Dateien lesen / schreiben

```
$zeile = <STDIN>;     # Std-Eingabe (1 Zl.)
@zeilen = <STDIN>;    # Std-Eing. (alle Zl.)
print "text";         # Standard-Ausgabe
print STDERR "bla";   # Standard-Fehler

printf "%s %d %f", "abc", 12, 3.14;

@zeilen = <>;   # UNIX-Filter-like (Diamond)

$ok = open(INP, "< eingabe");
$erste = <INP>;
foreach $zeile (<INP>) {...}
while (<INP>) {...} # $_ statt $zeile
close(INP);

open(OUT, "> vorherlösch");
print OUT "text";
close(OUT);
open(OUT, ">> anhäng");
```

Von/auf Prozesse lesen/schreiben

```
open(INP, "KMDO |"); # von KMDO lesen
open(OUT, "| KMDO"); # in KMDO schreiben
```

Dateien und Ordner

```
@files = glob "*";       # Filename-
@baks  = glob "*.bak";   # Globbing

unlink("weg", @old);     # Löschen
rename("alt", "neu");    # Umbenennen

-e DATEI   # Datei existiert (exists)
-z DATEI   # Datei hat Länge 0 Byte (zero)
-s DATEI   # Grösse der Datei (size)
-f DATEI   # Normale Datei (file)
-d DATEI   # Verzeichnis (directory)
-r DATEI   # Datei lesbar (readable)
-w DATEI   # Datei schreibbar (writable)
-x DATEI   # Datei ausführbar (executable)
-M DATEI   # Dateialter in Tagen (modified)
```

Bezeichner: *GROSS/Klein + getr. Namensraum*

$a ≠ $A ≠ @a ≠ @A ≠ %a ≠ %A ≠ &a ≠ &A

Pattern Matching und Substitution

```
$a = "ein böses Mädchen";
if ($a =~ m/bös/) {       # match
    print "gefunden";
}
$a =~ s/bös/gut/;         # substitute
# danach $a eq "ein gutes Mädchen"
```

Reguläre Ausdrücke (Regex)

`.`	Jedes Zeichen (*außer* `\n`)
Z`*`	Zeichen *Z* beliebig oft (*auch 0-mal*)
Z`+`	Zeichen *Z* ein- oder mehrmals
Z`*?` *Z*`+?`	Wie `*`/`+` (*matcht möglichst wenig*)
Z`?`	*Z* null- oder einmal
Z`{n,m}`	*Z* `n` bis `m`-mal
`^` `$`	Anfang / Ende des Strings
`\d` `\D`	Ziffer / keine Ziffer (*0-9 digit*)
`\w` `\W`	Wortzeichen / kein W. (*0-9_a-zA-Z*)
`\s` `\S`	Leerraum / kein Leerr. (*space*)
`\b` `\B`	Wortgrenze / keine Wortg. (*break*)
`[abc]`	Eines der Zeichen `abc`
`[^abc]`	Keines der Zeichen `abc`
`(...)`	Merken für später
`$1` `\1`	Was in 1. / 2. / ... Klammer war

Regex Beispiele

```
s/^\s#.*//          # Kommentar entf.
s/<\/?B>//          # <B> + </B> entf.
s/<\/?FONT.*?>//    # FONT-Tags entf. (v1)
s/<\/?FONT[^>]*>//  # FONT-Tags entf. (v2)
```

Regex-Trick: Andere Zeichen statt //

```
$a = "<H2><B>hallo</B></H2>";
$a =~ s@</B>@@;     # einzeln
$a =~ s(</B>){};    # paarweise
```

Modifizierer

```
$a =~ s/bla/was/g; # Mehrfach (global)
$a =~ s/bla/was/i; # Groß/Klein egal
$a =~ s/bla/was/s; # Singleline (\n in .)
$a =~ s/bla/was/m; # Multiline (n* \n)
$a =~ s/bla/was/x; # Extended (Komment.)
```


Perl-Referenzkarte V2.0

OSTC GmbH (http://www.ostc.de)

$kalar (Zahl oder String/Zeichenkette)

```
$n = 1;          $m = 314.1516E-2;
$s = "hallo";    $t = "Hallo";
```

@rray (Liste)

```
@a = (1, "abc", undef);
@b = qw/ Tom Hans Rick /; # quote word
@c = ("A" .. "Z") oder (1 .. 99);
```

Array-Elemente

```
$a[0]=1; $a[1]="abc"; $a[2]=undef;
```

Array-Länge/-Index (*hier: 3 bzw. 2*)

```
$anz = @a; oder @a+0   # Skalarer Kontext!
$anz = scalar @a;      # Analog
$last = @#a;           # Index letztes El.
```

Array-Operationen

```
foreach (@a) {...}          # $_
foreach $elem (@a) {...}    # $elem
push(@a, "ende");    $e = pop(@a);
unshift(@a, "anf"); $a = shift(@a);
@part = splice(@a,1,2); # Hans Rick
@b = sort(@a);     @b = reverse(@a);
@a = split(/:/, "aa:bb:cc");
$s = join("-", @a);    # "aa-bb-cc"
```

Hash (%, Assoziatives Array)

```
%age = ("Tom" => 18, "Hans" => 52);
$age{"Rick"} = 36;
if (exists($age{"Hans"})) {...}
delete $age{"Rick"};
```

Hash-Schlüssel / -Werte (*ungeordnet!*)

```
@k = keys %age;
@v = values %age;
```

Schleife über Hash-El. (sortiert + 2* unsort.)

```
foreach (sort keys %age) {...}
foreach (keys %age) {...}
while (($k,$v) = each(%age)) {...}
```

Mathematik

```
$c = $a + $b;  $c += 2;
$c = $a - $b;  $c -= 2;
$c = $a * $b;  $c *= 2;
$c = $a / $b;  $c /= 2;
$rest = $a % $teiler;        # Modulo
$c = $a ** $b; $c **= 2;     # Potenz
++$c; $c++; --$c; $c--       # Inc/Dec
```

Strings

```
$text = $n . " Euro"; # Dot-Operator
$text .= "mehr text"; # Verketten

$muster = "-" x 80;   # Times-Operator
$muster x= 10;        # Vervielfachen

chomp($text);         # Entfernt "\n"
chop($text);          # Entf. letztes Zei.
length($text);        # Textlänge

$viel <<"EOF"         # Here-Operator
Text mit " und Variablen $a und
Zeilenumbrüchen bis EOF alleine.
EOF;                  #  ";" nicht vergessen!
```

Vergleiche

```
if ($s eq "Hallo")    # Strings
if ($n == 12345)      # Zahlen
```

Analog: **ne gt ge lt le cmp** *# 1/0/-1*
!= > >= < <= <=> *# Space-Ship*

Funktions-Aufruf (&)

```
print &FUNC(10, "Text");
```

Reihenfolge von Definition und Aufruf egal.

Funktions-Definition

```
sub FUNC {
    my ($nr, $text) = @_;
    KMDO; ...
    return $ergebnis;
}
```

*Parameter "namenlos" und "call-by-reference" in Array @_ übergeben (**my** = lokale Variable)*

Verzweigungen

```
if (BED1) {
    KMDO1;
} elsif (BED3) {   # beliebig oft
    KMDO3;
} else {
    KMDOn;
}

unless (BED) {     # if (!BED) ...
    KMDO1;
} else {
    KMDO2;
}
```

Bedingungen / Schleifen als Modifikatoren

```
KMDO if (BED);     # Klammer darf fehlen
print "Hier" if ($debug);
KMDO unless (--$zaehler == 0);
KMDO foreach (@person);
KMDO while ($zaehler != 0);
KMDO until ($zaehler == 0);
```

Klammern { } und ( ) nur hier weglassbar!

Bedingter Ausdruck (Dreiwertiger Operator)

```
$erg = ($x == 1) ? "An" : "Aus";
```

entspricht

```
if ($x) {
    $erg = "An";
} else {
    $erg = "Aus";
}
```

Boolesche Werte und Operatoren

False = `0`, `"0"`, `""`, `()`, `undef` *(5 Werte!)*

or (||) *und* **and** (&&) *liefern Wert des letzten ausgewerteten Ausdrucks,* **not** *(**!**) negiert Wert*

```
$plz = parm{"plz"} or 90425;
$dbg = parm{"dbg"} and parm{"lev"};
open(IN, $file) or
    die("$file nicht offen: $!\n");
```

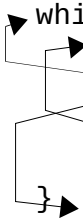

Zählschleifen

```
foreach $v (LISTE) { # $v enth. Elem.
    KMDO;
}

foreach (split(/\s/, $text)) {
    KMDO;            #  $_ enth. Worte
}

foreach $i (1..10) { # Bereichs-Operator
    KMDO;
}

for ($i = 1; $i < 10; ++$i) {
    KMDO;      # Analoge Zählschleife von 1..10
}
```

Bedingte Schleifen (kopf/fußgesteuert)

```
while (BED) {          until (BED) {
    KMDO;                  KMDO;
}                      }

do {                   do {
    KMDO;                  KMDO;
} until (BED);         } while (BED);
```

Schleifenabbruch/wiederholung (innerste)



```
while (BED) {
    KMDO;
    next if BED;    # continue in C
    last if BED;    # break in C
    redo if BED;    # nicht in C vorhanden
    KMDO;
}
continue { KMDO; } # kann fehlen
```

Sprungmarken (nur Großbuchstaben)

```
LABEL: KMDO;
goto / next / last / redo LABEL;
```

Programm-Abbruch

```
die("Fehler");      # "\n" od. Datei + Zl.nr
warn("Warnung");    # "\n" od. Datei + Zl.nr
exit(2);            # Exit-Status 0-255
```