde.
- Nachsehen, ob eine `after/`-Variante vorhanden ist, die Optionen überschreibt.
- Kontrolle, ob Plugins doppelt oder aus unerwarteten Pfaden geladen werden.

In Kombination mit `:verbose set` und `:verbose map` kann man so oft genau nachvollziehen, *welche* Datei für eine Einstellungsänderung verantwortlich ist.

`:echo has('...')`: Feature- und Plattform-Checks

Manche Konfigurationen sollen nur aktiv sein, wenn bestimmte Features oder Umgebungen vorhanden sind. Statt „blind“ Befehle auszuführen, kann man:

- `:echo has('nvim')`
- `:echo has('clipboard')`
- `:echo has('gui_running')`

verwenden. Die Ausgabe ist:

- 1 — Feature vorhanden
- 0 — Feature nicht vorhanden

In der vimrc nutzt man das typischerweise so:

```
if has('clipboard')
  set clipboard=unnamedplus
endif
```

oder für Neovim-spezifische Blöcke:

```
if has('nvim')
  " Neovim-spezifische Einstellungen
endif
```

Damit vermeidet man Fehlermeldungen wie „Unknown option“ auf falschen Plattformen oder in älteren Versionen.

Schrittweises Auskommentieren und optionale Blöcke

Wenn ein Fehler schwer zu lokalisieren ist, hilft oft nur:

- große Blöcke in der `vimrc` auszukommentieren,
- die Datei neu zu laden (`:source \$MYVIMRC`),
- und zu prüfen, ob der Fehler verschwindet.

Bewährte Techniken:

- **Binary Search**: jeweils die Hälfte der vimrc aktiv lassen, bis der problematische Block identifiziert ist.
- **Optionale Blöcke über** `:source`: statt alles in `vimrc` zu haben, z. B.:

  ```
  if filereadable(expand('~/.vim/config/ui.vim'))
    source ~/.vim/config/ui.vim
  endif
  ```

  Dann kann man testweise einzelne `source`-Zeilen auskommentieren, ohne große Blöcke zu löschen.

Wichtig ist, dass die vimrc *mehr* ist als ein Haufen Einzeiler: Klare Blöcke, sinnvolle Gruppierung und saubere Kommentare machen Debugging überhaupt erst praktikabel.

Übungen

1. **Optionen zurückverfolgen** Setze in deiner Konfiguration einige Optionen, z. B. `number`, `relativenumber`, `expandtab`. Starte VIM normal und führe für jede dieser Optionen aus:
   - `:verbose set number?`
   - `:verbose set relativenumber?`
   - `:verbose set expandtab?`

   Notiere dir, *aus welchen Dateien* die Werte stammen und ob dich eines der Ergebnisse überrascht (z. B. ftplugin statt vimrc).
2. **Mapping-Herkunft prüfen** Lege ein eigenes Mapping in deiner vimrc an, z. B.:

   ```
   nnoremap <leader>s :w<CR>
   ```

Starte VIM und überprüfe mit:

```
:verbose nnoremap <leader>s
```

ob das Mapping aus der erwarteten Datei stammt. Füge danach bewusst ein zweites (konfliktierendes) Mapping in einem anderen Skript ein und beobachte, welches „gewinnt“.

3. **Fehler künstlich erzeugen und finden** Füge in deine vimrc testweise eine ungültige Option ein, z. B.:

   ```
   set foobaroption
   ```

   Starte VIM und lies mit:

   ```
   :messages
   ```

   die Fehlermeldung nach. Entferne oder kommentiere die fehlerhafte Zeile und prüfe, dass beim nächsten Start keine entsprechende Meldung mehr auftaucht.

13.8.3. Saubere Fallbacks und Bedingungen

Eine robuste `vimrc` darf nicht voraussetzen, dass alle Features, Kommandos und Plugins auf jeder Maschine vorhanden sind. Stattdessen sollten Optionen immer *bedingungsgesteuert* gesetzt werden: nur dann, wenn das zugrunde liegende Feature tatsächlich existiert.

Zentrale Bausteine dafür sind:

- `has('feature')` – prüft, ob ein bestimmtes Feature vorhanden ist
- `exists(':Command')` – prüft, ob ein Ex-Kommando existiert
- `exists('*Function')` – prüft, ob eine Funktion definiert ist

Ziel ist, dieselbe Konfiguration auf verschiedenen VIM-/Neovim-Versionen und Plattformen verwendbar zu halten, ohne mit Fehlermeldungen zu starten.

Feature-Prüfungen mit `has()`

`has()` liefert 1 (wahr) oder 0 (falsch). Typische Beispiele:

```
if has('nvim')
  " Neovim-spezifische Einstellungen
endif
```

```
if has('clipboard')
  set clipboard=unnamedplus
endif

if has('termguicolors')
  set termguicolors
endif
```

Damit lässt sich sauber trennen zwischen:

- klassischem VIM vs. Neovim (`has('nvim')`)
- Versionen mit/ohne System-Clipboard (`has('clipboard')`)
- Terminals mit 24-Bit-Farbunterstützung (`has('termguicolors')`)

Plattformunterschiede abfedern

`has()` kann auch die Plattform erkennen:

- `has('unix')` – klassische Unix-Systeme (inkl. Linux, *BSD)
- `has('macunix')` – macOS
- `has('win32')` / `has('win64')` – Windows

Beispiel: unterschiedliche Standard-Shells:

```
if has('win32') || has('win64')
  set shell=powershell.exe
else
  set shell=/bin/bash
endif
```

Oder unterschiedliche Pfade/Tools:

```
if has('macunix')
  let g:grepprg = 'rg --vimgrep'
elseif has('unix')
  let g:grepprg = 'grep -Rn'
endif
```

So bleibt die `vimrc` portabel, ohne dass man separate Dateien pro Plattform pflegen muss.

Kommandos und Funktionen mit `exists()` testen

`exists()` prüft, ob bestimmte Vimscript-Elemente vorhanden sind:

- `exists(':Command')` – Ex-Kommando
- `exists('*FunctionName')` – Funktion
- `exists('g:variable')` – Variable

Beispiele:

```
" Nur dann Mapping setzen, wenn :LspDefinition existiert
if exists(':LspDefinition')
  nnoremap <leader>d :LspDefinition<CR>
endif

" Nur dann spezielle Statusline aktivieren, wenn Funktion existie
if exists('*MyStatusline')
  set statusline=%!MyStatusline()
endif
```

Damit vermeidet man Fehler wie:

- „Not an editor command: LspDefinition“ auf Systemen ohne LSP-Plugin
- „E117: Unknown function: MyStatusline“ wenn ein Skript nicht geladen wurde

Saubere Fallbacks statt harter Annahmen

Statt „blind“ Optionen zu setzen, empfiehlt es sich, Fallbacks explizit zu definieren:

```
if has('termguicolors')
  set termguicolors
  colorscheme gruvbox-material
else
  set t_Co=256
  colorscheme desert
endif
```

Hier wird:

- auf modernen Terminals ein 24-Bit-Schema gewählt,
- auf alten Terminals ein 256-Farben-Setup verwendet.

Ähnlich beim Clipboard:

```
if has('clipboard')
  set clipboard=unnamedplus
else
  " Fallback: explizite Yank-/Paste-Befehle nutzen
endif
```

Die Idee: *Immer* definieren, was passieren soll, wenn ein Feature *nicht* verfügbar ist – statt stillschweigend davon auszugehen, dass es überall vorhanden ist.

Übungen

1. **Plattformabhängige Shell-Konfiguration** Erweitere deine `vimrc` um einen Block, der je nach Plattform unterschiedliche Shells setzt, z. B.:
   - `/bin/bash` oder `/usr/bin/zsh` auf Unix/macOS
   - `powershell.exe` auf Windows

   Teste die Konfiguration auf mindestens zwei unterschiedlichen Systemen (z. B. Linux-VM und Windows) oder simuliere sie, indem du den Block temporär mit `if 0` / `if 1` umschaltest.
2. **Clipboard-Feature sauber behandeln** Ergänze in deiner `vimrc` einen Block:

   ```
   if has('clipboard')
     set clipboard=unnamedplus
   else
     " Fallback-Kommentar: Wie gehst du ohne System-
   Clipboard vor?
   endif
   ```

   Starte VIM und prüfe mit:

   ```
   :echo has('clipboard')
   ```

   ob dein System das Feature wirklich besitzt. Dokumentiere im Fallback-Kommentar, wie du in diesem Fall mit Yanks/Copy arbeitest (z. B. „nur intern in VIM“).
3. **Plugin-abhängiges Mapping mit Fallback** Wähle ein Plugin-Kommando, das nicht überall verfügbar ist (z. B. `:Git` eines Git-Plugins oder ein LSP-Befehl). Erzeuge einen Block der Form:

```
if exists(':Git')
  nnoremap <leader>gs :Git status<CR>
else
  nnoremap <leader>gs :!git status<CR>
endif
```

Ziel: Derselbe Shortcut <leader>gs funktioniert sowohl mit als auch ohne Plugin – nur der „Weg“ zum Kommando ändert sich.

13.9. Beispiel-vimrc

13.9.1. Minimalbeispiel

Eine *minimal sinnvolle* `vimrc` soll zwei Dinge leisten:

1. VIM aus dem „kompatibel zu `vi`“-Modus holen.
2. Einige zentrale Standards setzen (Encoding, Suche, Einrückung, Anzeige), ohne sich gleich in hunderten Optionen zu verlieren.

Ein kompaktes, alltagstaugliches Minimalbeispiel könnte so aussehen:

```
" ------------------------------
" Basis: Moderne VIM-Grundlagen
" ------------------------------
set nocompatible
set encoding=utf-8
set fileencodings=utf-8,latin1

" ------------------------------
" Anzeige & Navigation
" ------------------------------
set number
set ruler
set showcmd
set wildmenu

" ------------------------------
" Suche
" ------------------------------
```

```
set ignorecase
set smartcase
set incsearch
set hlsearch

" ------------------------------
" Einrückung & Tabs
" ------------------------------
set expandtab
set shiftwidth=4
set tabstop=4
set smartindent

" ------------------------------
" Syntax & Dateitypen
" ------------------------------
syntax on
filetype plugin indent on
```

Erläuterungen zum Minimalbeispiel

`set nocompatible` Schaltet veraltete `vi`-Kompatibilität ab und aktiviert das moderne VIM-Verhalten (empfohlen in praktisch allen Setups).

`encoding` / `fileencodings` Stellt sicher, dass intern UTF-8 verwendet wird und beim Laden zuerst UTF-8 versucht wird. `latin1` dient als Fallback für ältere Dateien.

`number`, `ruler`, `showcmd`, `wildmenu`
- `number`: Zeilennummern in der linken Spalte.
- `ruler`: Cursorposition in der Statuszeile.
- `showcmd`: zeigt unvollständige Befehle unten an.
- `wildmenu`: komfortable Completion im Kommandozeilenmodus.

`ignorecase`, `smartcase`, `incsearch`, `hlsearch`
- `ignorecase`: Suche ignoriert Groß-/Kleinschreibung.
- `smartcase`: hebt `ignorecase` automatisch auf, sobald der Suchbegriff Großbuchstaben enthält.
- `incsearch`: zeigt während der Eingabe schon Treffer an.
- `hlsearch`: markiert alle Treffer im Text.

`expandtab`, `shiftwidth`, `tabstop`, `smartindent` •
`expandtab`: Tabs werden zu Spaces.
- `shiftwidth=4`: Einrückbreite für [<] [>], [<Tab>] etc.
- `tabstop=4`: Anzeige- und Einfügebreite von Tabs.
- `smartindent`: einfache automatische Einrückung (für viele Sprachen ausreichend).

`syntax on`, `filetype plugin indent on` Aktiviert Syntax-Highlighting sowie das Dateityp-System mit Plugins und Einrückungsregeln. Ohne diese Zeilen bleibt VIM deutlich „stummer“ und verzichtet auf viele Komfortfunktionen.

Übungen

1. **Eigene Minimal-`vimrc` anlegen** Erzeuge im Home-Verzeichnis eine Datei `/~.vimrc` oder `$HOME/vimrc` (je nach Plattform) und übertrage das oben gezeigte Minimalbeispiel. Starte VIM neu und prüfe mit:
   - `:set number?`
   - `:set ignorecase?`

   ob die Optionen wie erwartet aktiv sind.
2. **Tab-/Einrückungsprofil anpassen** Ändere im Minimalbeispiel:
   - `shiftwidth=2`
   - `tabstop=2`

   Öffne eine Datei, rücke einige Zeilen mit [<] und [>] ein und beobachte den Unterschied. Ziel: Ein Gefühl dafür bekommen, wie stark Einrückungsoptionen das Schreibgefühl beeinflussen.
3. **Sucheinstellungen bewusst testen** Nutze in einer Testdatei verschiedene Suchbegriffe:
   - [/] gefolgt von `foo`
   - [/] gefolgt von `Foo`

   und beobachte, wie sich `ignorecase` und `smartcase` auswirken. Schalte danach testweise `:set noignorecase` und `:set nosmartcase` und wiederhole dieselben Suchen, um den Unterschied klar zu sehen.

13.9.2. Strukturierte Arbeits-vimrc

Für den Alltag reicht eine Minimal-vimrc oft aus – langfristig wird eine klar strukturierte, gut kommentierte Arbeits-vimrc aber deutlich angenehmer zu pflegen. Ziel

ist eine Datei, die du auch nach Monaten noch verstehst, gefahrlos ändern kannst und in logische Blöcke unterteilt ist.

Ein mögliches Gerüst könnte so aussehen:

```
" ============================================================
"  Arbeits-vimrc – strukturiert und erweiterbar
" ============================================================

" 1) Grundmodus & Basis
set nocompatible
set encoding=utf-8
set fileencodings=utf-8,latin1
set backspace=indent,eol,start
set hidden
set undofile
set undodir=~/.vim/undo

" 2) Anzeige & Oberfläche
set number
set relativenumber
set cursorline
set showcmd
set showmode
set ruler
set list
set listchars=tab:»· ,trail:· ,extends:…,precedes:…

syntax on
set termguicolors
colorscheme desert

" 3) Suche & Navigation
set ignorecase
set smartcase
set incsearch
set hlsearch
set wildmenu
```

```
set wildmode=longest:full,full

" 4) Einrückung & Tabs
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab
set autoindent
set smartindent

" 5) Dateien, Backup & Swap
set backup
set backupdir=~/.vim/backup//
set writebackup
set swapfile
set directory=~/.vim/swap//

" 6) Leader & Grund-Mappings
let mapleader = " "

nnoremap <leader>w  :w<CR>
nnoremap <leader>q  :q<CR>
nnoremap <leader>e  :edit ~/.vimrc<CR>
nnoremap <leader>sv :source ~/.vimrc<CR>

" Fenster-Navigation
nnoremap <leader>h <C-w>h
nnoremap <leader>j <C-w>j
nnoremap <leader>k <C-w>k
nnoremap <leader>l <C-w>l

" 7) Autocommands & augroups
augroup trim_trailing_whitespace
  autocmd!
  autocmd BufWritePre * %s/\s\+$//e
augroup END
```

```
" 8) Dateityp-System
filetype plugin indent on

" 9) Optionale Plugin-Struktur (nur vorbereitet)
" source ~/.vim/plugins.vim
" source ~/.vim/plugin-config.vim
```

Wichtige Strukturgedanken dahinter:

- **Klare Blöcke statt Einzeiler-Chaos** Jeder Abschnitt ist deutlich kommentiert und kapselt ein Thema: Basis, Anzeige, Suche, Einrückung, Dateien, Mappings, Autocommands, Dateitypen, Plugins.
- **Leader-Mappings zentral** `mapleader` steht weit oben, alle `<leader>`-Mappings sind gebündelt. Das hilft, Kollisionen und Doppelbelegungen zu erkennen.
- **Autocommands immer in augroups** Durch `augroup` + `autocmd!` vermeidest du doppelte Autocommands beim Neuladen der vimrc.
- **Pfadwahl für Backup/Undo/Swap bewusst** Eigene Verzeichnisse für Backup-, Undo- und Swap-Dateien halten Projektordner sauber.
- **Plugin-Konfiguration ausgelagert** Die eigentliche vimrc bleibt übersichtlich; Details wandern später in Dateien wie `plugins.vim` oder `plugin-config.vim`.

Übungen

1. Übertrage das obige Gerüst in deine eigene vimrc und passe die Kommentarblöcke an deine Themen an (z. B. „LaTeX“, „Python“, „Git“).
2. Ergänze mindestens zwei eigene Leader-Mappings:
   - eines für das Öffnen deines „Hauptprojekts“,
   - eines für ein häufig genutztes Kommando (z. B. `:make` oder `:grep`).

   Teste sie im Alltag und ändere sie, bis sie sich natürlich anfühlen.
3. Lege die Verzeichnisse für `backup`, `swap` und `undo` tatsächlich an (z. B. `/~.vim/backup`, `/~.vim/swap`, `/~.vim/undo`) und prüfe mit neuen Dateien, ob dort Dateien angelegt werden.

13.9.3. Übungen zur eigenen `vimrc`

Die vorherigen Abschnitte haben einzelne Optionen, Blöcke und Beispiele vorgestellt. In diesem Abschnitt geht es darum, aus dem Beispiel eine *eigene*, belastbare Arbeits-`vimrc` zu bauen – Schritt für Schritt und mit bewusstem Testen.

Übung 1: Sicheres Experimentier-Setup

1. Erzeuge eine Sicherung deiner aktuellen `vimrc`:
   - `cp ~/.vimrc ~/.vimrc.backup` (oder entsprechender Pfad)
2. Lege ein kleines Git-Repository nur für deine Konfiguration an (optional, aber empfohlen):

   ```
   cd ~/.vim
   git init
   git add vimrc
   git commit -m "Ausgangszustand der vimrc"
   ```

3. Vereinbare mit dir selbst eine Regel: Änderungen an der `vimrc` nur noch in kleinen Schritten, mit klaren Kommentaren und ggf. Git-Commit nach jedem funktionierenden Zustand.

Ziel: Du kannst jederzeit auf einen funktionierenden Stand zurückspringen und musst keine Angst vor Experimenten haben.

Übung 2: Optionen bewusst ein- und ausschalten

1. Nimm dir den Block „Anzeige und Benutzeroberfläche“ aus deiner `vimrc` vor und kommentiere alle Zeilen mit `number`, `relativenumber`, `cursorline`, `hlsearch`, `incsearch` aus.
2. Aktiviere nun *nur* eine Option wieder, z. B.:
   - `:set number`
   - später `:set relativenumber`

   Arbeite jeweils ein paar Minuten damit und entscheide bewusst, ob du die Option dauerhaft behalten willst.
3. Wiederhole das mit:
   - `:set hlsearch` / `:set nohlsearch`
   - `:set incsearch` / `:set noincsearch`
   - `:set list` / `:set nolist`

Ziel: Du kennst den Effekt jeder Option aus eigener Erfahrung – nicht nur aus Beschreibungen.

Übung 3: Eigene Leader-Mappings definieren

1. Lege deinen <leader> fest, falls noch nicht geschehen, z. B.:

   ```
   let mapleader = " "
   ```

2. Definiere mindestens drei eigene Normal-Mode-Mappings:
   - <leader>w für `:w`
   - <leader>q für `:q`
   - ein drittes Mapping für eine Aktion, die du oft brauchst (z. B. `:make`, `:grep` oder das Öffnen deiner Notizen).
3. Arbeite ein paar Tage damit und passe die Belegung an, wenn du merkst, dass eine andere Taste intuitiver ist.

Ziel: Dein <leader>-Bereich bildet deinen persönlichen Werkzeugkasten ab, nicht irgendwelche Beispielbelegungen.

Übung 4: Autocommands und `ftplugin`-Dateien einsetzen

1. Erzeuge eine kleine `augroup` in deiner `vimrc`, z. B.:

   ```
   augroup trim_trailing_whitespace
     autocmd!
     autocmd BufWritePre * %s/\s\+$//e
   augroup END
   ```

   Teste das Verhalten beim Speichern verschiedener Dateien.
2. Lege für `markdown` oder `tex` eine eigene `ftplugin`-Datei an, z. B.:

   ```
   " ~/.vim/ftplugin/markdown.vim
   setlocal textwidth=80
   setlocal spell
   ```

   Öffne eine Markdown-Datei und prüfe mit `:setlocal`, ob die Optionen wirklich nur dort gelten.

Ziel: Du verlagerst Dateityp-spezifische Einstellungen aus der zentralen `vimrc` in saubere, lokale Konfigurationsbausteine.

Übung 5: Eigene Arbeits-`vimrc` dokumentieren

1. Gehe Abschnitt für Abschnitt durch deine `vimrc` und ergänze überall kurze Kommentare:
   - „Warum gibt es diesen Block?“
   - „Was ist die beabsichtigte Wirkung?“
2. Schreibe am Anfang der `vimrc` einen kurzen Header mit:
   - Datum / Version,
   - Hinweis auf Backup/Repository,
   - kurzer Beschreibung deines Setups (z. B. „Fokus auf Python/LaTeX“).

Ziel: Deine `vimrc` ist nicht nur funktional, sondern auch in einem Jahr noch verständlich – für dich selbst und für andere.

Schlussbemerkung

VIM lernst du nicht auswendig, VIM lernst du, indem du es benutzt. Dieses Handbuch will dir dabei helfen – nicht mit Zaubertricks, sondern mit klaren Konzepten, nachvollziehbaren Beispielen und wiederholbarer Praxis.

VIM bleibt am Anfang oft sperrig. Die eigentliche Stärke zeigt sich erst, wenn du bereit bist, ein Stück Arbeit hinein zu stecken: Operatoren verstehen, Bewegungen verinnerlichen, Textobjekte bewusst einsetzen, Makros ausprobieren, Register nutzen. Mit jeder bewussten Übung wird aus einem angeblich »komischen Editor« ein Werkzeug, das dir Arbeit abnimmt, statt sie zu erzeugen.

In diesem Sinne ist VIM ein Stück »Geist von Unix«: kleine, scharfe Bausteine mit klarer Aufgabe, die du frei kombinieren kannst. Dieses Handbuch liefert dir dafür einen strukturierten Rahmen. Es ersetzt deine eigene Praxis nicht, sondern soll dir ermöglichen, gezielt zu experimentieren, Fehler zu analysieren und funktionierende Arbeitsweisen zu festigen.

Das Dokument ist bewusst nicht als abgeschlossene »Endfassung« gedacht, sondern als lebendige Sammlung von Werkzeugen, Konzepten und Beispielen, die mit deinen Fähigkeiten mitwachsen kann. Wenn du beim Lesen auf Stellen stößt, die unklar sind oder sich in der Praxis als unhandlich erweisen, dann ist das kein Defekt, sondern ein Anlass, sie zu schärfen – im Text und in deinem eigenen Umgang mit VIM.

Wenn du bereit bist, VIM wirklich zu benutzen, statt nur darüber zu lesen, wird sich das, was hier steht, Schritt für Schritt von der Theorie in deine Finger verlagern. Genau dort gehört es hin.

Index

Tasten
- ', 136, 138
- (, 2, 6–10, 12, 19, 20, 44, 55–60, 63, 64, 67, 68, 70, 71, 73, 76, 80, 83, 85–87, 90, 91, 93–97, 101–104, 106, 107, 120, 122, 123, 131, 169, 185
-), 9, 10, 19, 44, 45, 57, 64, 120
- *, 26, 27, 176, 179
- +, 16, 17, 26, 27, 36, 77, 78, 80, 175–179, 181, 238, 246, 247, 249, 250, 262
- ,, 343, 353
- -, 16, 17, 238, 246, 247, 249, 250, 262, 280
- ., 27, 65–67, 75, 86–92, 99–101, 110, 112, 115, 122, 123, 127, 131, 133, 138, 168, 169, 353
- /, 5, 15, 65–68, 78–80, 87, 109–112, 125, 131, 138, 162, 169, 176, 183, 259, 263, 336, 337, 339, 393
- :, 5, 7, 29, 106, 176, 265, 344
- ;, 29, 121, 146, 169, 170, 172, 173
- <, 9, 11, 102, 103, 106, 107, 119, 121, 122, 124, 126, 129, 132, 217, 238, 246, 247, 249, 251, 262, 393
- <BS>, 82, 83, 333, 336
- <Down>, 183
- <Esc>, 69
- <Space>, 353, 354
- <Tab>, 253, 333, 335
- <Up>, 183
- =, 96, 106, 126, 132, 221, 237, 238, 249–251, 262
- >, 9, 11, 119, 121, 124, 126, 127, 129, 132, 133, 216, 217, 238, 246, 247, 249, 251, 262, 393
- ?, 5, 65, 68, 78–80, 109–112, 336, 337, 339
- [, 1, 19, 24, 57, 58, 94, 101–104, 106, 107, 183, 228
- #, 176

$, 6–11, 14, 15, 19, 21, 36, 48, 52, 54, 55, 61, 65, 68, 69, 74, 79, 83, 85, 87, 128, 131, 138, 171, 183, 210
%, 9, 19, 20, 55–57, 59, 60, 73, 79, 87, 176, 183, 186
^, 14, 15, 21, 183, 184, 186
_, 176, 177, 179–181, 247–250
\, 260–263, 291, 292
<Down>, 171
<Up>, 171
<leader>, 347–352
<Tab>, 393
↑, 109, 112
{, 9–11, 19, 20, 37, 45, 46, 55, 57, 58, 60, 63, 67, 68, 70, 76, 79, 80, 93–96, 101–107, 120, 122, 188, 189, 197, 200, 201
}, 9, 10, 19, 20, 45, 46, 57, 61, 62, 64, 65, 67, 68, 106, 120, 200, 201
], 19, 57, 183, 228
0, 4, 7, 10, 14, 15, 21, 25, 26, 36, 48, 52, 61, 69, 74, 78, 80, 128, 129, 131, 157, 167, 169–178, 180, 183, 184, 186, 204, 210, 246–248
1, 4, 10, 19, 25, 26, 48, 169, 171–173, 176, 178–180, 204, 246–248
2, 8, 10, 11, 13, 16, 17, 19, 26, 43, 45, 48, 53, 62–65, 67, 69, 70, 72, 75, 79, 88, 89, 91, 170, 174, 178, 188, 198, 199
3, xix, 8, 10, 11, 13, 15, 16, 19, 21, 25, 45, 53, 62, 64–68, 70–73, 81, 82, 84, 87, 88, 90, 91, 121, 178, 188, 198, 212, 219, 246
4, 16, 48, 82, 247, 248
5, 10, 13, 17, 53, 64, 67, 68, 71, 87, 90, 91, 157, 170–173, 220, 246, 247
6, 276, 278, 279, 293–295
7, 10, 82
8, 246
9, 167, 176, 178–180
A, 2, 23–25, 27–30, 35, 36, 48, 52, 89, 130–133, 146, 168–173, 176–178, 181
a, 1, 2, 7–12, 19–21, 23–25, 28–30, 32, 55, 56, 58, 60, 63, 64, 67, 68, 70, 71, 76–80, 86, 93–107, 122, 125, 137–139, 145, 167–170, 172, 174–186, 188–190, 192, 197–199, 203, 204, 211, 212, 219–221, 259–261
B, 16, 41–43, 176
b, 7, 9, 10, 15–18, 21, 41–43, 47, 49–55, 62, 78, 79, 87, 120, 121, 128, 132, 137–139, 145, 168, 174–177, 180, 182–186, 189, 190, 239, 263, 345, 348
Backspace, 24, 25
Break, 175

C, 199–201, 203
c, 2, 3, 5–9, 11–15, 18–21, 27, 28, 30, 33, 41–46, 56, 58, 59, 68–75, 80, 81, 83–107, 111, 119, 121–124, 126, 128, 129, 132, 167, 169, 170, 175, 179–182, 184, 185, 189, 190, 192, 194, 195, 197–201, 203, 205, 210, 212, 214–216, 220, 221, 223, 225, 227, 228, 245, 260, 262, 264
Ctrl, xix, 1, 3–5, 17, 18, 24–27, 29, 34–39, 47–51, 53–55, 87, 127–133, 136, 138, 153, 157, 160–162, 175, 183, 186, 188, 216, 231–251, 255, 256, 259–264, 276, 278, 279, 291–295, 342, 343, 352
D, 219, 220
d, xix, 2–4, 6–15, 17–21, 36–39, 41–51, 53–58, 61–69, 73–75, 77, 78, 86–88, 90, 91, 93–107, 109–112, 119–124, 126, 128–130, 132, 147, 162, 169, 176–182, 185, 186, 188, 210, 217, 219, 220, 229, 263, 264, 344, 348
E, 16, 41–44, 219, 220, 229
e, 6, 9–11, 15, 16, 21, 38, 39, 41–43, 61, 62, 68, 69, 72, 74, 83, 87, 120, 121, 128, 132, 350
Enter, 5, 6, 65–68, 78–80, 87, 102, 103, 109–112, 125, 131, 150–152, 164, 169, 170, 183, 228, 260, 263, 280, 286, 292, 344, 353
Esc, xix, 1–5, 11, 24–29, 31, 32, 34, 52, 65, 73, 82, 84, 87–90, 95, 99, 101, 119–124, 128–131, 156, 167, 168, 170–173, 346
F, 120, 132, 183
f, 2, 17, 18, 29, 38, 47, 49–51, 53, 54, 59, 87, 107, 120, 121, 128, 132, 133, 157, 169, 175, 183, 184, 186–190, 197, 199, 204, 211, 212, 215, 219, 220, 229, 263, 347, 348
G, 9, 18, 19, 48–51, 87, 125, 157, 199–201, 221, 235, 237, 253, 263
g, 8, 9, 11, 12, 18, 19, 48–51, 84, 85, 125, 132, 133, 157, 199–201, 237, 239, 253, 262, 304, 305, 346, 348
H, 46, 47, 49, 51, 52
h, 1, 3, 12–14, 21, 24, 25, 35, 36, 59, 73, 84, 119, 120, 124, 128, 132, 160, 183–186, 233, 234, 238–245, 255, 256, 260–263, 278, 339, 345, 348
I, 2, 4, 23, 24, 128, 130–133, 173
i, 1–3, 6–12, 19, 20, 23, 24, 26–29, 32, 55, 56, 58–60, 63–65, 67, 68, 70, 71, 73, 75–80, 83–107, 120, 122, 123, 125, 131, 133, 167, 169, 170, 180, 182–186, 259–261, 280, 292
j, 1, 3, 4, 10, 13–18, 21, 24, 53–55, 62, 63, 67, 69, 73, 75, 79, 84, 85, 87–90, 111, 119–128, 132, 150, 160, 171–173, 175, 186, 188, 189, 199, 203, 204, 210, 212, 219, 220, 233, 234, 238–245, 256, 260–262, 292, 313, 345, 346, 348

k, 1, 3, 4, 10, 13, 14, 16–18, 21, 24, 53–55, 67, 84, 85, 111, 119–121, 124, 125, 128, 132, 150, 160, 186, 199, 203, 210, 233, 234, 238–245, 256, 260–262, 292, 313, 345, 348
L, 46, 47, 51, 52
l, 1, 3, 13, 14, 21, 38, 59, 67, 73, 84, 119, 120, 124, 128, 132, 160, 183–186, 233, 234, 238–245, 255, 256, 260–263, 278, 345, 348
M, 46, 47, 192, 194, 195, 197, 201–205, 207, 208, 210, 211, 214–217, 221, 223, 225, 227, 228
m, 136–139, 145, 189, 190, 201, 202, 204–208, 350, 352
N, 5, 109–112, 259, 261, 263, 339
n, 5, 16, 17, 37–39, 53, 54, 70, 71, 87, 109–112, 231, 232, 260–263, 291, 292, 339, 350
O, 2, 3, 23, 24, 30–33, 88, 199–201, 203, 221, 223, 225, 227, 228
o, 1–3, 23, 24, 30–33, 37–39, 88, 119, 120, 132, 144, 189, 190, 192, 194, 195, 197–201, 203, 212, 220, 221, 223, 225, 227, 228, 260, 261
P, 66, 74–79
p, 4, 7–9, 11, 12, 26, 27, 38, 39, 46, 63, 64, 66–68, 70, 71, 73–80, 83, 85, 86, 90, 91, 93–97, 100, 101, 120, 122, 124–126, 131, 133, 136, 176–184, 186, 188, 197, 199, 211, 212, 219–221, 232, 233, 241, 242, 244, 256, 259, 261, 349, 350
Q, 7, 156, 157, 286, 287, 350
q, 21, 24, 25, 28, 54, 59, 67, 70, 91, 92, 132, 133, 160, 161, 167–177, 181, 184–186, 204, 347, 349, 350
R, 4, 5, 33–35, 80, 82–86, 192, 194, 195, 197, 201–205, 208, 210, 211, 214, 217, 225, 227, 280
r, 4, 5, 26, 27, 29, 33, 34, 36, 39, 80–82, 85–87, 123, 129, 132, 133, 136, 138, 200–202, 204–208, 342, 343
S, 70
s, 2, 99–101, 181, 183, 185, 186, 231–234, 343
T, 120, 132, 183, 304, 305
t, 29, 38, 39, 47, 52–55, 87, 107, 120, 121, 128, 132, 183, 186, 239, 304, 305, 347, 352
Tab, 25, 26, 275–277
U, 8, 9, 11, 12
u, 9, 11, 15, 17, 18, 24, 25, 29, 34–36, 39, 49–51, 53–55, 62, 63, 75, 88, 103, 106, 115, 126, 128, 136–140, 143, 144, 146, 147, 152–155, 158, 159, 172, 174, 186, 228, 232, 280, 342, 343
V, 3, 4, 87, 122, 124–127, 132, 133, 150–153, 158, 159, 216, 220

v, xix, 3, 4, 25–27, 56, 87, 93, 94, 96–104, 106, 119–125, 127–133, 153, 188, 216, 232, 233
W, 15, 16, 41–45, 52, 67, 68, 98, 99, 101, 239, 241
w, xix, 2, 3, 6–12, 15, 16, 21, 24, 25, 27–29, 33, 35–37, 39, 41–45, 52, 54, 55, 61–65, 67–80, 83–99, 101, 120–122, 128, 131–133, 160, 161, 167, 169–186, 231–251, 255, 256, 259–264, 278, 345, 348–350, 352
X, 29, 33, 107, 120, 123, 128, 129, 169, 175, 183, 184, 186, 211, 212, 216–218, 221, 223, 225
x, 24, 36, 38, 67, 74, 80–82, 86, 87, 89, 91, 136, 197, 211–216
y, 3, 6–9, 11–14, 18, 20, 21, 26, 27, 38, 39, 41, 43–47, 55–58, 74–80, 91, 93–96, 98, 99, 102–104, 106, 111, 119, 121, 124, 126, 129, 132, 169, 176–182, 185, 188, 210, 232, 233, 259, 261, 348
Z, 156, 157, 176, 177, 286, 287
z, 47, 48, 51–55, 87, 167, 174, 176, 184, 187–190, 192, 194, 195, 197–208, 210–221, 223, 225, 227–229, 239
Tastenkombinationen
Ctrl-6, 276, 278, 279, 293–295
Ctrl-[, 1, 24
Ctrl-\, 260–263, 291, 292
Ctrl-b, 17, 18, 47, 49–51, 53–55, 87, 183, 186, 263
Ctrl-Break, 175
Ctrl-c, 5, 175, 264
Ctrl-d, 17, 18, 38, 39, 48–51, 53–55, 162, 186, 263, 264
Ctrl-e, 38, 39
Ctrl-f, 17, 18, 38, 47, 49–51, 53, 54, 87, 157, 183, 186, 263
Ctrl-h, 24, 25, 35, 36
Ctrl-l, 38
Ctrl-n, 37–39, 260–263, 291, 292
Ctrl-o, 37–39
Ctrl-p, 38, 39
Ctrl-q, 25
Ctrl-r, 26, 27, 29, 34, 36, 39, 136, 138, 342, 343
Ctrl-t, 38, 39
Ctrl-u, 17, 18, 24, 25, 29, 35, 36, 39, 49–51, 53–55, 186
Ctrl-v, xix, 3, 4, 25, 26, 87, 127–133, 153, 188, 216
Ctrl-w, 24, 25, 29, 35, 36, 39, 160, 161, 231–251, 255, 256, 259–264, 278, 352
Ctrl-x, 38

Ctrl-y, 38, 39