UNIX
Tipps und Tricks

Version 1.24 — 17.12.2014

OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
E-Mail: tb@ostc.de
Web: www.ostc.de

Inhaltsverzeichnis

1 Einleitung

Die folgenden Abschnitte stellen eine Zusammenfassung kleinerer und größerer Tipps und Tricks dar, die sich beim Arbeiten mit UNIX als nützlich herausgestellt haben, weil sie:

- Die Arbeit erleichtern
- Zeit sparen
- Fehler vermeiden helfen

Eine gründliche und vollständige Einführung in Kommandos wie `vi`, `sh`, `sed`, `awk`, ... ist nicht beabsichtigt, Grundkenntnisse darin werden einfach vorausgesetzt.

1.1 Konventionen

Folgende Konventionen werden im weiteren verwendet:

- Kommentare zu einem Kommando werden in der gleichen Zeile durch `#` vom eigentlichen Kommando abgetrennt (das Zeichen `#` leitet in vielen UNIX-Tools einen Kommentar ein).
- Vom Benutzer einzusetzende Kommandos, Dateinamen oder Parameter werden `GROSS` geschrieben.
- Einige wenige der genannten Programme sind nicht auf jedem Rechner verfügbar (ausprobieren!), z.B. `cflow`, `indent`, ..., da eventuell mit dem dort installierten Betriebssystem keine C-Entwicklungsumgebung mehr mitgeliefert wird.
- Anstelle eines Rechnernamens wird in den Beispielen `HOST` verwendet, bitte beim Ausprobieren immer den konkreten Rechnernamen einsetzen.

1.2 Allgemeines

- UNIX ist ein offenes System, in dem *kooperativ* gearbeitet wird. D.h. wenn es nicht unbedingt notwendig ist (private Dateien), wird anderen der Zugang zu den eigenen Verzeichnissen und Dateien nicht verwehrt.
- **Verzeichnisse** (insbesondere das `HOME`-Verzeichnis) sollten nicht zu viele Dateien enthalten (Faustregel: eine Bildschirmseite). Datei- und Verzeichnisnamen sollten weder einbuchstabig noch sehr lang sein.
- Das `HOME`-Verzeichnis sollte frei von Arbeitsdateien sein, d.h. nur Unterverzeichnisse (und Konfigurations-Dateien) enthalten.

- Für **temporäre Dateien**, die ohne Überprüfen wieder gelöscht werden können, entweder eine eindeutige Extension (`.tmp`) oder ein-, zwei- oder dreibuchstabige Namen der Form `x`, `yy`, `zzz`, `...` verwenden.

  Eine weitere Alternative ist das Ablegen im Verzeichnis `/tmp` und die Verwendung der Prozeßnummer `$$` als Bestandteil der Dateinamen in **Shell-Skripten**.

- Sogenannte **Profile-, Konfigurations-** oder **Initialisierungs-Dateien** bzw. **-Verzeichnisse** kann man **verstecken**, indem sie einen `.` als erstes Zeichen erhalten. Diese Dateien werden von `ls` nicht angezeigt (außer die Option `-a` (all) oder das Suchmuster `.[^.]*` wird verwendet). Sie werden auch vom Suchmuster `*` nicht erfaßt, sind aber ansonsten völlig normale Dateien, die erzeugt, editiert und gelöscht werden können.

  Das Suchmuster `.*` sollte nicht verwendet werden, da es auch auf die Verzeichnisse `.` und `..` paßt, und so deren Inhalt ebenfalls mit aufgelistet wird.

- Vorschlag für eine **persönliche Verzeichnisstruktur**:

`~/bin`	Eigene ausführbare Programme
`~/mail`	UNIX-Mails
`~/private`	Private Daten (`chmod go-rwx,o+t`)
`~/public`	Zum Austausch von Daten (`chmod go+rwx`)
`~/shbin`	Eigene Shell-Skripten
`~/src`	Projekte
`~/src/PROJECT1`	Quellen von PROJECT1 (src = source)
`~/src/PROJECT2`	Quellen von PROJECT2 (src = source)
`~/src/PROJECT2/RCS`	Zugehöriges RCS-Verzeichnis (oder symb. Link)
`~/tmp`	Temporäre Dateien (jederzeit wieder löschbar)
`~/test`	Zum Testen von Skripten, ...
`~/text`	Texte, Notizen, ...

- Eine reichhaltige Auswahl von Initialisierungsdateien findet man in den `HOME`-Verzeichnissen der anderen Benutzer. Dort gibt es teilweise auch **unsichtbare Unterverzeichnisse**, die weitere Initialisierungdateien enthalten.

- Zeilen können unter UNIX durch einen Backslash `\` am Zeilenende verlängert werden. Dies funktioniert auf der Befehlszeile, in C-Quellen, Shell-Skripten, Makefiles, ... Durch den Backslash wird der Zeilenvorschub **maskiert** und nicht als Zeilenende betrachtet. Hinter dem Backslash darf kein zusätzliches Zeichen stehen (z.B. ein Leerzeichen), da sonst nicht der Zeilenvorschub, sondern dieses Zeichen maskiert wird.

```
echo "Dies ist eine Text\
zeile verteilt auf mehr\
ere Zeilen"
```

- Unter MS-WINDOWS können über den **Zwischenspeicher** Texte (und Kommandos) zwischen WINDOWS bzw. DOS- und Terminal-Fenstern hin- und herkopiert werden. Dazu mit der Maus den zu kopierenden Text **markieren** und mit `Strg-C` (Copy) bzw. `RETURN` in den Zwischenspeicher ablegen. Das Zielfenster aktivieren, den Cursor an die gewünschte Stelle setzen und den **Einfügemodus** aktivieren. Dann `Strg-V` (Fill)

bzw. `Alt-SPACE` + `r`=Bearbeiten + `e`=Einfügen drücken und der Text wird an der Cursorposition eingefügt. Anschließend den Einfügemodus wieder abschalten.

Achtung: `TAB`s werden dabei in Leerzeichen umgewandelt (`makefile`!), Steuerzeichen in die zwei Zeichen ^ + Buchstabe, ...

- Unter X-WINDOWS können Texte mit der Maus zwischen Terminal-Fenstern hin- und herkopiert werden. Dazu mit der linken Maustaste den zu kopierenden Text **markieren**. Das Zielfenster aktivieren, den Cursor an die gewünschte Stelle setzen und mit der mittleren Maustaste (bzw. der linken + rechten Maustaste gleichzeitig) den Text an der Cursorposition einfügen.

  Achtung: `TAB`s werden dabei in Leerzeichen umgewandelt (`makefile`!), Steuerzeichen in die zwei Zeichen ^ + Buchstabe, ...

- Sehr viele UNIX-Tools sind auch unter DOS und WINDOWS verfügbar. Dazu ist das MKS-Toolkit (Mortice Kern Systems, $\approx$ 300.- Euro) zu installieren.

1.3 Literatur

- Peek, O'Reilly, Loukides, *UNIX Power Tools*, O'Reilly & Associates.
- Gilly, *UNIX in a Nutshell*, O'Reilly & Associates.
- Abrahams, Larson, *UNIX for the Impatient*, Addison Wesley.
- Gulbins, Obermayer, *UNIX*, Springer.
- Kernighan, Pike, *Der UNIX-Werkzeugkasten*, Hanser.
- Christine Wolfinger, *Keine Angst vor UNIX*, VDI-Verlag.

2 Kommandos

2.1 Kommandozeile

Generell lassen sich Kommandos durch vollständiges Neutippen eingeben bzw. korrigieren. Häufig will man jedoch nur kleine Änderungen oder Korrekturen an einem bereits eingegebenen Kommando machen bzw. ein Kommando wiederholen. Deshalb ist es oft günstiger (weniger fehlerträchtig und schneller), ein bereits eingegebenes Kommando wieder hervorzuholen und geeignet zu verändern.

- Auch wenn der Rechner auf eine Eingabe scheinbar nicht reagiert, d.h. die eingetippten Zeichen nicht sofort angezeigt werden (weil er z.B. überlastet ist oder gerade das vorher eingetippte Kommando ausführt), geht keine Eingabe verloren. D.h. es kann auf der Kommandozeile beliebig weitergetippt werden (einschließlich `BACKSPACE`, ...). Sobald ein Kommando beendet ist, wird die noch vorhandene Eingabe ausgewertet und die Befehle darin ausgeführt.

- Viele Kommandos warten bei fehlender Angabe von Dateien auf eine Eingabe von *stdin*, d.h. nach dem Aufruf warten sie auf Eingaben auf der Kommandozeile. Scheint ein Kommando zu **hängen**, obwohl es eigentlich sehr schnell ausgeführt werden müßte, versuchsweise `Strg-D` (am Zeilenanfang!) eintippen (Dateiende unter UNIX), es wird dann in diesem Fall beendet.
- Bitten den Unterschied zwischen `BACKSPACE` und `DELETE` beachten:
  - ▷ `BACKSPACE` löscht das Zeichen vor dem Cursor (spart den Cursor zurückzubewegen).
  - ▷ `DELETE` löscht das Zeichen unter dem Cursor.
- Bitten den Unterschied beim Einfügen von Text in grafischen Oberflächen und in Terminals beachten:
  - ▷ In grafischen Oberflächen steht der Cursor *zwischen* zwei Zeichen, neue Zeichen werden daher *vor* dem 2. Zeichen eingefügt.
  - ▷ In Terminals steht der Cursor *auf* einem Zeichen, neue Zeichen werden *vor* diesem Zeichen eingefügt (nicht danach, wie man vermuten könnte).
- `BACKSPACE` funktioniert beim *login* nicht, dafür aber `DELETE`, d.h. man kann vertippte Benutzernamen/Paßworte *nur* mit `DELETE` korrigieren.
- `~` steht für das eigene `HOME`-Verzeichnis und kann als Pfadbeginn benutzt werden (`/` dahinter nicht vergessen). Beispiel:

```
ls ~/src/proj1/*.c   # nicht ls ~src/proj1/*.c !!!
```

- `~USER` steht für das `HOME`-Verzeichnis des Benutzers `USER`, es kann ebenfalls als Pfadbeginn verwendet werden. Beispiel:

```
ls ~USER/src/*
```

- `cd DIR` (*change directory*) funktioniert auch ohne komplette Pfadangabe, sofern der Name des Vaterverzeichnisses in der Umgebungsvariablen `CDPATH/cdpath` steht (`echo $CDPATH/cdpath` zeigt den aktuellen Inhalt an). Beispiel:

```
cd proj1    # = cd ~/src/proj1 (falls cdpath = ...:~/src:...)
```

- `pwd` (*print working directory*) zeigt das Verzeichnis an, in dem man sich gerade befindet. Beispiel:

```
pwd    # -> z.B. /home/HOST/USER
```

- Nach dem Ablegen eines neuen ausführbaren Programms in einem Pfadverzeichnis `rehash` ausführen, da es sonst von der C-Shell nicht gefunden wird.

- Sollen mehrere Kommandos hintereinander ausgeführt werden, können sie durch `;` getrennt in einer Kommandozeile angegeben werden:

```
echo "TEST1"; sleep 1; echo "TEST2"
```

- Will man die Ausgaben einer Kommandofolge gleichzeitig umlenken oder die ganze Kommandofolge in den Hintergrund schicken, so sind die Kommandos zu klammern:

```
( echo "TXT1"; sleep 1; echo "TXT2" ) > xxx      # Nur stdout umlenken
( echo "TXT1"; sleep 1; echo "TXT2" ) >& xxx     # stdout+stderr... (csh)
( echo "TXT1"; sleep 1; echo "TXT2" ) > xxx 2>&1 # stdout+stderr... (bash)
( echo "TXT1"; sleep 1; echo "TXT2" ) &          # In Hintergrund schalten
```

- Zum Kopieren oder Verschieben von Dateien muss man weder im Quell- noch im Zielverzeichnis stehen:

```
pwd   # -> z.B. /home/HOST/USER/src
cp /src/PROJECT/src/proj1/*.[ch] proj1/src
```

  kopiert alle C-Quell- und Header-Dateien aus dem Verzeichnis `/src/PROJECT/src/proj1` in das Verzeichnis `/home/HOST/USER/src/proj1/src`.

- Kein selbstentwickeltes Kommando oder Shell-Skript `test` nennen, da dies ein Shell-Befehl ist, der in Shell-Skripten für das Testen von Bedingungen verwendet wird. Shell-Skripte rufen dann statt dessen das eigene Programm `test` auf, was die merkwürdigsten Folgen haben kann.

2.1.1 History (csh)

- `^ALT^NEU` ändert im letzten ausgeführten Kommando den Text `ALT` in den Text `NEU` um (der zuerst passende Text wird 1x ersetzt) und führt das veränderte Kommando dann aus. Beispiel:

```
gerp Func *.c    # Tippfehler, mist@*!?
^er^re           # Oder auch ^gerp^grep
```

  führt das Kommando `grep Func *.c` aus.

  Hinweis: Ist der Buchstabe nach dem `^` (Hütchen) ein Vokal (`aeiou`), muss evtl. ein Leerzeichen nach dem `^` eingetippt werden, damit beide Zeichen getrennt erscheinen (tastaturabhängig!).

- Ab dem Login werden alle abgesetzten Kommandos von $1 - n$ durchnumeriert, `history` liefert eine Liste der letzten n Kommandos (einstellbar über `set history=NN` in der Datei `.cshrc` bzw. `HISTSIZE=NN` in der Datei `.bashrc`). Mit der Option `-r` werden die neuesten Kommandos zuerst aufgelistet.

- Will man eine Liste der abgesetzten Kommandos `CMD` in der Datei `xxx` aufheben, kann das mit:

```
history -r | grep CMD > xxx
```

erreicht werden.

- `!NR` führt das Kommando mit der Nummer `NR` aus (vorher mit `history -r | more` anzeigen lassen).
- `!PREFIX` führt das letzte Kommando aus, das den Anfang `PREFIX` hatte, d.h. hat man ein Kommando `make PGM` und anschließend ein paar andere Kommandos ausgeführt, so kann das Kommando mit `!m`, `!ma`, `!mak`, ... erneut ausgeführt werden.

  Hinweis: Der Präfix muss so lang gewählt werden, dass man kein anderes Kommando mit dem gleichen Präfix erwischt, das später verwendet wurde.
- `!!` führt das letzte Kommando aus. Dies läßt sich vor allem dazu ausnützen, das *Ergebnis des letzten Kommandos* in einem anderen Kommando zu verwenden. Beispiel:

```
grep -l Func *.c   # -> Liste aller Dateien die Func enthalten
vi `!!`            # -> Diese Liste editieren
```

2.1.2 Commandline-Edit und Filename-Completion (tcsh)

- Mit Cursor-Auf/Ab lassen sich die letzten n Kommandos (einstellbar) wieder holen und dann per `RETURN` ausführen.
- `Strg-A` springt zum *Anfang*, `Strg-E` springt zum *Ende* der aktuellen Befehlszeile.
- Die sogenannte **Filename-Completion** per `TAB`-Taste erleichtert das Eintippen von Kommandos. `TAB` (`Strg-I`) versucht Kommandonamen (1. Wort), Dateinamen (2.- n. Wort), Variablennamen (nach `$`), Benutzernamen (nach `~`) und Rechnernamen (nach `@`) hinter denen der Cursor steht, zu vervollständigen. Ist dies nicht eindeutig möglich, werden alle möglichen Fortsetzungen angezeigt (bzw. beim 2. Tippen von `TAB`).
- `Strg-D` (*directory*) zeigt den Inhalt des aktuellen Verzeichnisses an.

2.1.3 Aliase (csh)

- Mit `alias` lassen sich Abkürzungen oder neue Kommandos schnell und einfach (über)definieren. Mit `unalias` kann ein Alias wieder entfernt werden. `alias` allein liefert eine Liste aller aktuell gültigen Aliase, `alias CMD` liefert den Alias des Kommandos `CMD` (falls einer existiert):

`alias ls "ls -F"`	`ls` steht für `ls -F` (nicht rekursiv!)
`unalias ls`	`ls`-Alias wieder löschen
`alias ls`	`ls`-Alias ausgeben
`alias`	Alle Aliase ausgeben

- Es empfiehlt sich. die eigenen Aliase in einer Datei `~/.alias` abzulegen und in die Datei `~/.cshrc` einen Aufruf der Form:

```
source $HOME/.alias
```

einzufügen.

- Nach Änderungen an der Datei `~/.alias` oder `~/.profile` oder `~/.cshrc` und `~/.login` können die Änderungen durch Eingabe von `source FILE` auf der Kommandozeile sofort wirksam gemacht werden.
- Es empfiehlt sich folgender Alias `h` (*history*), um die letzten Kommandos schnell anzeigen zu können:

```
alias h 'history -r | more'
```

- Damit man zu `ls` nicht immer Optionen bzw. `more` dahinter eintippen muss, empfehlen sich folgende Aliase (die am besten in `~/.alias` aufzunehmen sind):

`alias la  "ls -aF \!*" | more`	Alle Dateien
`alias ll  "ls -lF \!*" | more`	Dateien im langen Format
`alias ld  "ls -lF \!* | grep /^[dl]/" | more`	Nur Verzeichnisse+Links
`alias lr  "ls -RF \!*" | more`	Dateien rekursiv
`alias ls  "ls -F \!*" | more`	Dateityp mit ausgeben
`alias lss "ls -lF \!*" | sort -r +4 -5 | more`	Nach Größe sortieren
`alias lu  "ls -lFt \!* | head"`	10 neuesten Dateien
`alias lo  "ls -rlFt \!* | head"`	10 ältesten Dateien

- Als Alias empfiehlt sich folgendes Kommando `ff` (**File-Find**), das den Norton-Utilities unter DOS nachempfunden ist (Dateinamen mit Sonderzeichen `*?[]` sind beim Aufruf in `"..."` einzuschließen):

```
alias ff 'find . -name \!* -print'
```

Aufruf z.B. durch:

```
ff "*.sh"
```

- Die Aliase `va="vi ~/.alias"` und `sa="source ~/.alias"` erleichtern die Definition und Aktivierung neuer Aliase.

2.1.4 Datei-Umlenkung (sh + csh)

- Die `sh` bzw. `csh` kennen folgende Syntax für Datei-Umlenkungen:

Umlenkung	sh	csh
stdout in *file* speichern	`prog > file`	`prog > file`
stderr in *file* speichern	`prog 2> file`	—
stdout und *stderr* in *file* speichern	`prog > file 2>&1`	`prog >& file`
stdin aus *file* holen	`prog < file`	`prog < file`
stdout ans Ende von *file* anhängen	`prog >> file`	`prog >> file`
stderr ans Ende von *file* anhängen	`prog 2>> file`	—
stdout und *stderr* ans Ende von *file* anhängen	`prog >> file 2>&1`	`prog >>& file`
stdin bis *text* von Tastatur holen (Here-Dok.)	`prog <<text`	`prog <<text`
stdout von *prog1* in *prog2* pipen	`prog1 | prog2`	`prog1 | prog2`
stdout und *stderr* von *prog1* in *prog2* pipen	`prog 2>&1 | prog2`	`prog |& prog2`

2.2 Shell-Skripte

Shell-Skripte lassen sich mit DOS-Batchdateien vergleichen und bieten die Möglichkeit, lange und öfter auszuführende Kommandofolgen zu automatisieren. Im Gegensatz zur DOS Batch-Sprache stellen sie allerdings deutlich mehr Möglichkeiten wie z.B. if-then-elif-else-endif-Kaskaden, Schleifenkonstrukte, Funktionen, Auswertung von Ausdrücken, usw. zur Verfügung.

- Mit Shell-Skript ist immer ein **Bourne-Shell-Skript** gemeint (`sh`), da C-Shell-Skripten aus Geschwindigkeitsgründen sowie Fehlern und Beschränkungen der C-Shell nicht zu empfehlen sind (obwohl prinzipiell möglich). Diese Skripten sind dann auch portabel.
- Eigene Shell-Skripten mit der Extension `.sh` versehen, um sie leicht ausfindig machen zu können (bei installierten Shell-Skripten kann die Endung `.sh` fehlen).
- Shell-Skripten können entweder mit der Shell gestartet werden durch:

  ```
  sh SCRIPT
  ```

  oder sie werden als ausführbar gekennzeichnet durch:

  ```
  chmod u+x SCRIPT
  ```

  und können dann durch Eingabe ihres Namens `SCRIPT` aufgerufen werden.
- Shell-Skripten grundsätzlich mit folgender 1. Zeile beginnen lassen:

  ```
  #!/bin/sh
  ```

  Grund: Kommentare werden in Shell-Skripten durch `#` eingeleitet. Da aber ein `#` als erstes Zeichen in der ersten Zeile eines Shell-Skripts dafür sorgt, dass zur Ausführung des Skripts die C-Shell aufgerufen wird, muss obige Zeile immer die erste Skript-Zeile sein. Sie erzwingt (über den Kernel) die Ausführung unter Kontrolle der normalen Bourne-Shell (UNIX-Konvention). Analog erzwingt eine 1. Zeile der Form:

  ```
  #!/bin/grep MUSTER
  #!/bin/sed -f
  #!/bin/awk -f
  #!/usr/local/bin/perl -f
  ...
  ```

  die Ausführung der Datei per *gawk* oder *perl* oder ...
- Um ein Shell-Skript auf **Signale** reagieren zu lassen, muss das Kommando `trap` verwendet werden. Üblicherweise sollte ein Shell-Skript alle seine Zwischendateien aufräumen und eine Abbruchmeldung ausgeben. Im folgenden Beispiel werden die Signale 0, 1, 2 und 15 abgefangen, bei ihrem Eintreffen eine Meldung ausgegeben, temporäre Dateien im Verzeichnis `$HOME/tmp` gelöscht und dann das Programm abgebrochen:

```
trap "echo 'cleaning up...'; rm -f $HOME/tmp/$$*; exit" 0 1 2 15
```

- Da Shell-Skripte häufig recht *adhoc entworfene Programme* sind, sollten sie durch einen Kopf beschrieben werden und eine Usage-Meldung ausgeben, damit man auch nach zwei Wochen noch weiß, wozu sie eigentlich gedient haben und wie sie aufgerufen werden:

```
#!/bin/sh
#-------------------------------------------------------------
# Dient zum Löschen des Verzeichnisbaumes ab dem /-Verzeichnis
#-------------------------------------------------------------
...
Usage() {
    ...
}
```

- Ein kleines Shell-Skript zum abgesicherten Ersetzen aller DOS-Umlaute und des scharfen ß durch ihr zweibuchstabiges Äquivalent (`2ascii.sh`) und alle sonstigen Zeichen außerhalb des Bereichs 32-127 durch Leerzeichen:

```
#!/bin/sh
for file in "$@"; do
   echo -n "converting $file..."
   sed 's/\204/ae/g;s/\224/oe/g;s/\201/ue/g' $file |
      sed 's/\216/Ae/g;s/\231/Oe/g;s/\232/Ue/g;s/\341/ss/g' |
      tr -c "[ -~]" " " > $$.tmp
# Nur ersetzen, falls kein Fehler auftrat
   if [ $? -eq 0 ]; then
       mv $$.tmp $file
       echo "done"
   else
       echo "error"
   fi
done
```

2.3 Suchen von Kommandos bzw. Kommandobeschreibungen

- `which CMD` (oder `type CMD`) liefert den vollständigen Zugriffspfad auf das Kommando `CMD` bzw. die Information, dass es ein in die Shell eingebauter Befehl (Shell-Builtin) oder ein Alias oder eine Funktion ist. Beispiele:

`which vi`	→ `/ucb/vi`
`which echo`	→ `echo: shell built-in command.`
`which ll`	→ `aliased to ls -lF !* | more`

Hierdurch läßt sich feststellen, welches Programm durch ein eingegebenes Kommando wirklich ausgeführt wird (heißt in der Korn-Shell `type`).

- `man NAME` (*manual*) liefert eine Beschreibung des Kommandos `NAME`, einer Konfigurationsdatei `NAME` oder einer C-Funktion `NAME` und gibt sie per `more` auf dem Bildschirm aus. `man man` liefert eine Beschreibung des `man`-Kommandos. `man [-s] 5 NAME` liefert eine Beschreibung von `NAME` aus der Sektion 5 der Online-Manuals.

  Hinweis: Sucht man ein Kommando, dessen Namen man nicht kennt, so lohnt es sich oft, das Manual zu einem *ähnlichen Kommando* aufzurufen und die Querverweise zu anderen Kommandos im `SEE ALSO`-Abschnitt anzusehen (mit `/SEE` suchen).

- Beim Umlenken auf Datei oder den Drucker ist zu beachten, dass `man` Worte durch Unterstreichen auszeichnet, indem es nach jedem Buchstaben ein `BACKSPACE` und einen Unterstrich schickt. Die Ausdrucke sehen entsprechend seltsam aus. Durch folgendes Sed-Kommando können diese Sequenzen entfernt werden (`rmctrlh_.sh`):

  ```
  man CMD | sed 's/CTRLH_//g' > xxx   # CTRLH_ entfernen
  ```

- Ausgedruckt wird eine Manual-Seite durch folgenden Kommando (troff):

  ```
  man -t CMD | lp(r)
  ```

  ```
  man CMD | sed 's/CTRLH_//g' > xxx   # CTRLH_ entfernen
  ```

- `apropos NAME` sucht in einer Datei mit einzeiligen Kurzbeschreibungen aller Kommandos und C-Funktionen Einträge, die `NAME` enthalten. `apropos apropos` liefert eine Kurzbeschreibung von `apropos`.

- `whatis NAME` sucht in einer Datei mit einzeiligen Kurzbeschreibungen aller Kommandos und C-Funktionen Einträge, die den Kommandonamen `NAME` enthalten. `whatis whatis` liefert eine Kurzbeschreibung von `whatis`.

- Viele Kommandos geben beim Aufruf ohne Argumente oder bei Angabe einer Option `-h` oder `--help` eine **Usage-Meldung** aus.

- Häufig verstehen Kommandos eine **Unmenge von Optionen**, die man sich kaum merken kann. Daher nur die Optionen merken, die man öfter braucht, und zwar über folgende **Eselsbrücke**: *Optionen stehen für ein Wort, das ihre Funktion beschreibt*. Wie die folgende Liste von typischen Standard-Optionen zeigt, kann die gleiche Option (leider) bei vielen Kommandos verschiedene Funktionen auslösen:

`-a`	all, append
`-b`	???
`-c`	count, command
`-d`	debug, delimiter, directory
`-e`	edit, execute, expand
`-f`	field, file, foldcase, force
`-g`	global
`-h`	head(er), help
`-i`	ignore case, input, interactive
`-j`	???
`-k`	keep, key
`-l`	line, list files, login, long format
`-L`	follow symbolic links
`-m`	multicolumn
`-n`	newer, noexec, non-interactive, noprint, numeric
`-o`	output, out, order by
`-p`	print, process id
`-q`	quick, quiet
`-r`	recursive, reverse, root directory
`-R`	Recursive
`-s`	silent, size, sort, subdirectories
`-t`	tab char , tail, time
`-u`	unbuffered, uniq, update
`-v`	vice versa, verbose, visual
`-w`	warning, wide format, width, words
`-x`	exclude, execute, extended
`-y`	yes
`-z`	zip

- `ident FILE` bzw. `what FILE` liefern die von RCS/SCCS im Quellcode der Datei `FILE` eingefügten Identifizierungen (`$Id...$` bzw. `@(#)...\n`) zurück. Man kann sie auf beliebige (auch binäre) Dateien und Bibliotheken anwenden, um deren Versionsnummer zu erhalten.
- `strings CMD` liefern alle druckbaren Zeichenketten des Kommandos `CMD` (Usage-Meldung, Fehlermeldungen, Variablennamen, ...).

2.4 Wichtige Zugriffspfade

`/home`	`HOME`-Verzeichnisse aller Anwender
`/src/PROJECT/src`	Quelldateien für Projekte
`/usr/bin`	Ausführbare Programme
`/usr/4bin`	Ausführbare Programme (System IV)
`/usr/5bin`	Ausführbare Programme (System V)
`/usr/include`	System-Includedateien
`/usr/include/sys`	System-Includedateien
`/usr/lib`	Systembibliotheken
`/usr/local/lib`	Eigene Bibliotheken
`/usr/local/include`	Eigene Includedateien
`/usr/local/lib/tex`	TEX-Programme
`/usr/local/lib/tex/inputs`	LATEX-Styledateien
`/usr/local/lib/tex/inputs/local`	Eigene LATEX-Styledateien
`/usr/ucb`	Ausführbare Programme (BSD)

2.5 Wichtige Dateien

Bei jedem Login werden standardmäßig eine Reihe von Dateien angezogen, die Aliase definieren, Umgebungsvariablen setzen, den Pfad erweitern, usw. Folgende Dateien werden dabei ausgewertet (*in genau dieser Reihenfolge*):

1.	`/etc/profile`	Beim Bourne-Shell-Login
1.	`$HOME/.profile`	Beim Bourne-Shell-Login
2.	`$HOME/.bashrc`	Bei jedem Start einer Bash-Shell
2.	`$HOME/.cshrc`	Bei jedem Start einer C-Shell
3.	`$HOME/.tcshrc`	Bei jedem Start einer `tcsh` (statt `.cshrc` falls vorhanden)
4.	`$HOME/.login`	Beim ersten C-Shell-Login
5.	`$HOME/.logout`	Beim C-Shell-Logout
	`$HOME/.alias`	Eigene Aliase

Durch Angabe von

```
echo "Starting FILENAME..."; sleep 1
```

am Anfang und

```
echo "Leaving FILENAME..."; sleep 1
```

am Ende jeder dieser Dateien kann man den Ablauf verfolgen.

2.6 Umgebungsvariablen

- Viele Programme suchen sich Informationen aus Umgebungsvariablen heraus und verwenden sie zur Initialisierung, zum Zugriff auf Dateien, usw. Beispiel:

gcc	INCLUDE
less	LESS, ...
ls	LS_OPTIONS, LS_COLORS
make	CC, LD, ...
tex	TEXINPUTS
vi	EXINIT

- Durch folgende Kommandos können Umgebungsvariablen gesetzt, gelöscht und angesehen werden:

`export VAR=text`	`VAR` gleich `text` setzen (bash)
`setenv VAR text`	`VAR` gleich `text` setzen (csh)
`unset VAR`	`VAR` löschen
`env`	Alle Umgebungsvariablen anzeigen
`echo $VAR`	Wert der Umgebungsvariablen `VAR` anzeigen
`env | grep XYZ`	Alle Umgebungsvariablen mit `XYZ` im Namen anzeigen

- Die Pfadvariable `PATH` kann durch folgendes Kommando um neue Verzeichnisse verlängert werden (vorne oder hinten anhängen):

```
setenv PATH $PATH:/usr/local/test   # Hinten anhängen (csh)
setenv PATH /usr/local/test:$PATH   # Vorne anhängen (csh)
PATH=$PATH:/usr/local/test          # Hinten anhängen (bash)
PATH=/usr/local/test:$PATH          # Vorne anhängen (bash)
```

  Diese Sequenz kann nur direkt eingetippt, als `alias` definiert oder per `source` aus einer Datei gelesen werden. In einem Shell-Skript ausgeführt, setzt sie nur die Pfad-Variable für die zur Ausführung dieses Skripts gestartete Shell. *Nach dem Verlassen dieser Shell gilt wieder der alte Pfad!*

2.7 Benutzer und Gruppen

Die Benutzer und Gruppen werden normalerweise von einem zentralen Namens-Server verwaltet. Daher enthalten die üblichen Dateien (`/etc/passwd` und `/etc/group`) zu Verwaltung dieser Informationen keine brauchbaren Daten und das übliche Kommando `passwd` zum Ändern eines Passwortes funktioniert nicht (`yp` = **yellow pages**/**NIS**).

- Mit `yppasswd` kann das eigene Paßwort netzweit geändert werden.
- Mit `ypcat passwd` erhält man eine Liste aller Benutzer.
- Mit `ypcat groups` erhält man eine Liste aller Benutzergruppen. ???
- Mit `ypcat hosts` erhält man eine Liste aller Rechner.
- Mit `su USER` kann man ein anderer Benutzer werden (sofern man dessen Paßwort kennt), behält aber seine eigene Umgebung und das aktuelle Verzeichnis bei. Mit `exit` wird man wieder zum vorherigen Benutzer.

- Mit `su - USER` kann man ein anderer Benutzer werden (sofern man dessen Paßwort kennt), und erhält auch die Umgebung des Benutzers und sein `HOME`-Verzeichnis. Mit `exit` wird man wieder zum vorherigen Benutzer.
- Mit `rlogin HOST` loggt man sich auf einem anderen Rechner ein. In der Datei `~/.rhosts` können die Rechner und Benutzer angegeben werden, die das dürfen, ohne dass sie nach dem Paßwort gefragt werden (Vorsicht, potentielle Sicherheitslücke!). Inhalt z.B.:

```
HOST1 USER1
HOST2 USER2
...
```

- `who` oder `finger` liefern eine Liste aller aktuell eingeloggten Benutzer und ihres Terminals. `finger USER` liefert Informationen zum Benutzer `USER`.
- `whoami` oder `who am i` gibt den Benutzernamen aus.
- `id` gibt den Benutzernamen + User-ID, den aktuellen Gruppennamen + Group-ID sowie alle weiteren Gruppennamen + Group-IDs aus, zu denen der aktuelle Benutzer gehört.
- Mit `newgrp GROUP` kann die aktuelle Gruppe geändert werden (nur für das neu Anlegen von Dateien relevant):

```
newgrp tex
```

- Mit `chgrp GROUP FILE...` können Dateien einer anderen Gruppe zugeordnet werden:

```
chgrp sdws *.c
```

- Mit `chown USER FILE` können Dateien einem anderen Benutzer zugeordnet werden. **Vorsicht:** das kann der Besitzer der Datei 1x machen, dann gehören sie jemand anderem (und nix geht anschließend mehr damit).

2.8 Zugriffsrechte

- Mit `chmod MODE FILE` kann der Zugriffsmodus von Dateien geändert werden, `MODE` kann eine Zeichenkette der Form `[ugoa][+-=][rwxst]` sein, mit:

```
u = user          + = hinzufügen       r = read
g = group         - = wegnehmen        w = write
o = others        = = absolut setzen   x = execute
a = all                                s = set user/group id
    (user + group + others)            t = sticky
```

- Mit `ls -l` werden diese Rechte am Zeilenanfang folgendermaßen angezeigt:

```
*rwx       = user  read/write/execute rights
|   rwx    = group read/write/execute rights
|      rwx = other read/write/execute rights
|
file type  - = Normale Datei
           d = Verzeichnis
           l = Symbolischer Link
           c = Character Device (Terminal, Drucker)
           b = Block Device (Platte, Band)
           s = Socket
           p = Named Pipe
```

- Die 9 Rechte-Bits werden oft als dreistellige Oktalzahl dargestellt bzw. können so auch beim Kommando `chmod` angegeben werden:

user	read	r-------- = 400
user	write	-w------- = 200
user	execute	--x------ = 100
group	read	---r----- = 040
group	write	----w---- = 020
group	execute	-----x--- = 010
other	read	------r-- = 004
other	write	-------w- = 002
other	execute	--------x = 001

Beispiele:

```
      uuugggooo          uuugggooo          uuugggooo
123 = --x-w--wx    456 = r--r-xrw-    007 = ------rwx
```

- Das Recht `x` (executable = ausführbar) bedeutet bei:
  - ▷ **Textdateien**, dass die Datei als Shell-Skript aufgerufen werden kann.
  - ▷ **Binären Dateien**, dass die Datei ein ausführbares Programm ist.
  - ▷ **Verzeichnissen**, dass das Verzeichnis mit `cd` betreten werden kann.
- Hat man das **Schreibrecht** *für eine Datei, aber nicht auf das Verzeichnis*, in dem sie steht, so kann die Datei zwar verändert, aber sie kann nicht gelöscht, umbenannt oder verschoben werden (Veränderung an der Datei, nicht am Verzeichnis!).

  Hat man das **Schreibrecht** *für ein Verzeichnis, aber nicht auf eine Datei darin*, so kann die Datei zwar nicht verändert, aber sie kann gelöscht, umbenannt und verschoben werden (Veränderung am Verzeichnis, nicht an der Datei!).
- Es gibt drei weitere Rechte (die sich auch als 4. Oktalzahl ansprechen lassen), die folgende Möglichkeiten bieten:
  - ▷ Mit dem **Set-User-ID-** bzw. dem**Set-Group-Id-Bit** für Dateien können Programme unter einer anderen Berechtigung als der eigenen Benutzer- oder Gruppenberechtigung ausgeführt werden (z.B. `passwd`).

- ▷ Mit dem **Set-Group-Id-Bit** können Verzeichnisse ihre Gruppen-Kennung an alle darin angelegten Dateien und Verzeichnisse (inklusive dieses Rechts) weitergeben (für Arbeitsgruppen-Verzeichnisse sinnvoll).
- ▷ Mit dem **Sticky-Bit** kann bei Verzeichnissen festgelegt werden, dass statt den Verzeichnisrechten die Dateirechte auch beim Löschen und Überschreiben gelten (z.B. `/tmp`).

user	setuid	`--s------ = 4000`
group	setgid	`-----s--- = 2000`
all	sticky	`--------t = 1000`

- `umask` (*user mask*) liefert im Oktalcode die Rechte, die beim Anlegen einer Datei automatisch entzogen werden:

```
umask   # -> 022, d.h. Schreibrecht für Group und Others entzogen
```

  Diese Maske ist normalerweise auf `022` voreingestellt, sie kann mit `umask MASK` auch gesetzt werden. Aus Gründen der Offenheit und Fairneß sollten allerdings die Leserechte für andere nur bei wirklich privaten Dateien entzogen werden, indem das Kommando `chmod go-rw` oder `chmod 600` verwendet wird. Folgendes Kommando ist daher *nicht* anzuwenden:

```
umask 077   # Entzieht Group und Others sämtliche Rechte
```

- Für den Superuser (`root`) gelten *keinerlei Zugriffsbeschränkungen*, d.h. er darf alle Dateien einsehen, verändern und löschen.

2.9 Terminal

- Wenn das Terminal mal spinnt (weil z.B. eine Binärdatei auf dem Bildschirm ausgegeben wurde) oder nicht mehr zu reagieren scheint, kann es mit folgenden Befehlen meist wieder in seinen Grundzustand zurückgesetzt werden:

```
stty sane
reset
```

  Diese Befehle sollten mit `Strg-J` eingeleitet *und* beendet werden, da die `RETURN`-Taste möglicherweise nicht mehr richtig interpretiert wird (`Strg-J` funktioniert immer).

- Die Backspace-Taste kann durch folgenden Befehl mit der `erase`-Funktionalität belegt werden (löschen der zuletzt eingegebenen Zeichen).

```
stty erase Strg-V Backspace
```

- Läßt man versehentlich eine sehr große Datei mit irgendeinem UNIX-Kommando auf den Bildschirm ausgeben, so wirkt der Abbruch mit `Strg-C` manchmal nicht sofort. Macht man aus dem Terminalfenster ein Symbol, dann wird die Bildschirmausgabe wesentlich schneller durchgescrollt (kein Bildschirmupdate notwendig).
- Kann das Terminal nicht mehr in einen vernünftigen Zustand gebracht werden, die Terminal-Emulation einfach abbrechen und neu starten.

2.10 Prozesse (csh/ksh)

In jeder Shell können ein Prozeß im Vordergrund und beliebig viele Prozesse im Hintergrund laufen. In der C-Shell/Bash kann zusätzlich jederzeit zwischen diesen Prozessen gewechselt werden.

- Mit `Strg-C` läßt sich das aktuell im Vordergrund laufende UNIX-Kommando abbrechen.
- Mit `&` am Ende eines Kommandos schickt man das Kommando in den Hintergrund (Ausgaben sollten auf Datei umgelenkt werden, sonst stören sie den Vordergrundprozeß):

```
find / -name "core" -print > core.lst &
```

- Ein im Vordergrund laufendes Programm kann mit `Strg-Z` unterbrochen werden. Es ist dann gestoppt und wird mit `bg` (*Background*) im Hintergrund bzw. mit `fg` (*Foreground*) wieder im Vordergrund gestartet.
- Mit `jobs` erhält man eine durchnumerierte Liste aller Hintergrundprozesse. Durch `fg JOBNR` kann der Hintergrundprozeß `JOBNR` in den Vordergrund geschaltet werden, durch `kill %JOBNR` kann er abgebrochen werden.
- `ps` (*processes*) liefert eine Übersicht über die eigenen Prozesse, die Prozeßnummer `PID` kann mit `kill PID` (vom Prozeß ignorierbar) oder `kill -9 NR` (nicht vom Prozeß ignorierbar) verwendet werden, um einen (eigenen) Prozeß zu beenden (Signal 9 bricht einen Prozeß sicher ab).
- `ps -elf | grep gawk` oder `ps aux  grep awk`— liefert eine Übersicht über alle `gawk`-Prozesse auf dem Rechner.
- Prozesse werden normalerweise beendet, wenn ihr Vaterprozeß beendet wird. Mittels `nohup` (*no hangup*) kann ein vom Terminal aus im Hintergrund gestarteter Prozeß dagegen abgesichert werden, d.h. auch wenn das Terminal beendet wird, läuft der Prozeß weiter. Sinnvollerweise sollten alle Ein- und Ausgaben der Prozesses aus einer Datei kommen/in eine Datei umgelenkt werden. Default für die Ausgabe ist die Datei `nohup.out`. Beispiel:

```
nohup CMD < INPUT >& OUTPUT &
```

2.11 Plattenplatz

- `du -k` (*disk used*) gibt den belegten Platz im aktuellen Verzeichnis und seinen Unterverzeichnissen (in KByte) aus.
- `df -k .` (*disk free*) gibt den belegten und freien Platz der Platte aus, auf der das aktuelle Verzeichnis liegt.

- `g(un)zip`, `compress/uncompress` bzw. `pack/unpack` (ent)komprimieren Dateien und/oder ganze Verzeichnisbäume. Das Komprimieren lohnt sich, wenn große Datenmengen längere Zeit nicht gebraucht werden.
- Mittels `zcat` bzw. `zmore` können per `gzip` gepackte Dateien *on-the-fly* für Verarbeitungszwecke bzw. zum Ansehen entpackt werden, ohne zusätzlichen Plattenspeicher zu belegen.
- Deutlich mehr Plattenplatz läßt sich sparen, wenn:
  - ▷ Alle Quellen mit RCS verwaltet und keine unnötigen Sicherheitskopien angelegt werden.
  - ▷ Zwischendateien (`*.dvi, *.aux, *.o, ...`) gelöscht werden (natürlich nur, wenn sie sich wieder herstellen lassen). Dazu sollte jedes `makefile` ein Ziel `clean` besitzen, das alle nicht benötigten Dateien löscht und nur zum Lesen ausgecheckte Dateien mit `rcsclean` freigibt:

```
clean:
TAB rm *.o *.dvi *.aux *.tmp ...   # TAB=Tabulator
TAB rcsclean
TAB ...
```

3 Dateien

3.1 Dateinamen anzeigen/suchen

- Dateinamen können mit beliebig komplizierten Mustern gesucht werden. Neben normalen Zeichen lassen sich auch Wildcards angeben. Dabei gilt:

`*`	Steht für eine beliebig lange Folge beliebiger Zeichen
`?`	Steht für ein beliebiges Zeichen
`[a-z]`	Steht für ein Zeichen aus der Menge `a-z`
`[!a-z]`	Steht für ein Zeichen *nicht* aus der Menge `a-z` (sh, ksh)
`[^a-z]`	Steht für ein Zeichen *nicht* aus der Menge `a-z` (csh)

Beispiele:

- ▷ Die Namen aller C-Quelldateien im aktuellen Verzeichnis, allen seinen UnterVerzeichnissen und UnterUnterVerzeichnissen anzeigen:

```
ls *.c */*.c */*/*.c
```

- ▷ Alle C- und H-Dateien im aktuellen Verzeichnis auflisten:

```
ls *.[ch]    # oder ls *.c *.h
```

- ▷ Alle Dateien (`-a`) mit Dateinamen der Länge 2, 3 und 4 auflisten (wg. `-d` bei Unterverzeichnissen nur ihren Namen statt ihren Inhalt):

```
ls -ad ?? ??? ????
```

Die gesamte Kommandozeilenlänge darf etwa 20.000 Zeichen erreichen. Statt die Dateien aufzulisten, können sie auch einem Editor oder sonstigen Kommandos übergeben werden. Beispiel:

```
vi `grep error *.[ch]`
```

- `ls` (*list*) kennt eine Reihe von Optionen, um die zu einer Datei angezeigten Informationen zu erweitern. Folgende Optionen sind nützlich:

`-a`	all	Alle Dateien anzeigen (auch solche die mit `.` beginnen)
`-A`	almost all	Analog `-a`, nur `.` und `..` weglassen)
`-d`	directory	Nur Verzeichnisnamen, nicht Verzeichnisinhalte anzeigen
`-F`	filetype	Dateityp hinter Dateinamen anzeigen (`/`=Verzeichnis, `*`=ausführbare Datei, `@`=symbolischer Link, `=`=Socket)
`-l`	long	Zugriffsrechte, #Links, Benutzer, Gruppe, Länge, Datum/Zeit und Dateiname anzeigen
`-R`	recursive	Inhalte aller Unterverzeichnisse ebenfalls anzeigen
`-r`	reverse	Sortierung umkehren
`-t`	time	Chronologische Reihenfolge (Inhaltsänderung, neueste Datei zuerst)
`-u`	usage	Chronologische Reihenfolge (Zugriffszeit)
`-c`	change	Chronologische Reihenfolge (Inode-Änderung)

- `find` sucht ab einem Verzeichnis in diesem und in allen seinen Unterverzeichnissen nach Dateien, die bestimmte Kriterien erfüllen. Das Verhalten von `find` ist durch viele Optionen beeinflußbar, die wichtigsten sind:

`-exec CMD ...`	Kommando `CMD` ausführen (`{}` steht darin für aktuellen Dateinamen)
`-mtime NUM`	Alter in Tagen auswählen (`+NUM` = mindestens, `-NUM` = höchstens)
`-name "REGEX"`	Datei per Suchmuster `REGEX` auswählen
`-newer FILE`	Neuer als Datei `FILE`
`-perm NUM`	Zugriffserlaubnis `NUM` auswählen (z.B. `777`, `-NUM`=mindestens)
`-print`	Gefundenen Dateinamen ausgeben (am Ende)
`-type .`	Dateityp auswählen (`f`=file, `d`=directory, `l`=symbolic link, ...)
`-user USER`	Anwender `USER` auswählen
`-a`	Und
`-o`	Oder
`!`	Nicht
`\(...\)`	Klammern

Beispiele zum Aufruf von `find`:

▷ Sucht alle vom aktuellen Verzeichnis aus erreichbaren C-Quelldateien und gibt ihren Namen (+ Zugriffspfad aus):

```
find . -name "*.c" -print
```

▷ Sucht ab dem `HOME`-Verzeichnis alle einbuchstabigen Dateien der Form `a-z` und löscht sie (`{}` steht für den jeweils gefundenen Namen, das Kommando nach `-exec` muss mit `;` abgeschlossen werden, der `;` muss mit `\` gequotet werden, da er sonst von der Shell interpretiert und nicht an `find` weitergegeben wird):

```
find ~ -name "[a-z]" -exec rm {} \;
```

▷ Sucht ab dem `ROOT`-Verzeichnis alle `core`-Dateien und löscht sie (das kann natürlich nur der Benutzer `root` machen):

```
find / -name "core" -exec rm {} \;
```

▷ Sucht ab dem `HOME`-Verzeichnis des Benutzers `USER` alle Verzeichnisse und sammelt sie in der Datei `dirs`:

```
find ~USER -type d -print > dirs
```

3.2 Die verschiedenen Dateizeiten

- UNIX kennt 3 verschiedene Dateizeiten, die **last modification time**, die **last (inode) change time** und die **last access time**. Der Unterschied zwischen einer *change time* und einer *modification time* entspricht dem Unterschied zwischen dem Ändern einer Packungsbeschriftung (Eigenschaft der Datei) und dem Ändern des Packungsinhalts (Eigenschaft des Dateiinhalts). Bei:

  ```
  chmod a-w FILE
  ```

  handelt es sich um einen *Change* von `FILE`. Bei:

  ```
  echo foo >> FILE
  ```

  handelt es sich um eine *Modification* von `FILE`. Ein *Change* ändert den **Inode** der Datei, eine *Modification* ändert den Dateiinhalt.

- Die **access time** ist der Zeitpunkt der letzten Lese- oder Schreiboperation auf eine Datei. Das Lesen einer Datei ändert ihre *access time*, aber nicht ihre *change time* (die Information über die Datei bleibt unverändert) und auch nicht ihre *modification time* (der Dateiinhalt bleibt ebenfalls unverändert).

- Gelegentlich wird die *change time* fälschlicherweise als **creation time** bezeichnet, dies ist nicht korrekt.

- Bei `ls -l` wird üblicherweise die *modification time* ausgegeben. Mit folgenden Schaltern werden die Dateien nach einer der 3 Dateizeiten sortiert (neuestn zuerst) ausgegeben:

`-c`	change	change time
`-t`	time	modification time
`-u`	used	access time

  Durch zusätzliche Angabe der Option `-r` (reverse) läßt sich diese Reihenfolge jeweils umkehren.

3.3 Dateiinhalte anzeigen

- Auf keinen Fall für das reine Anzeigen von (großen) Dateien den `vi` verwenden (dauert sehr lange, kann System blockieren, erfordert doppelten Speicherplatz), sondern je nach beabsichtigter Tätigkeit die Tools `grep`, `head`, `tail`, `more`, `...` verwenden.
- `more FILE` bzw. `CMD | more` zeigt die Datei `FILE` bzw. die Ausgabe des Programms `CMD` aufgeteilt auf Bildschirmseiten an. Neben `more` gibt es weitere ähnliche bzw. sogar leistungsfähigere Anzeigeprogramme wie `pg` und `less`. `more` gibt am unteren Bildschirmrand aus, wieviel Prozent der Datei schon angezeigt worden sind. Folgende Kommandos erlauben ein schnelles Durchblättern der angezeigten Datei:

SPACE	page	Nächste Seite
RETURN	next	Nächste Zeile
b	back	Vorherige Seite (back)
q	quit	Anzeige beenden (quit)
h	help	Hilfe anzeigen (ausprobieren!)
/MUSTER	search	Muster vorwärts suchen
?MUSTER	search	Muster rückwärts suchen
n	search	Suche in gleiche Richtung wiederholen
N	search	Suche in andere Richtung wiederholen
//	search	Suche vorwärts wiederholen
??	search	Suche rückwärts wiederholen
v	vi	`vi` mit angezeigter Datei aufrufen

- `cat FILE(S)` gibt die angegebenen Dateien aneinandergehängt auf dem Bildschirm aus, mit dem Umlenkpfeil kann das Ergebnis in einer Datei abgelegt werden.
- `head/tail FILE(S)` gibt die ersten/letzten 10 Zeilen (oder n Zeilen bei Angabe der Option $-n$) einer Textdatei auf dem Bildschirm aus.
- Möchte man die Ausgabe eines Programms gleichzeitig auf eine Datei umlenken und am Bildschirm sehen, kann man entweder gleich beim Programmaufruf `tee` oder nachträglich `tail -f` verwenden:
  - ▷ `tee` dupliziert den *stdin* und gibt ihn zusätzlich auf die angegebene Datei aus (mit Option `-a` (*append*) wird an die Datei `FILE` angehängt):

    ```
    CMD ... | tee [-a] FILE
    ```

  - ▷ `tail -f` gibt das Ende der angegebenen Datei aus und versucht einmal pro Sekunde, in der Datei neu hinzugekommenen Text auszugeben:

    ```
    CMD ... > FILE &    # In den Hintergrund schicken
    tail -f FILE        # Auch in einem anderem Fenster
    ```

- Soll die Fehler-Ausgabe eines Programms ignoriert werden, kann sie auf `/dev/null` (**Null-Device**, auch *schwarzes Loch* genannt) umgeleitet werden. Diese spezielle Datei verschluckt einfach die Daten, die ihr geschickt werden:

  ```
  CMD ... 2> /dev/null
  ```

- Braucht man *mindestens eine Datei* in einer Dateiliste, kann `/dev/null` verwendet werden, beim ersten Leseversuch von dieser Datei ist sofort das Dateiende erreicht.

3.4 Suchen von Dateien mit bestimmten Inhalten

Generell braucht zum Suchen nach einem bestimmten Text nicht das gesamte Suchmuster, sondern nur ein genügend langer Teil davon angegeben werden (Zeitersparnis und Tippfehlervermeidung). In jedem Editor gibt es auch eine schnelle Möglichkeit, den letzten Suchbefehl zu wiederholen (z.B. im `vi n/N`=next).

- `grep` sucht in **Textdateien** nach einem angegebenen Text und gibt im Erfolgsfall den Dateinamen zusammen mit den passenden Textzeilen aus. Der Text läßt sich durch **Reguläre Ausdrücke** beschreiben, im einfachsten Fall ist das eine Folge von Zeichen. Das folgende Kommando sucht z.B. den Text `FuncName` in allen C-Dateien des aktuellen Verzeichnisses:

```
grep "FuncName" *.c
```

- Folgende **Meta-Zeichen** haben eine Sonderbedeutung, und müssen mit `\` gequotet werden, wenn sie *as is* gesucht werden sollen:

`.`	Beliebiges Zeichen
$x*$	0 oder mehr Auftreten des Zeichens x davor
$\backslash x$	Zeichen x quotieren
`[abc]`, `[a-z]`	Zeichen `abc`, `a-z`
`[^abc]`, `[^a-z]`	Alle Zeichen außer `abc`, `a-z`
`^`	Zeilenanfang
`$`	Zeilenende

- Kommt keines dieser Sonderzeichen und kein Zeichen, das für die Shell Sonderbedeutung hat, im Suchstring vor, muss er nicht unbedingt in `"..."` oder `'...'` gesetzt werden:

```
grep FuncName *.c
```

- `grep` kennt eine Reihe von nützlichen Optionen, mit denen die Suche beeinflußt werden kann:

`-c`	count	Nur die Anzahl der passenden Zeilen ausgeben
`-h`	hide	Dateinamen nicht ausgeben (bei mehr als einer Datei)
`-i`	ignore case	Groß-/Kleinschreibung ignorieren
`-l`	list	Nur die Dateinamen auflisten, nicht die Textzeilen
`-n`	number	Zeilennummer den passenden Zeilen voranstellen
`-v`	vice versa	Die *nicht* passenden Zeilen ausgeben

- Beispiele zum Aufruf von `grep`:

  ▷ Sucht alle Zeilen, die mit `#` beginnen (z.B. Präprozessorkommandos):

```
grep "^#" *.c
```

- ▷ Sucht alle Zeilen, die ein `#` enthlten, aber nicht am Zeilenanfang (z.B. fehlerhafte Präprozessorkommandos):

```
grep "^[^#].*#" *.c
```

- ▷ Sucht alle Leerzeilen:

```
grep "^$" *.c
```

- ▷ Sucht alle Zeilen, die keine Leerzeilen sind:

```
grep -v "^$" *.c
```

- ▷ Sucht alle Zeilen, die nur aus Leerzeichen ␣ und Tabulatoren `TAB` bestehen:

```
grep "^[ TAB]*$" *.c
```

- ▷ Sucht nach Zeilen, die ausschließlich aus Ziffern (mind. einer) bestehen:

```
grep "^[0-9][0-9]*$" *.c
```

- ▷ Sucht die LaTeX-Styledatei, die den Befehl `\timeofday` enthält:

```
grep "\\timeofday" /usr/local/lib/tex/inputs/*.sty \
                   /usr/local/lib/tex/inputs/local/*.sty
```

- ▷ Sucht im `makefile` nach einem Leerzeichen ␣ nach einem Backslash am Zeilenende:

```
grep "\\ $" makefile
```

- `strings` sucht aus beliebigen (auch binären) Dateien *alle* sinnvollen Zeichenfolgen heraus und gibt sie auf *stdout* aus. Ist z.B. unklar, in welcher der Systembibliotheken eine bestimmte Funktion `FuncName` steht, so kann diese Bibliothek durch folgende Anweisung ermittelt werden:

```
strings /usr/lib/* | grep _FuncName
```

  Hinweis: Der Unterstrich vor dem Funktionsnamen wird Systemfunktionen automatisch vom Compiler vorangestellt.

- Mit `strings` kann auch eine Liste aller Optionen, Fehlermeldungen und Textkonstanten (wie z.B. der Usage-Meldung) aus ausführbaren Programmen extrahiert werden.

3.5 Dateiinhalte sortieren/selektieren/bearbeiten

- `wc` (*word count*) zählt die Anzahl Zeichen, Worte und Zeilen in einer oder mehreren Dateien (und gibt bei mehreren Dateien ein Total aus). Folgende Optionen sind nützlich:

`-c`	character	Anzahl Zeichen zählen
`-w`	word	Anzahl Worte zählen
`-l`	line	Anzahl Zeilen zählen

- `sort` sortiert die Zeilen einer ASCII-Textdatei nach beliebig wählbaren Sortierkriterien (*binäre Daten können nicht sortiert werden*). Es sortiert standardmäßig **lexikographisch** beginnend mit dem 1. Zeichen jeder Zeile. Folgende Optionen sind nützlich:

`-b`	blank	Führende Leerzeichen und Tabulatoren in Feldern ignorieren
`-f`	fold	Groß-/Kleinschreibung ignorieren
`-n`	numeric	Arithmetisch sortieren (Std: lexikographisch)
`-o`$file$	output	Auf Datei $file$ ausgeben (Std: stdout, gleiche Datei erlaubt)
`-r`	reverse	Sortierung umkehren (Std: aufsteigend)
`-t`c	terminator	Feldtrennzeichen ist c (Std: Tabulator)
`-u`	unique	Identische Zeilen nur 1x ausgeben
+n	skip to field	n Felder überspringen
−n	stop at field	Bis Feld n sortieren
+$n.m$	skip to field	n Felder + m Zeichen überspringen
−$n.m$	stop at field	Bis Feld n + m Zeichen sortieren

- `uniq` (*unique*) entfernt hintereinander vorkommende gleiche Zeilen aus einer (sortierten) Textdatei. Folgende Optionen sind nützlich:

`-c`	count	Anzahl gleicher Zeilen zählen
`-d`	duplicate	Nur mehrfach vorkommende Zeilen ausgeben
`-u`	unique	Nur einfach vorkommende Zeilen ausgeben
+n	skip to field	n Felder überspringen
−n	stop at field	Bis Feld n sortieren

- `tr` (*translate*) übersetzt Zeichen in andere Zeichen. Folgende Optionen sind nützlich:

`-c`	complement	Alle Zeichen außer den angegebenen ersetzen
`-d`	delete	Angegebene Zeichen löschen
`-s`	squeeze	Wiederholte Ausgabezeichen nur 1x ausgeben

- `cut` schneidet aus einer Textdatei beliebige Spaltenbereiche heraus. Folgende Optionen sind nützlich:

`-c`$list$	columns	Spalten auswählen ($list$=n,n−m,...)
`-d`c	delimiter	Zeichen c ist Feldtrennzeichen (Std: Tabulator)
`-f`$list$	fields	Felder auswählen ($list$=n,n−m...)

- `paste` fügt mehrere Textdateien zusammen, indem es die Zeilen mit gleicher Zeilennummer nebeneinander stellt. Folgende Optionen sind nützlich:

`-`	stdin	Datei ist stdin (mehrfach erlaubt)
`-d`$text$	delimiter	Liste von Spaltentrennern (Std: Tabulator)
`-s`	same	Zeilen einer Datei aneinanderhängen

- `join` fügt mehrere Textdateien zusammen, indem es Zeilen mit dem gleichen Schlüssel nebeneinander stellt. Folgende Optionen sind nützlich:

-an	alone	
-e$text$	empty	Leere Ausgabefelder durch $text$ ersetzen
-jn m	join	
-o$n.m$	output	
-tc	terminator	Feldtrennzeichen ist c (Std: Tabulator)

- `sed` (*stream editor*) führt mit Hilfe von Regulären Ausdrücken beliebige Such-/und Ersetzungsoperationen, Einfügungen und Löschungen von Zeilen in einer Textdatei durch. Siehe getrennte Beschreibung.
- `awk` (*Aho, Weinberger, Kernighan* sind die Autoren) ist eine mächtige C-ähnliche Interpreter-Sprache, die ebenfalls zur Verarbeitung von Textdateien geeignet ist und vor allem erweiterte Reguläre Ausdrücke, assoziative Arrays und Zeichenketten als eigenen Datentyp kennt. Siehe getrennte Beschreibung.
- `perl` (*practical extraction language*) ist ein Zusammenfassung der UNIX-Kommandos `sh`, `sed`, `awk`, `tr`, `grep`, `sort`, `uniq`, extrem ausdrucksfähige reguläre Ausdrücke, C-Bibliotheksfunktionen, UNIX-Systemaufrufen, Paketkonzept, Objektorientierung, Sicherheitsaspekte, usw. Diese Programmiersprache ist daher sehr mächtig und sehr umfangreich und ersetzt in zunehmenden Maße die klassischen Skript-Sprachen `sh`, `awk`, `sed` und sogar die Programmiersprache `C`.

3.6 Dateien und Verzeichnisse vergleichen

- `diff file1 file2` zeigt die Unterschiede zwischen 2 Dateien (auf Zeilenbasis) an. Zeilen, die nur in der ersten (linken) Datei `file1` vorkommen, sind mit `<`, Zeilen, die nur in der zweiten (rechten) Datei `file2` vorkommen, sind mit `>` gekennzeichnet. Eignet sich vor allem zum Vergleich einer geänderten Quelldatei mit dem Original, um Änderungen herauszufinden. Beispiel:

```
file1 enthält:          file2 enthält:
    line1                   add1
    line2                   line1
    line3                   line4
    line4                   add4
                            add5
```

Das Kommando `diff file1 file2` ergibt dann:

```
0a1
> add1
2,3d2
< line2
< line3
4a4,5
> add4
> add5
```

Unter UNIX sieht `diff` schon einen Unterschied, wenn ein zusätzliches `Strg-M` am Zeilenende steht (vom DOS-Editor eingesetzt), unter DOS wird dieser Unterschied ignoriert.

- `dircmp DIR1 DIR2` (oder auch `diff` mit der Option `-r`) zeigt die Unterschiede zwischen zwei Verzeichnisse an. Es werden alle Dateien getrennt aufgelistet, die:
  - ▷ Nur im ersten Verzeichnis enthalten sind (`Only in ...`)
  - ▷ Nur im zweiten Verzeichnis enthalten sind (`Only in ...`)
  - ▷ Im ersten und zweiten Verzeichnis enthalten und verschieden sind (`... differ`)
  - ▷ Im ersten und zweiten Verzeichnis enthalten und gleich sind (`... are identical`)

  Beispiel (die mit `*` gekennzeichnete Datei ist verschieden):

```
dir1 enthält:          dir2 enthält:
    file1                  file2
    file2                  file3 *
    file3 *                file4
```

  Das Kommando `dircmp dir1 dir2` ergibt dann:

```
Only in dir1: file1
Files dir1/file2 and dir2/file2 are identical
Files dir1/file3 and dir2/file3 differ
Only in dir2: file4
```

3.7 Ausdrucken

- `lp FILE` oder `CMD ... | lp` (*line print*) druckt den Text der Datei `FILE` oder die Ausgabe des Kommandos `CMD` auf dem Standarddrucker aus. Mit der Option `-h` (*head*) wird die Ausgabe der Kopfseite unterdrückt.
- `lpstat` (*line print status*) liefert eine Übersicht über die in der Druckerwarteschlange stehenden Dateien, ihren Besitzer und ihre Jobnummer.
- `cancel NR` entfernt (nur eigene) Dateien aus der Druckerwarteschlange (die Jobnummer `NR` wird von `lpstat` ausgegeben).
- Statt der obigen drei Kommandos `lp`, `lpstat` und `cancel` werden in BSD-UNIX-Systemen (und auch in Linux) die drei Kommandos `lpr`, `lpq` (*queue*) und `lprm` (*remove*) verwendet.
- Mit `man -t CMD` lp— (*manual troff*) kann die Beschreibung zu dem UNIX-Kommando `CMD` ausgedruckt werden.
- `pr` (*print*) bereitet Dateien zum Ausdrucken auf, indem es Tabulatoren durch wählbar viele Leerzeichen ersetzt, eine Seitenaufteilung vornimmt, die Zeilen und Seiten durchnumeriert, Seitenköpfe und -füße erzeugt, usw. (bitte die Manualseite ansehen).

4 Tools

4.1 Vi

Grundkenntnisse des `vi` (*visual editor*) werden vorausgesetzt, eine ausführliche Beschreibung ist ebenfalls verfügbar. Zum `vi` gibt es auch mehrere Schnellreferenzen, die auf ein/zwei Seiten sämtliche Kommandos beschreiben.

- Beim Start wird nach einer Datei namens `.exrc` (oder `.vimrc` beim Vim) im `HOME`-Verzeichnis und im aktuellen Verzeichnis gesucht, und ihr Inhalt vom `vi` ausgewertet.
- Beim Start wird auch die Umgebungsvariable `EXINIT` ausgewertet. Diese sollte folgendes Kommando beinhalten:

  ```
  setenv EXINIT "source $HOME/.viinit"
  ```

  das die Datei `.viinit` aus dem Hauptverzeichnis einliest. Dort können beliebige `vi`-Einstellungen, Befehls- und Makrodefinitionen durchgeführt werden, wie z.B:

  ```
  set ts=4                # tab stop 4 Zeichen
  set sw=4                # shift width 4 Zeichen
  set directory=/tmp      # Verzeichnis für temporäre Dateien
  set shell=/bin/sh       # Standard-Shell
  ```

4.1.1 Nützliche Optionen des Vi

- Zeilennumerierung an-/ausschalten:

  ```
  :set nu      # number
  :set nonu
  ```

- Anzeige von `TAB` (als `^I`) und Zeilenenden (als `$`) an-/ausschalten:

  ```
  :set list
  :set nolist
  ```

- Anzeige der korrespondierenden öffnenden Klammer während der Eingabe einer schließenden Klammer an-/ausschalten:

  ```
  :set sm      # showmatch
  :set nosm
  ```

- Automatischen Zeilenumbruch bei den letzten 10 Zeichen einer Zeile ein- oder ausschalten:

  ```
  :set wrapmargin=10
  :set wrapmargin=0
  ```

- Aktuellen Eingabemodus in der untersten Bildschirmzeile rechts anzeigen:

```
:set showmode
:set noshowmode
```

- Die Tabulatorweite kann folgendermaßen auf 4 Zeichen eingestellt werden:

```
:set ts=4    # tab stop
:set sw=4    # shift width
```

4.1.2 Nützliche Kommandos des Vi

- Das *aller*letzte Kommando kann mit `u` (*undo*) zurückgenommen werden (auch globales Suchen und Ersetzen).
- *Sämtliche Änderungen* in der aktuellen Zeile können mit `U` (*undo*) zurückgenommen werden (solange die Zeile nicht verlassen wurde).
- Das letzte (Änderungs)Kommando kann mit `.` wiederholt werden.
- Das letzte Suchkommando kann mit `n` (*next*) in der gleichen, mit `N` (*Next*) in der umgekehrten Richtung wiederholt werden.
- Sprung zum Dateiende erfolgt mit `G` (*Goto*).
- Sprung zum Dateianfang erfolgt mit `1G`.
- Sprung zur korrespondierenden Klammer `( { [` bzw. `] } )` erfolgt mit `%`.
- Groß/Kleinschreibung eines Zeichen vertauschen erfolgt mit `~`.
- Zwei Zeichen vertauschen erfolgt mit `xp` (Cursor steht auf dem linken Zeichen).
- Zwei Zeilen vertauschen erfolgt mit `ddp` (Cursor steht auf der oberen Zeile).
- Verlassen der aktuellen Datei ohne Abspeichern mit `:q!`.
- Zwischen Funktionsköpfen kann per `[[` und `]]` hin- und hergesprungen werden (sofern sie am Zeilenanfang stehen und ihr Körper eingerückt ist).
- Mehrere Dateien können durch `vi FILES` editiert werden. Zur nächsten Datei kann mit `:n` geschalten werden. Wechsel zwischen aktueller und vorheriger Datei durch `Strg-6` oder `:e#`. Alle angegebenen Dateien können durch `:args` angezeigt werden. Zur ersten Datei kann mit `:rew` zurückgesprungen werden.

4.1.3 Weitere nützliche Kommandos des Vi

- Sprung zu einer C-Funktion (nur möglich falls mit `ctags *.c` die Datei `tags` angelegt wurde):

  ```
  :tag FuncName
  ```

  Oder auch auf den Anfang des Funktionsnamens stellen und `Strg-+` eintippen (*telnet* funkt da leider dazwischen, wie schaltet man das ab ???).

- Ein Kommando `CMD` an die Shell absetzen (z.B. `co -l %` oder `ctags *.c`):

  ```
  :!CMD
  :!!     # wiederholt letztes Kommando
  ```

- Das Ergebnis eines Kommandos `CMD` in den Text aufnehmen (nach der aktuellen Zeile):

  ```
  :r!CMD ...
  ```

- Den ganzen Text durch `CMD` verarbeiten und durch das Ergebnis ersetzen (kann durch `u` zurückgenommen werden):

  ```
  1G
  !GCMD
  ```

- Den Inhalt einer Datei `FILE` *nach* der aktuellen/ersten Zeile einfügen mit:

  ```
  :r FILE
  :0r FILE
  ```

- Der `vi` kann durch `Strg-Z` in den Hintergrund geschickt werden, man befindet sich dann wieder in der Shell. Mit `fg` (*foreground*) kommt man wieder in die unterbrochene Editorsitzung zurück (bitte nicht vergessen, sonst gehen beim Ausloggen die Editoreingaben verloren!).

4.2 Weitere Utilities

- `touch FILE` aktualisiert das Änderungdatum der Datei `FILE`, ohne sie zu verändern. Existiert `FILE` noch nicht, wird eine *leere Datei* mit diesem Namen erzeugt.
- `cal` (*calendar*) gibt den aktuellen Monat aus.
- `date` gibt das aktuelle Datum und die aktuelle Uhrzeit aus.
- `time CMD ...` oder `/usr/5bin/time CMD ...` messen die echte Ausführungszeit eines Programms `CMD` und geben sie am Ende des Programms aus.
- `smiley` produziert zufällige, quergestellte Gesichter.
- `banner TEXT` stellt den `TEXT` in Großschrift dar.

5 Programmierung

- Mit dem `vi` kann die Anzeige von Tabulatoren (`^I`) und Zeilenenden (`$`) durch die Option:

  ```
  :set list
  :set nolist
  ```

  ein- bzw. ausgeschalten werden.

- Am Anfang/Ende eines Quelltexts die Klammersequenzen `{[(` bzw. `}])` in Kommentar setzen. Dann kann per Editor-Kommando *Sprung zur korrespondierenden Klammer* (`%` im `vi`) schnell überprüft werden, ob die Klammernsetzung keine groben Fehler enthält.

- `ctags *.c` erzeugt eine Datei names `tags`, in der alle Funktionen und Makros der angegebenen C-Quellen alphabetisch sortiert in folgender Form aufgelistet werden:

  ```
  FuncName TAB FileName TAB /Suchmuster/   # TAB=Tabulator-Zeichen
  ```

- Um eine alphabetisch sortierte Liste aller Funktionen in den angegebenen Quellen zu erhalten, kann man folgendes Kommando angeben:

  ```
  ctags *.c
  cat tags | sed 's/TAB.*//' | uniq   # TAB=Tab-Taste
  ```

  Der `sed`-Aufruf (*stream editor*) ersetzt ab dem ersten `TAB` in einer Zeile den Text bis zum Zeilenende durch nichts, `uniq` entfernt hintereinander stehende doppelte Zeilen.

- `cflow *.c` erzeugt eine Beschreibung der Aufrufhierarchie der Funktionen in den angegebenen C-Quellen, und gibt sie auf *stdout* aus (nicht auf allen Maschinen verfügbar).

- `indent IN OUT` formatiert eine C-Quelle nach (definierbaren) Regeln in ein einheitliches Format um. Kennt sehr viele Schalter, die es aus der Datei `indent.pro` (profile) im aktuellen Verzeichnis liest. Eine sinnvolle Konvention ist (nicht auf allen Maschinen verfügbar):

  ```
  -bad -bap -bbb -cli1 -i4 -l80 -nce -nfc1
  ```

- In `makefile`-Makros zählen alle Leerzeichen, d.h. folgendes Makro hat vor und hinter dem Ersetzungstext jeweils ein Leerzeichen (Leerzeichen durch _ gekennzeichnet):

  ```
  FILE=_test_
  EXE=.exe
  ```

  Achtung: Soll der Inhalt von `FILE` mit dem von `EXE` konkateniert werden, so steht das Leerzeichen dazwischen:

  ```
  $(FILE)$(EXE)   # -> "_test_.exe" und nicht "test.exe"
  ```

- UNIX-Makefiles (sowie Awk- und Shell-Skripten) dürfen kein `Strg-M` und kein `Strg-Z` enthalten, wie es von DOS-Editoren als Zeilendende/Dateiende eingefügt wird. C-Quellen dagegen dürfen das `Strg-M` enthalten, es wird vom Compiler ignoriert. Zum Entfernen von `Strg-M` und `Strg-Z` aus Dateien kann unter UNIX folgendes Skript (`rmcr.sh`) verwendet werden:

```
#!/bin/sh
for f in "$@"; do
    echo -n "converting $f..."
    tr -d '[\015\034]' < $f > $$.tmp
    if [ "$?" -eq "0"]; then
        echo "done"
        mv $$.tmp $f
    else
        echo "error"
    fi
done
```

- Um `make` dazu zu bringen, eine Quelldatei `FILE` neu zu übersetzen, muss sie nicht in den Editor geladen und gespeichert werden. Entweder löscht man die zugehörige Objektdatei (`xxx.o`, nicht `xxx.c`), oder man gibt `touch FILE` an, um das Datum der Datei `FILE` zu aktualisieren.

- Verschiedene Maschinen `HOST` laufen eventuell unter verschiedenen Betriebssystemversionen oder C-Bibliotheken. Dies kann zur Folge haben, dass Programme mit Debuginformationen (Schalter `-g`), die auf dem einen Rechner erstellt wurden, auf dem anderen Rechner nicht lauffähig sind. Programme ohne Debuginformationen sind auf beiden Rechnern lauffähig. **Achtung:** Eingebundene Bibliotheken können Debuginformationen enthalten.

- Gemäß UNIX-Konvention sollte jedes Programm (mit der Anweisung `exit(n)`) einen **Exit-Status** zurückliefern. Dieser Wert besagt, dass das Programm erfolgreich durchgeführt werden konnte (0) oder ob irgendein Fehler auftrat ($\neq 0$), der Wert kann die Fehlerart näher beschreiben. Dieser Exit-Status kann von Shell-Skripten (über `$?`) abgefragt werden.

- Ausgaben von Programmen, die nur für Debugzwecke oder zur Anzeige von Fehlermeldungen gedacht sind, aber nichts mit dem eigentlichen Output zu tun haben, sollten auf *stderr* ausgegeben werden. Sie können so wahlweise gemeinsam mit oder getrennt von den Ausgaben auf *stdout* in eine Datei umgeleitet werden.

  Die Usage-Meldung eines Programms sollte *nicht* auf *stderr* ausgegeben werden.