3.19 Datei-Handles und Datei-Tests

3.19.1 Öffnen, Lesen und Schließen einer Datei

Ein **Datei-Handle** ist eine **Verbindung zur Außenwelt**, z.B. zu einer Datei. Wir haben bereits eines kennengelernt: `STDIN`.

Weiterhin gibt es die Datei-Handles `STDOUT` und `STDERR`. Ihre **Bedeutung** ist **wie in C** und sollte bekannt sein.

Außerdem hatten wir bereits den **Diamant-Operator** `<>` kennengelernt. Er ist eine **Kurzschreibweise für** `<ARGV>`, was genau die **bekannte Magie** hat.

Die Funktion `print` kann als **erstes Argument** ein **Datei-Handle** nehmen, dann geht die Ausgabe der folgenden Liste dorthin:
`print-handle.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print STDOUT "Normale Ausgabe.\n";
print STDERR "Ausgabe auf Fehler-Handle.\n";
```

▶▶ `./print-handle.pl`

▶▶ `./print-handle.pl >/dev/null`

▶▶ `./print-handle.pl 2>/dev/null`

Das gleiche geht **auch mit** `printf`.

Man beachte, daß das **Datei-Handle nicht** von einem **Komma gefolgt** ist, damit es nicht Teil der Liste wird.

`STDOUT` wäre im obigen Beispiel das **Default** und könnte auch weggelassen werden.

Diese drei Datei-Handles sind **automatisch offen**. Man kann aber **weitere öffnen**:
`open-ex.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
open TEXT, "redo-loop.txt";
close TEXT;
#
my $filename = "redo-loop.txt";
open TEXT, "< $filename";
open AUSGABE, "> ausgabe.txt";
while(<TEXT>) {
  print AUSGABE $_;
}
close TEXT;
close AUSGABE;
#
open AUSGABE, ">> ausgabe.txt";
  print AUSGABE "P.S.: Diese Zeile wurde angehaengt.\n";
close AUSGABE;
```

Die Grundform von `open` hat **zwei Argumente**, ein Datei-Handle und einen Dateinamen.

Per Konvention werden Datei-Handles **immer groß** geschrieben. Da sie **nicht** von einem **Sonderzeichen eingeleitet** werden, ist so garantiert, daß sie nicht mit gegenwärtigen oder zukünftigen Schlüsselwörtern **kollidieren**.

Mit `close` wird ein offenes Datei-Handle wieder **geschlossen**.

Per **Default** wird eine Datei **zum Lesen** geöffnet. Man kann das aber **auch explizit** angeben indem man ein **<-Zeichen** vor den Dateinamen stellt. Das schützt vor Problemen, wenn der **Dateiname aus** einer **Variablen** kommt, die vielleicht mit einem >-Zeichen beginnt.

Mit dem >-Zeichen wird eine **Datei zum Schreiben** geöffnet. Sie wird dabei entweder **angelegt** oder ggf. **auf Länge Null** gekürzt.

Mit dem **doppelten** >-Zeichen wird eine Datei ebenfalls zum Schreiben geöffnet. Aber wenn die Datei existiert, wird an sie **angehängt**, anstatt sie auf Länge Null zu setzen. Das ist **nützlich für Log-Dateien**.

▶▶ `./open-ex.pl`

▶▶ `cat ausgabe.txt`

Diese ganze **Syntax** ist bewußt an die **Shell-Syntax** angelehnt.

Wenn vor oder nach dem **Dateinamen Whitespace** steht, wird dieser von `perl` automatisch **entfernt**. Das ist z.B. praktisch, wenn eine **Variable** mit dem Dateinamen noch ein **Newline am Ende** enthält. **Will man** das **nicht**, so muß man **vor** die **Variable** ggf. `./` setzen und **ans Ende** ein `\0`-Byte anhängen.

Das **Schließen** einer Datei-Handle **kann man** auch **weglassen**. Bei **Programmende** werden alle **automatisch** geschlossen. **Auch** bei einem **neuen Öffnen** wird das Datei-Handle ggf. erst geschlossen. Trotzdem ist das Schließen **nützlich**, um die **Puffer sofort rauszuschreiben**.

Datei-Handles werden **benutzt, wie** es schon von `<STDIN>` und `print STDOUT ...` **bekannt** ist.

Das **Öffnen** einer Datei **kann schiefgehen**, z.B. wg. eines falschen Dateinamens oder wegen fehlender Rechte.

Wenn man vom zugehörigen Datei-Handle **hinterher** zu **lesen** versucht, bekommt man sofort ein **Datei-Ende**. Wenn man zu **schreiben** versucht, wird die **Ausgabe weggeworfen**. Falls **Warnungen** eingeschaltet sind, werden wir allerdings gewarnt.

Aber man kann auch den **Return-Wert** von `open` **abprüfen**:
`open-unless.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $infilename = "redo-loop.txt";
my $success =
  open TEXT, "< $infilename";
unless($success) {
  print "Kann Datei '$infilename' nicht oeffnen!\n";
  exit 1;
}
my $outfilename = "ausgabe.txt";
my $success =
  open AUSGABE, "> $outfilename";
unless($success) {
  print "Kann Datei '$outfilename' nicht oeffnen!\n";
  exit 1;
}
while(<TEXT>) {
  print AUSGABE $_;
}
close TEXT;
close AUSGABE;
```

3.19.2 Programmabbruch mit `die`

Diese **Prüfung** geht aber noch **wesentlich eleganter**.

Perl kann ohnehin ein Programm **mit** einer **Fehlermeldung** auf STDERR **beenden**, wenn notwendig. Beispiel: **Division durch Null**.

Diese Funktionalität kann man mit der `die`-Funktion auch **selbst nutzen**. Außerdem ist der `or`-**Operator** hier sehr nützlich:
`open-die.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
open TEXT, "< xyzzy" or die "Can't open file 'xyzzy'";
while(<TEXT>) {
  print;
}
```

Die `die`-Funktion **druckt** ihr **Argument** und **beendet** das Programm dann mit einem **Nicht-Null-Exit-Code**.

Anmerkung: **Wichtig** ist hier die **niedrige Präzedenz** des `or`-Operators. Der `||`-Operator hätte hier eine **Auswahl zwischen** dem **Dateinamen und** der nachfolgenden **Funktion** gemacht, die immer zugunsten des Dateinamens ausgegangen wäre. Es wäre also effektiv **gar keine Prüfung** durchgeführt worden.

▶▶ `./open-die.pl`

Es wird die **aktuelle Zeilennummer** des `perl`-**Skripts** mit ausgegeben, und ggf. auch die Zeilennummern aller **offenen Eingabedateien**. Das ist gut fürs **Debugging**. Wenn der Fehler ggf. aber beim **Benutzer** liegt, will man das **nicht**. Dann sollte man ein **Newline** ans Ende der Meldung schreiben:
`open-die2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use English;
use strict;
open TEXT, "< xyzzy" or die "Can't open file 'xyzzy': $ERRNO\n";
while(<TEXT>) {
  print;
}
```

▶▶ `./open-die2.pl`

Wie wir sehen, wird wird die **Variable** `$ERRNO` durch die **Fehlermeldung des Betriebssystems** ersetzt. Es wird einfach **die letzte gesetzte** Fehlermeldung genommen. Falls **kein Fehler** auftritt, wird die Variable **nicht verändert**. Wir haben hier die Langform benutzt, die **Kurzform** wäre `$!`. Außerdem bekommen wir **durch** `use diagnostics` noch eine **ausführliche Erläuterung** des Fehlers.

Die Funktion `warn` ist eine **Variante** von `die`, die das Programm **nicht beendet**. Es wird **nur** eine **Warnmeldung** auf STDERR gedruckt.

3.19.3 Öffnen von Pipes

Man kann auch von **Pipes lesen** und auf Pipes **schreiben**:
`open-pipe.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
open INTEXT, "gunzip -c numbers.txt.gz |";
open OUTTEXT, "| gzip > numbers-new.txt.gz";
while(<INTEXT>) {
  s/7/sieben/g;
  print OUTTEXT $_;
}
```

▶▶ `./open-pipe.pl`

Ein **Pipe-Symbol am Ende** zeigt an, daß das vorstehende ein Kommando ist, das die **Eingabe** für uns **liefern** soll. Ein Pipe-Symbol **am Anfang** bedeutet, daß dieses ein Kommando ist, das unsere **Ausgabe aufnimmt**.

Das erste Kommando **dekomprimiert** die Eingabe, das zweite **komprimiert** sie wieder.

Wenn **Shell-Metazeichen** im Kommando vorkommen, wird eine Shell gestartet, um sie zu interpretieren. Beispiel hier: **Ausgabeumleitung**.

Dies alles gilt auch für das Öffnen des impliziten Datei-Handles `ARGV` des Diamant-Operators. Deshalb kann jedes `perl`-Programm, das den Diamant-Operator benutzt, automatisch auch die Eingabe aus einer Pipe holen. Auf Seite 62 hatten wir ein solches Programm vorgestellt.

▶▶ `view my-scope.pl`

▶▶ `./my-scope.pl "gunzip -c numbers.txt.gz |"`

Lesen und Schreiben auf **Pipes** hat allerdings einen **Nachteil**: Es ist **schwer festzustellen, ob** das Programm **erfolgreich gestartet** wurde. Meist bekommt man aus internen Gründen **keine Fehlermeldung**.

Mehr Informationen zum Datei-Öffnen gibt es in `man perlopentut`.

3.19.4 Datei-Tests

Oft muß man **prüfen, ob** eine **Datei existiert**:
`ftest-exist.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use English;
my $outname = 'output.txt';
die "File $outname already exists!\n"
  if -e $outname;
open OUTTEXT, "> $outname" or die "Can't open $outname: $ERRNO\n";
print OUTTEXT 'bla';
```

▶▶ `./ftest-exist.pl`

▶▶ `./ftest-exist.pl`

Ein Datei-Test **beginnt** immer **mit** einem **Minus**, auf das ein Buchstabe folgt.

Man kann sie auf **Dateinamen oder** auf **Datei-Handles** anwenden.

Es gibt eine ganze Reihe **weiterer** Datei-Tests:

Datei-Tests

Datei-Test	Bedeutung
`-r`	lesbar (readable)
`-w`	schreibbar (writable)
`-x`	ausführbar (eXecutable)
`-o`	gehört diesem User (owner)
`-e`	existiert (exists)
`-z`	existiert und hat Länge Null (zero)
`-s`	existiert und hat Länge größer Null, Wert ist Länge (size)
`-f`	normales File (file)
`-d`	Verzeichnis (directory)
`-l`	symbolischer Link (link)
`-t`	ist ein TTY [nur für Datei-Handles] (tty)
`-T`	Text-Datei, vermutlich (text)
`-B`	Binär-Datei, vermutlich (binary)
`-M`	Änderungs-Alter, in Tagen (modification)
`-A`	Zugriffs-Alter, in Tagen (access)
`-C`	Inode-Änderungs-Alter, in Tagen (change)
...	...

Kurs Unix-Tools, WS 01/02, Jan Bredereke 23

Die **Lese-/Schreib-Tests** prüfen **nur** die **Zugriffsbits**. Beim späteren Zugriff kann natürlich immer noch etwas **schiefgehen**.

Der **Normal-File-Test** `-f` ist **auch** wahr, wenn es sich um ein **symbolisches Link** auf ein normales File handelt.

Der **TTY-Test** kann **nur** auf **Datei-Handles** angewandt werden. Bei Files wäre er immer False. Er ist nützlich, wenn man z.B. von `STDIN` wissen will, ob das Programm **interaktiv** laufen kann.

Die **Text-/Binär-Tests öffnen** die **Datei** und schauen sich den Anfang an. **Frage: Wann** sind diese beiden Tests **nicht komplementär?** Wenn die Datei **nicht existiert** (beide False) oder wenn sie **leer** ist (beide True).

Die **Datei-Alter-Tests** geben eine **Fließkommazahl** zurück. Zwischen kurz vor **Mitternacht** und kurz nach Mitternacht liegt also ein Wert viel kleiner als 1.0.

Weiterhin messen die **Datei-Alter-Tests relativ** zu dem Zeitpunkt, an dem das `perl`-**Skript gestartet** wurde. Das ist **sinnvoll,** wenn man **relative Datei-Alter vergleichen** will. Dann sollte man den gleichen Bezugspunkt nehmen. Eine **Konsequenz** ist, daß man auch ein **negatives Alter** bekommen kann, wenn das Programm schon eine Weile läuft. Man kann den **Bezugszeitpunkt** aber auch **aktualisieren,** durch `$^T = time;`

Wenn der Datei-Test **keinen Parameter** hat, arbeitet er auf der **Default-Variaben** `$_`. **Vorsicht** dabei **mit Mehrdeutigkeiten** bei dem, was folgt. Will man z.B. die **Dateigröße** in Bytes **durch 1000 teilen**, könnte der **folgende Schrägstrich** auch als Dateiname gedeutet werden. Am besten setzt man **Klammern** um einen Datei-Test ohne Parameter.

Datei-Tests kosten relativ viel **Zeit**. Wenn man **mehrere Tests** auf der gleichen Datei oder dem gleichen Handle macht, kann man auf die **gepufferten Daten** des ersten Tests zurückgreifen. Dies geht über das magische Datei-Handle namens `_`. Beispiel:
`ftest-repeat.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use English;
foreach (@ARGV) {
  print "$_\n"
    if (-T) and -s _ > 300 and -M _ > 14;
}
```

▶▶ `./ftest-repeat.pl *`

Dies **findet** alle **Text**-Dateien im aktuellen Verzeichnis, die **länger als 300** Bytes sind und innerhalb der letzten **zwei Wochen nicht verändert** worden sind.

3.20 Module

3.20.1 Benutzung einfacher Module

Perl bietet **sehr viele vorgefertigte Module**. **Warum** sollte man sie **benutzen?**

Beispiel: basename-Program. Es soll den Dateinamen ohne den Verzeichnispfad davor liefern:
`basename-simple.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $name = shift @ARGV;
defined $name or die "Aufruf: basename-simple <dateiname>\n";
$name =~ s#.*/##;
print "$name\n";
```

▶▶ `./basename-simple.pl /usr/bin/perl`

Dieses Programm **prüft sogar** ab, ob ein **Parameter** angegeben ist.

Aber es hat **mehrere Probleme. Welche?**

Erstens: Ein **Verzeichnisname** könnte ein **Newline enthalten**:

▶▶ `./basename-simple.pl '/usr/strange` und ▶▶ `dir/bin/perl'` (d.h. mit Newline)

Anmerkung: **Mit** dem **Modifikator `/s`** knnte man dieses Problem **beheben**.

Zweitens: Dies ist **Unix-spezifisch**. `perl` läuft z.B. aber auch unter den Microsoft-Betriebssystemen, MacOS und VMS und AmigaOs. Alle haben andere Konventionen über Dateipfade (Backslash, zwei Doppelpunkte, eckige Klammern).

Drittens: Jemand hat dieses **Problem bereits einmal gelöst**.

Es gibt eine ganze Anzahl an **Standardmodulen,** die jeder `perl`-Distribution beiliegen. Und es gibt noch viel mehr Module im **Comprehensive Perl Archive Network (CPAN)**.

Liste der Standardmodule:

▶▶ `man perlmodlib` ab „Standard Modules“

Beispiel: `File::Basename`

Doku dazu:

▶▶ `perldoc File::Basename`

Damit sieht unser Programm so aus:
`basename-mod.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename;
my $name = '/usr/bin/perl';
my $basename = basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
fileparse_set_fstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";
```

Die **neuen** Funktionen werden **aufgerufen, als wären sie eingebaute** Funktionen.

▶▶ `./basename-mod.pl`

Jetzt ist das Programm **sofort nach VMS portabel**, wie man sieht.

(Anmerkung: Das File::Basename-Modul (und „Learning Perl") hat einen Bug: Newlines in Verzeichnisnamen behandelt es falsch, es fehlt der `/s`-Modifikator! Trotzdem...)

Das Modul File::Basename **exportiert** noch **einige weitere Funktionen**, die wir hier nicht benutzen, z.B. `dirname`.

Was ist, wenn wir **schon** eine **lokale Funktion** `&dirname` haben, z.B. für **„Direction-Name"**? Namenskollision!

Man kann auch **nur** eine **ausgewählte Liste** von Funktionen aus einem Modul **importieren**:

`basename-mod2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename qw/ basename fileparse_set_fstype /; # <-------- !
my $name = '/usr/bin/perl';
my $basename = basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
fileparse_set_fstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";
```

Was ist, wenn wir **nicht einmal die benötigten** Funktionen importieren wollen oder können? **Leere Liste importieren** und über den **vollen Namen** der Funktionen zugreifen:

`basename-mod3.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename qw/ /;
my $name = '/usr/bin/perl';
my $basename = File::Basename::basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
File::Basename::fileparse_set_fstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = File::Basename::basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";
```

Weitere Dokumentation zu Modulen findet man in `man perlmod` .

3.20.2 Einige wichtige Standardmodule

Einige wichtige Standardmodule

Modul	Beschreibung
Cwd	get pathname of current working directory
Fatal	make errors in builtins or Perl functions fatal
File::Basename	split a pathname into pieces
File::Copy	copy files or filehandles
File::Find	traverse a file tree
File::Path	create or remove a series of directories
File::Spec	portably perform operations on file names
Getopt::Long	extended processing of command line options
Getopt::Std	process single-character switches with switch clustering
I18N::Collate	compare 8-bit scalar data according to the current locale
Math::BigFloat	arbitrary length float math package
Math::BigInt	arbitrary size integer math package
POSIX	asin, cosh, floor, frexp, isupper, isalpha, creat, open, asctime, clock, . . .
Term::ReadLine	interface to various readline packages
Sys::Hostname	try every conceivable way to get hostname
Text::Tabs	expand and unexpand tabs per the Unix expand(1) and unexpand(1)
Text::Wrap	line wrapping to form simple paragraphs
Time::Local	efficiently compute time from local and GMT time

Kurs Unix-Tools, WS 01/02, Jan Bredereke 24

(Man bekommt die **vollständige Liste** der Standardmodule mit `man perlmodlib` .)

3.20.3 Das Comprehensive Perl Archive Network (CPAN)

Hier gibt es **viele weitere Module** zum **Herunterladen**.

Adresse: www.cpan.org

Man kann nach diversen Kategorien und Stichworten **suchen**.

Bei der Installation hilft `man perlmodinstall` .

Wenn man unter Unix ist, kann man das **Herunterladen und** das **Installieren** mit dem CPAN-**Modul automatisieren**. Siehe `perldoc CPAN` .

3.21 Objekt-Orientierung?

Gibt es in `perl`. Aber um sie einführen zu können, **braucht man** eine Reihe von **Konzepten**, die selbst **noch nicht eingeführt** worden sind. Beispiel: Referenzen.

Ein Tutorial findet man in `man perltoot`.

Ansonsten gelten die **Grundregeln**:

1. Ein **Objekt** ist eine **Referenz**, die weiß, **zu welcher Klasse** sie gehört.
2. Eine **Klasse** ist ein **Package**, das **Methoden liefert**, die mit Objekt-Referenzen umgehen können.
3. Eine **Methode** ist ein **Unterprogramm,** das eine Objekt-Referenz **als erstes Argument** erwartet (oder ein Package-Name).

Dies wird ausführlich in `man perlobj` erläutert.

3.22 Prozeß-Management

3.22.1 Die `system`-Funktion

Man kann aus einem `perl`-Programm heraus **andere Programme starten**. (Diese Programme sind dann natürlich **betriebssystem-spezifisch**.)
`system.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
system "date";
system 'ls -l $HOME/.skel';
system "xeyes &";
```

Die Funktion heißt `system` . Der erste Aufruf gibt das Datum mit der Unix-Funktion `date` aus.

Wenn man **Shell-Variablen** benutzt, sollte man in `perl` **einfache Anführungszeichen** nehmen, wie im zweiten Aufruf.

Wenn man **Unix-Shell-Metazeichen** verwendet, wie etwa das **Kaufmanns-Und** im dritten Aufruf, wird automatisch eine **Unix-Shell zur Ausführung** benutzt.

▶▶ `./system.pl`

Aber **Vorsicht**:
`system2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $filename = shift;
chomp $filename;
system "ls -l $filename";
```

▶▶ `./system2.pl 'system2.pl ; xeyes &'`

Die **Benutzereingabe liefert Shell-Metazeichen**. Die `system`-Funktion interpretiert sie. Bei entsprechenden Eingaben kann dies böse enden. Ggf. **auch aus Versehen**.

Man kann die Shell vermeiden, wenn man `system` **mit mehr als einem Argument** aufruft:
`system3.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $filename = shift;
chomp $filename;
system "ls", "-l", $filename;
```

▶▶ `./system3.pl 'system2.pl ; xeyes &'`

3.22.2 Ausgaben einfangen mit Backquotes

Man kann die **Standard-Ausgabe** eines **externen Programms** in eine Variable **einfangen**:
`backquote.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
chomp(my $systemdate = `date`);
print "Das System denkt, dass das Datum '$systemdate' ist.\n";
```

▶▶ `./backquote.pl`

Man beachte, daß `perl`, anders als die Shell, das **letzte Newline nicht selbsttätig entfernt**.

Die Backquotes **entsprechen** der **Ein-Argument-Form** von `system`, was die Interpretation von Shell-Metazeichen betrifft. Es gibt kein Gegenstück zur Mehr-Argument-Form.

Der **Standard-Input** des externen Programms **bleibt** mit dem Standard-Input des `perl`-Programms verbunden, wie bei der Shell-Version der Backquotes auch. **Ebenso** die **Fehler-Ausgabe**.

Man kann die Backquotes auch **im Listenkontext** verwenden:
`backquote2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my @w_list = `w`;
foreach (@w_list) {
  if( /^([^ ]+) +pts.{17} {6}.*/ ) {
    print "$1\n";
  }
}
```

▶▶ `./backquote2.pl`

Wir bekommen ein **Feld von Ausgabezeilen**. Wir **bearbeiten** es hier weiter, **wie** wir es **damals** mit `sed` getan haben. (Das `w`-Kommando liefert u.a. die Liste der **eingeloggten Benutzer**.)

3.23 Ausblick auf einiges fortgeschrittenes `perl`

Prozeß-Signale: Unix-Prozesse kennen sog. Signale, z.B. SIGINT (= Ctrl-C). Ein Unix-Programm kann sie z.T. **abfangen und darauf reagieren**. Ein `perl`-Programm kann das ebenfalls, indem es in dem Hash `%SIG` den Namen eines Unterprogramms als Wert für den Namen des Signals einträgt, also z.B. für `INT`.

Damit kann man z.B. auch gut **Temp-Dateien löschen** lassen.

Environment-Variablen: Unix-Prozesse haben Environment-Variablen, z.B. `$HOME` und `$PATH`. Mit dem Hash `%ENV` kann man darauf zugreifen.

Datenbanken: Man kann ein **Hash an eine Datenbank binden**. Damit bleibt der Inhalt über den Programmlauf hinaus erhalten. Eine **einfache Datenbank** wird mit `perl` **mitgeliefert**, aber man **kann auch andere nehmen**, wie Oracle, Sybase, Informix, mySQL und andere.

Editieren von Dateien am Platz von der Kommandozeile: Die Kommandozeilen-option `-i` bewirkt, daß die Dateien des Diamant-Operators `<>` am Platz editiert werden:
`in-place.sh`:

```
#! /bin/sh
perl -p -i.bak -w -e 's/Zeile/Textzeile/' redo-loop.txt
```

▶▶ `./in-place.sh`

Die Option `-p` erzeugt die **bekannte Schleife** um das Programm. Falls man **nur** `-i`, ohne `.bak`, verwendet, gibt es **keine Backup-Datei**.

Abfangen von Programmabbrüchen mit `eval` Wenn ein Programm z.B. bei einer **Division durch Null** nicht abbrechen soll, kann man die fraglichen Kommandos innerhalb eines eval-Blockes ausführen lassen. Die Variable `$@` enthält hinterher ggf. die Fehlermeldung, sonst ist sie leer.

Elemente einer Liste mit `grep` **auswählen:** Eine Kurzform zum effizienten Auswählen:

`grep.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use English;
{
  my @odd_numbers;
  foreach (1..50) {
    push @odd_numbers, $_ if $_ % 2;
  }
  print "@odd_numbers\n";
}
{
  my @odd_numbers = grep { $_ % 2 } 1..50;
  print "@odd_numbers\n";
}
{
  open FILE, 'redo-loop.txt' or die "Can't open: $ERRNO\n";
  my @matching_lines = grep { /\bZeile\b/i } <FILE>;
  print @matching_lines;
}
{
  open FILE, 'redo-loop.txt' or die "Can't open: $ERRNO\n";
  my @matching_lines = grep /\bZeile\b/i, <FILE>;
  print @matching_lines;
}
```

Die **erste Form** ist nicht falsch, aber etwas **ineffizient**. Der `grep`-**Operator** wählt aus einer Liste aus. Im Block wird `$_` **an das aktuelle Element gebunden**. Wenn der Block **True** ergibt, kommt das Element in die Ergebnisliste.

Der **Name** des Operators kommt vom entsprechenden Unix-Kommando. `perl`'s grep ist allerdings mächtiger. Die **dritte Form** zeigt einen entsprechenden Einsatz.

Es gibt auch eine **einfachere Syntax**, die die **vierte Form** zeigt. Wenn man nur einen einfachen Ausdruck braucht, kann man die geschweiften **Klammern weglassen** und ein **Komma** schreiben.

Elemente einer Liste transformieren mit `map`:
Ähnlich wie `grep`, aber es wird **nicht ausgewählt**, sondern **elementweise transformiert**:
`map.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use English;
my @text = qw/ Dies ist ein Test /;
my @neu_text = map { "\u$_" } @text;
print "@neu_text\n";
#
@neu_text = map "\u$_", @text;
print "@neu_text\n";
```

Der **Backslash-Escape** `\u` macht den nächsten Buchstaben zum Großbuchstaben.

Es gibt wiederum eine **Kurzform** ohne geschweifte Klammern, wie die zweite Version zeigt.

Der Ausdruck wird **im Listen-Kontext** evaluiert, so daß ein Original-Listenelement ggf. **mehrere Ergebnis-Listenelemente** ergeben darf.

Hash-Slices: **Genau wie** man ein Slice von einem **Array** machen kann, kann man auch ein Slice von einem Hash machen. Das **Ergebnis** ist ebenfalls eine **Liste**. Gekennzeichnet wird es durch **geschweifte Klammern** und einen **Klammeraffen** vor dem Hash-Namen. (Ein Array-Slice hat eckige Klammern und ein Dollar-Zeichen.)

Konstanten: Es gibt auch Konstanten:
`constant.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
use constant DEBUGGING => 0;
use constant ONE_YEAR  => 365.2425 * 24 * 60 * 60;

if(DEBUGGING) {
  # ...
}
# ...
```

Vorteile: **Sicherer.** Und es kann besser **optimiert** werden. Im Beispiel wird der **Debugging-Code** ggf. **vollständig wegoptimiert**.

Sicherheit: Man kann ein `perl`-Programm ziemlich sicher machen, z.B. durch **Datenflußanalyse** mit dem **`taint`-Mechanismus.** Dabei führt `perl` Buch, welche Daten aus der Umgebung kommen, so daß man ihnen nicht trauen darf. Wenn solche Daten einen anderen Prozeß, eine andere Datei oder ein anderes Directory beeinflussen, bricht `perl` die Verarbeitung ab. Siehe `man perlsec` .

Debugging: `perl` hat einen guten **eingebauten Debugger**. Er ist kommandozeilenorientiert und selbst in `perl` geschrieben. Siehe `man perldebug`.

Common Gateway Interface (CGI): Das `CGI`-Modul erlaubt, **HTML-Formulare** zu generieren und auszuwerten.

Die Verwendung setzt allerdings sehr gute `perl`- und Sicherheitskenntnisse voraus, da **Hacker-Angriffe** auf CGI-Skripten **sehr beliebt** sind.

Referenzen: Ähnlich wie **C-Pointer,** aber **sicherer**.
Siehe `man perlreftut` und `man perlref` .

Komplexe Datenstrukturen: Sind möglich, z.B. zweidimensionale Felder, oder ein Array von Hashes, oder ein Hash von Hashes, ...
Siehe `man perldsc` und `man perllol` .

Überladen von Operatoren: Mit dem `overload`-Modul. Z.B. für die Operatoren auf den komplexen Zahlen.

Einbetten von Dokumentation in `perl`-Quellcode: `perl`s **eigene Dokumentation** ist in den Quellcode eingebettet, im **pod-Format** (plain old documentation). So kann man auch **eigene Module dokumentieren**, und `perldoc` wird sie gleich mit anzeigen. Siehe `man perlpod` .

Graphische Benutzerschnittstellen (GUIs): Die Tk-Module bieten dies.