

---

# Perl Aufbaukurs

Dirk Rüdiger

[dirk.ruediger@partmaster.de](mailto:dirk.ruediger@partmaster.de)



# Seminarfahrplan

---

1	Funktionen und Subroutinen . . . . .	4
2	Fortgeschrittenes Arbeiten mit Dateien . . . . .	31
3	Referenzen und verschachtelte Datenstrukturen . . . . .	41
4	Packages, Module und Pragmas . . . . .	56
5	Dokumentation des Perl-Programmtextes . . . . .	84
6	Testen in Perl . . . . .	96
7	Programmieren in Perl – ein Styleguide . . . . .	112
8	Perl Tipps . . . . .	124

# Auf nach Perl

---

- Perl wurde 1986 von Larry Wall entwickelt, es fand mittlerweile viele Freunde und eine große Entwickler-Gemeinde
- Perl ist eine Interpreter-Sprache, aktuell ist Version 5.10
- Distributionen sind für eine Vielzahl von Betriebssystemen verfügbar
- Das Hauptquartier: <http://www.Perl.com>
- Für Windows:
  - Cygwin Project: <http://www.cygwin.com/>
  - ActivePerl von ActiveState: <http://www.ActiveState.com/>
- Funktionserweiterungen für viele Anwendungsgebiete als Module Wiederverwendbar
- Comprehensive Perl Archive Network (CPAN) stellt große Auswahl an Modulen bereit

---

# Funktionen und Subroutinen

- Parameterübergabe an Funktionen
- Ausführen von anderen Programmen
- Eigene Subroutinen: Parameterübergabe und Rückgabewerte
- Lokalisieren von Variablen
- Fehlerbehandlung

# Parameterübergabe an Funktionen

---

- Funktionen haben keine formalen, benannten Parameter
- Es können beliebig viele Parameter übergeben werden
  - Übergabe erfolgt in Form einer Argumente-Liste
  - Interpretation der Argumente-Liste erfolgt in der Funktion
  - Somit ist flexible Parameterübergabe möglich

```
print $name, $alter, @kinder, "\n";  
print "$name_ $alter_ @kinder\n";  
calculateArea(4, 3);          # =4*3;  
calculateArea(4, 3, 1, 2); # =4*3-1*2;
```

# Parameterübergabe an Funktionen

---

- Array oder Hash in der Argumentliste wird aufgelöst
- Referenzen auf Arrays oder Hashes werden auch als Referenz an die Funktionen übergeben (*Pass by Reference*)

```
$number = 25;
```

```
@array = (2, 3, 4);
```

```
mySub($number, @array); # ist mySub(25, 2, 3, 4);
```

```
mySub($number, \@array); # ist mySub(25, (2, 3, 4));
```

# Eigene Subroutinen

---

- Subroutinen (eigene Funktionen) sind benannte Anweisungsblöcke
- Sie dienen zur Erhaltung der Übersichtlichkeit der Programmstruktur sowie für Wiederverwendbarkeit von Programm-Code
- Perl unterscheidet syntaktisch nicht zwischen den Bezeichnungen Unterprogrammen, Funktionen, Subroutinen, Prozeduren etc.
- Subroutinen haben eigenen Namensraum
  - Namen beginnen immer mit &
  - Präfix kann bei Eindeutigkeit weg gelassen werden
- Subroutinen-Aufruf kann vor dessen Deklaration erfolgen

# Auswerten der übergebenen Parameter

---

- In Subroutine enthält
  - `@_` eine Liste aller Argumente der Funktion
  - `$_` jeweils das erste Element der Argumente-Liste

- Die Funktionsaufrufe

`$value = &func; # und`

`$value = func(); # oder: $value = func;`

erscheinen identisch, jedoch

- Gibt `&func` den Zugriff auf den Inhalt von `@_` des Aufrufers frei
- Stellt `func()` ein neues, leeres `@_` bereit

# Definieren von Vorgabewerten

---

- In Funktionen können lokale Variablen mit Vorgabewerten initialisiert werden
  - \$a mit \$b initialisieren, wenn \$b == 0 nicht zulässig ist  
`$a = $b || "13"; # wenn ($b == undef || $b == 0)`
  - \$a mit \$b initialisieren, wenn \$b == 0 zulässig ist  
`$a = defined($b) ? $b : "13"; # wenn ($b == undef)`
  - Übergeben eines Vorgabewertes, falls \$a noch nicht gesetzt wurde  
`$a ||= "13"; # wenn ($a == undef || $a == 0)`
  - \$a mit dem ersten "wahren" Wert initialisieren  
`$user = $ENV{USER} || $ENV{LOGNAME} || getlogin();`
- Anmerkung: Übergeben eines Wertes an eine Variable, wenn diese noch keinen Wert hat erfolgt mit ||

# Rückgabewerte von Funktionen

---

- Subroutinen geben immer ein Ergebnis an das aufrufende Programm zurück

```
return $wert;
```

- Implizit wird das Ergebnis der letzten Anweisung zurückgegeben
- Rückgabewert kann Variable beliebigen Typs sein

```
return ( $scalar , \@array , \%hash );
```

- Ein Fehlerstatus wird zurückgegeben, indem kein Wert übergeben wird.

```
return;
```

Dies kann im aufrufenden Programm geprüft werden:

```
if ( @array = func() ) { ... }
```

# Rückgabewerte von Funktionen

---

- Anmerkung zum Fehlerstatus:

- `return`; gibt einen undefinierten Wert zurück und entspricht der Anweisung

```
return (wantarray ? () : undef);
```

- Bei der Auswertung in der aufrufenden Anweisung

```
if (@array = func()) { ... }
```

wird dann im Array-Kontext `@array = ()` und im skalaren Kontext als Anzahl der Elemente (`=0`) ausgewertet, was in beiden Fällen einen falschen Wahrheitswert liefert.

- Wird im Gegensatz dazu die Funktion mit

```
return (undef);
```

quittiert, so ist das Ergebnis ein Array mit einem Element ( `$#a == 1`) und damit wahr.

# Skalarer Kontext und List-Kontext

---

- Rückgabewert kann abhängig von Kontext sein
- wantarray() hat drei mögliche Rückgabe-Werte:

```
if(wantarray()) { }           # list context
elsif(defined wantarray()) { } # scalar context
else { }                       # void context
```

```
mysub();                       # void context
$value = mysub();             # scalar context
if(mysub()) { ... }          # scalar context
@array = mysub();             # list context
print mysub();                # list context
```

# Undefinierte Funktionen abfangen

---

- Abfangen nicht-definierter Funktionen mit Funktion AUTOLOAD
- AUTOLOAD wird statt ursprünglicher Funktion ausgeführt
- AUTOLOAD springt nur für nicht-definierte Funktionen in dem selben Package ein!
- \$AUTOLOAD enthält den (absoluten) Namen der Funktion

```
sub AUTOLOAD {  
    (my $sub = $AUTOLOAD) =~ s /. * :: // ;  
    print ( "$sub(" , join ( " , _" , @_ ) , " ); \n" );  
}
```

- Der Einsatz von AUTOLOAD sollte nur dem Notfall vorbehalten sein!

# Prototypisieren von Funktionen

---

- Durch Prototypen kann Struktur der Argumente-Liste vordefiniert werden.
- Prototypisierung von Funktionen dient in Perl
  - zur Kontextbeschreibung – selbst-definierte Funktionen können nun wie eingebaute Funktionen aufgerufen werden,
  - nicht zur Festlegung und Durchsetzung einer Aufruf-Syntax.
- Prototypen werden vom Compiler geprüft, nicht vom Interpreter!

# Prototypisieren von Funktionen

---

- Der Prototyp
  - wird in Klammern hinter dem Funktionsnamen oder -definition angegeben
  - besteht aus null oder mehreren Leerzeichen, Backslash oder Datentyp-Zeichen

```
sub funca($$) { ... } # Skalare oder Array (!)
sub funcb(\@@) { ... } # Referenz , Array
sub funcc(\@\%) { ... } # Referenzen
sub funcd() { ... } # keine Argumente
```

- Gültigkeit der Funktionsargumente kann erst zur Ausführungszeit der Funktion geprüft werden.
- Der einzig empfehlenswerte Grund für Prototypen ist die Nachbildung von eingebauten Funktionen

# Prototypisieren von Funktionen

---

- Anwendung: Weglassen der Klammern

```
# Ohne Prototyp: alle Argumente an &funca
@result = funca(3, 5); # oder: =funca 3, 5;
# mit Prototyp: sub funca($)
@result = funca 3, 5; # (funca(3), 5)
```

- Anwendung: Nachahmen eingebauter Funktionen

```
sub mypush(\@@) {
    my $aref = shift; my @any = @_; # ...
}
mypush @array, $s2, $s3, %hash;
```

⇒ erster Parameter ist Array und wird als Referenz übergeben.

# Prototypisieren von Funktionen

---

- Wenn `func($)` definiert ist, so bringt `func(1, 3)` eine Fehlermeldung beim Kompilieren, `func(@array)` wird jedoch wegen des Prototyps im skalaren Kontext behandelt und ist "korrekt".

# Ermitteln des aktuellen Funktionsnamens

---

- Der aktuelle Kontext kann mit der Funktion `caller()` ermittelt werden
- Optionaler Parameter gibt die Position der angefragten Funktion in Aufruf-Stapel an (0= aktuelle Funktion, 1= aufrufende Funktion, ...)

```
while (($package, $filename, $line, $subroutine,
      $hasargs, $wantarray, $evaltext,
      $is_require, $hints, $bitmask) = caller($i++)) {
    print "Package=%s, Sub=%s, Line=%s\n",
          $package, $subroutine, $line;
}
```

# Bearbeitung von Ausnahmenbedingungen

---

- Funktionen sollen eindeutigen Wert mit `return` zurück geben
- Aufrufende Funktion kann den Rückgabewert auswerten
- Bei schwerwiegenden Fehlern (wenn Programmablauf nicht fortführbar) können Exceptions mit `die()` ausgelöst werden
- Exceptions können in aufrufender Funktion mit `eval()` sicher abgefangen und ausgewertet werden  $\Rightarrow$  unbedingter Programmabbruch vermeidbar

# Bearbeitung von Ausnahmenbedingungen

---

- Spezielle Variable `$@` hält empfangene Fehlermeldungen

```
eval { $var = func() };  
if ($@) {  
    warn "func_raised_exception:_$@\n";  
}
```

- Es können auch nur bestimmte Exceptions abgefangen werden

```
eval { $var = func() };  
if ($@ && $@ !~ /Disk/) {  
    die; # nutzt $@ implizit  
}
```

- Bei Modulen besser `carp()` und `croak()` aus dem Carp-Modul einsetzen

# Parameterübergabe mit benannten Argumenten

---

- Es können benannte Parameter an eine Funktion übergeben werden.
- Anonymes Hash wird als Argumentliste übergeben

```
sub perimeter { # Umfang von Kreis oder Rechteck
    my %args=( DIA => 0, UNIT => 'mm', B => 0, H => 0, @_ );
    if ($args{DIA}) {
        return sprintf "%.3f%s", $args{DIA}*3.14159, $args{UNIT};
    } else {
        return sprintf "%.3f%s", ($args{B}+$args{H})*2, $args{UNIT};
    }
}

print perimeter(DIA=>23, UNIT=>'cm'), "\n"; #72.257cm
print perimeter(B=>32, H=>168), "\n";      #400.000mm
```

- Inhalt des anonymen Hashs wird an ein Hash mit Vorgabewerten angehängt.

# Ignorieren einzelner Rückgabewerte

---

- Ausblenden von Rückgabewerten, die nicht interessant sind

- Einsetzen von `undef` in der Ergebnis-Liste

- ```
my ($pkg, undef, $line, $subr)=caller();
```

- Selektieren der gewünschten Werte mittels eines Slice

- ```
my ($pkg, $line, $subr)=(caller())[0,2,3];
```

- Linker Ausdruck kann aus leerer Liste bestehen, wenn Funktion im List-Kontext aufgerufen werden soll und Rückgabewert nicht interessant ist (zur Ausnutzung von Seiteneffekten).

- ```
() = function();
```

# printf, sprintf: Formatierung der Ausgabe

---

- Formatierung durch String mit Platzhaltern für Variablen
- Angabe der Platzhalter erfolgt typisiert: %<format><typ>

| Typ | Bedeutung             | Typ | Bedeutung                              |
|-----|-----------------------|-----|----------------------------------------|
| c   | Zeichen               | lu  | lange Dezimalzahl (vorzeichenlos)      |
| d   | Dezimale Ganzzahl     | ld  | lange Dezimalzahl                      |
| o   | Oktalzahl             | lo  | lange Oktalzahl                        |
| x   | Hexadezimale Ganzzahl | lx  | lange Hexadezimalzahl                  |
| e   | Exponentialzahl       | ld  | lange Hexadezimalzahl (Großbuchstaben) |
| f   | Gleitkommazahl        | g   | Gleitkommazahl (kompakt)               |
| s   | Zeichenkette          | u   | Dezimale Ganzzahl (vorzeichenlos)      |

- Negativer Wert für <format> bewirkt Linksausrichtung der Ausgabe

```
printf "%5.3f%% des %s in KW %02d.\n", 23.2, "Bestands", 7;  
# Ergibt: 23.200% des Bestands in KW 07.
```

# Umgang mit Zeitangaben

---

- `time` liefert die Anzahl der Sekunden seit *epoch* (01.01.1970) als Ganzzahl
- `localtime` liefert

– im Array-Kontext ein Array mit den Datumsbestandteilen

```
#0      1      2      3      4      5      6      7      8  
($sec , $min , $hour , $mday , $mon , $year , $yday , $isdst ) =  
    localtime ( time );
```

– im skalaren Kontext eine Zeichenkettenrepräsentation

```
# perl -e 'print scalar localtime(1221191407)'  
Fri Sep 12 05:50:07 2008
```

eines Datums.

- Bei Aufruf von `localtime()` ohne Argument wird das aktuelle Datum eingesetzt.

# Umgang mit Zeitangaben

---

- Bei Nutzung des Zeit-Array ist zu beachten, dass
  - die Zählung des Monatswerts bei 0 beginnt
  - die Zählung des Jahreswerts bei 1900 beginnt
- Beispiel:

```
@morgen = localtime (time + 24*60*60);  
printf "morgen=%02d.%02d.%04d\n" ,  
    $morgen[3] , ++$morgen[4] , $morgen[5] + 1900;
```

# Umgang mit Zeitangaben

---

- Für Verarbeitung von Datumswerten stehen eine Vielzahl von Modulen im CPAN bereit, z.B.:
  - Date::Parse – Parse date strings into time values
  - Date::Format – Date formatting subroutines
  - Time::Format – Easy-to-use date/time formatting.
  - Class::Date – easy date and time manipulation for perl
  - Calendar::Simple – Perl extension to create simple calendars

# Ausführen von anderen Programmen

---

- `system()` führt andere Programme als Kindprozess aus
- aufgerufenes Programm hat gleiche STDIN und STDOUT wie aufrufendes Programm
- Elternprozess wartet auf Ende der Ausführung des Kindprozesses
- Perl versucht, alle Buffer von offenen Dateien zu leeren – sicherer ist `autoflush`
- Wird Argumentliste übergeben, dann wird erstes Argument als Befehl gestartet → Umgehen der Shell

```
$status = system(" ps " , "-ax" );
```

- Wird skalares Argument übergeben, so wird der String an Shell übergeben, die den Befehl ausführt

```
$status = system(" ps -ax > /tmp/ps.out" );
```

# Ausführen von anderen Programmen

---

- Fehlerursachen können durch Analyse der Variable \$? ermittelt werden:

```
$exit_value = $? >> 8;
```

```
$signal_num = $? & 127;
```

```
$dumped_core = $? & 128;
```

# Ausgabe von anderen Programmen auswerten

---

- Backticks `` sind üblicher Weg zum Ausführen von Befehlen

```
$out = 'cmd';      # Ausgabe an String uebergeben
```

```
@out = 'cmd';     # Ausgabe an Array uebergeben
```

```
@out = qx{ cmd }; # Befehlsausfuehrung analog ``
```

- Ergebnis von `` wird erst nach Ende des Befehls übergeben
- `` sollte nur verwendet werden, wenn Ausgabe von Interesse – Sammeln der Ausgabe ist aufwendig. Sonst: `system`

# Ausgabe von anderen Programmen auswerten

---

- Alternativ kann mit `open()` eine Eingabepipe geöffnet werden

```
open(my $cmd, 'cmd|'); # Befehl ausführen
while (my $line = <$cmd>) {
    # do something with $line
}
close($cmd);
```

- Ergebnis von `open()` wird zeilenweise über den Dateideskriptor eingelesen
- ‘‘ und `open` rufen eine Shell auf, um die Programme zu starten
- das ist unsicher, wenn Skript unter privilegiertem Account läuft

---

# Fortgeschrittenes Arbeiten mit Dateien

- Erstellen von temporären Dateien
- Statusinformationen zu Dateien auswerten
- Blockieren von Dateien
- Setzen des Standardausgabekanals
- Steuern der Ausgabepufferung

# Erstellen von temporären Dateien

---

- Modul IO::File stellt Klassen-Methode new\_tmpfile() bereit, der Dateiname wird nicht bekannt

```
use IO::File;  
$fh = IO::File->new_tmpfile  
    or die "Error: _Couldn't open _tmpfile: _$!";
```

- Wenn Dateiname bedeutsam, dann ist dies mit der Funktion tmpnam() aus dem Posix-Modul möglich

```
use Posix; use IO::File;  
$name = tmpnam;  
$fh = IO::File->new($name, O_RDWR|O_CREAT|O_EXCL, 0400)  
    or die "Error: _Couldn't open _tmpfile: _$!";
```

- Mit \$fh=IO::File->new(...) wird Filehandle gelöscht, sobald Variable nicht mehr gültig oder Programm beendet ist!

# Sperren/Locken von Dateien

---

- “Locken” dient zur Steuerung des simultanen Zugriffs auf eine Datei
  - Ein Prozess sperrt die Datei
  - Andere Prozesse warten mit Dateizugriff bis zum Ende der Sperre
- Leicht möglich mit flock (FH, \$operation)
- flock betreibt “advisory locking”: Zeigt Zugriffe an, warnt, kann (ignorante) Parallelzugriffe nicht verhindern
- Mögliche Lock-Operationen

| Id | Fcntl   | Bedeutung                                                              |
|----|---------|------------------------------------------------------------------------|
| 1  | LOCK_SH | Shared lock (lesen erlaubt, schreiben nicht erlaubt)                   |
| 2  | LOCK_EX | Exclusive lock (lesen und schreiben nicht erlaubt)                     |
| 4  | LOCK_NB | Don't block when locking (nicht auf die Aufhebung der Blockade warten) |
| 8  | LOCK_UN | Unlock                                                                 |

## Beispiel: Modifizieren einer Datei ohne Temporärdatei

---

```
open(my $datei , '+<' , 'datei.txt ' ) or die " Error(open):_$_!\n" ;
# Einlesen der Datei
while (<$datei> ) {
    s/DATE/localtime/eg;
    $out .= $_;
}
# Zuruecksetzen des Positionszeigers
seek($datei , 0, 0) or die " Error(seek):_$_!\n" ;
print {$datei} $out or die " Error(print):_$_!\n" ;
# Dateilaenge aktualisieren
truncate($datei , tell($datei)) or die " Error(trunc):_$_!\n" ;
close $datei or die " Error(close):_$_!\n" ;
```

# Statusinformationen zu Dateien ermitteln

---

- `stat ()` gibt eine Liste von Informationen zu einer Datei zurück

| Element | Kurzwort             | Beschreibung                                            |
|---------|----------------------|---------------------------------------------------------|
| 0       | <code>dev</code>     | Gerätenummer des Dateisystems                           |
| 1       | <code>ino</code>     | Inode-Nummer der Datei                                  |
| 2       | <code>mode</code>    | Dateimodus (Typ und Permissions)                        |
| 3       | <code>nlink</code>   | Anzahl der (hard) Links auf die Datei                   |
| 4       | <code>uid</code>     | numerische User ID des Dateibesitzers                   |
| 5       | <code>gid</code>     | numerische Group ID des Dateibesitzers                  |
| 6       | <code>rdev</code>    | Raw-Gerätenummer bei Gerätedateien                      |
| 7       | <code>size</code>    | Dateigröße, in Bytes                                    |
| 8       | <code>atime</code>   | Letzte Zugriffszeit in Sekunden seit Epoch              |
| 9       | <code>mtime</code>   | Letzte Änderungszeit in Sekunden seit Epoch             |
| 10      | <code>ctime</code>   | Letzte Zugriffszeit, die Inode-Änderung zur Folge hatte |
| 11      | <code>blksize</code> | Ideale Blockgröße für blockweises Lesen und Schreiben   |
| 12      | <code>blocks</code>  | Anzahl belegter Blöcke                                  |

- Nutzbar in Bedingungen, schon bevor die Datei geöffnet wird

# Statusinformationen zu Dateien ermitteln

---

- `stat()` gibt leere Liste zurück, wenn Informationen nicht ermittelt werden konnten

```
( $dev , $ino , $mode , $nlink , $uid , $gid , $rdev , $size ,  
    $atime , $mtime , $ctime , $blksize , $blocks ) = stat FH;  
($mode , $uid , $gid , $size) = ( stat $file )[2 , 4 , 5 , 7];
```

- Kurzform mit Testoperatoren `-X`
- Ergebnis des Testoperators ist ein boolescher Wert
- Datei-Deskriptor `_` hält Cache des letzten `stat()`-Aufrufs

```
if ( -e $datei && -T _ ) { ... }
```

# Statusinformationen zu Dateien ermitteln

---

- Auswahl an Testoperatoren

| -X | Bedeutung                       | -X | Bedeutung                       |
|----|---------------------------------|----|---------------------------------|
| -r | Datei ist lesbar                | -f | Datei ist eine Datei            |
| -w | Datei ist schreibbar            | -d | Datei ist ein Verzeichnis       |
| -x | Datei ist ausführbar            | -l | Datei ist ein symbolischer Link |
| -e | Datei existiert                 | -p | Datei ist eine benannte Pipe    |
| -z | Datei hat Größe 0               | -S | Datei ist ein Socket            |
| -s | Datei hat Größe ungleich 0      | -b | Datei ist ein Block-Gerät       |
| -T | Datei ist eine Text-Datei       | -c | Datei ist ein Character-Gerät   |
| -B | Datei ist eine Binär-Datei      | -t | Datei ist ein Terminal          |
| -M | Letzte Zugriffszeit             |    |                                 |
| -A | Letzte Änderungszeit            |    |                                 |
| -C | Letzte Änderungszeit des Inodes |    |                                 |

# Standard-Ausgabekanal setzen

---

- Standard-Ausgabekanal muss bei Ausgaben nicht angegeben werden
- `select ()` setzt übergebenen Datei-Deskriptor als Standard-Ausgabekanal und gibt bisherigen Ausgabekanal zurück

```
open(my $log, '>', 'prog.log') or die "open_prog.log:_$!\n";
$oldfh = select $log;      # Ausgabekanal sichern
print scalar localtime; # Ausgabe nach $log!
select $oldfh;           # Zuruecksetzen auf STDOUT
close $log;
```

- `STDOUT` ist voreingestellter Ausgabekanal (Datei-Deskriptor)

```
print {*STDOUT} "Hallo_Welt";
print "Hallo_Welt"; # identisch
```

# Ausgabebufferung steuern

---

- Perl buffert Ausgaben
  - zeilenweise bei Ausgabe an ein Terminal
  - blockweise bei anderweitigen Ausgaben
- Nicht-gebufferte Ausgabe ist sinnvoll bei Ausgabe an eine Pipe
- Für nicht-gebufferte Ausgabe (z.B. mit `print ()` oder `write ()`) muss `autoflush` gesetzt werden

# Ausgabebufferung steuern

---

- Bufferung ist ausgeschaltet, wenn `$|` wahr ist.

```
open (my $out, '>', 'unbuffered.file')
    or die "Couldn't open unbuffered.file: $_$!\n";
$ofh = select($out);
$| = 1;
select($ofh);
```

- Eleganter mit Modul `FileHandle`

```
use FileHandle;
$fh = new FileHandle ">unbuffered.file";
$fh->autoflush(1);
```

oder

```
autoflush STDOUT 1;
```

---

# Referenzen und verschachtelte Datenstrukturen

- Referenzen in Perl
- Erzeugen von Referenzen
- Auflösen von Referenzen, Zugriff auf Referenten
- Verschachtelte Datenstrukturen

# Referenzen in Perl

---

- Mit Referenzen können beliebig komplexe Datenstrukturen angelegt werden
- Funktionsweise der Referenzen:
  - Variablen werden durch Namen und Verweis auf eine Speicher-Adresse abgebildet
  - Referenz ist ein (skalärer) Wert, der einen Verweis auf die Speicherstelle eines anderen Wertes enthält.
  - Variable und dessen Referenz(en) verweisen auf die selbe Speicherstelle  
⇒ Ändern des Wertes der Referenz ändert auch Wert der Variable

# Referenzen und Referenten

---

- der Wert, auf den die Referenz zeigt, ist der “Referent”
- Referent kann jeder eingebaute Datentyp (Skalar, Array, Hash, Referenz, Code, Glob) oder darauf basierender Datentyp sein

Referenz 0x542c5e (Referent)

ARRAY (0x542c5e)  $\Rightarrow$  (8, 24, 'Hallo Welt')

- Perl überprüft die Anzahl der Referenzen auf eine Variable
  - belegter Speicher wird erst freigegeben, wenn keine Referenzen auf den Referenten existieren
  - Referenzen sind so immer gültig

# Typisierte Referenzen

---

- Referenzen in Perl sind typisiert
  - Eine Referenz auf ein Array kann nicht als Referenz auf ein Hash behandelt werden.
  - Verstöße verursachen eine Runtime Exception.

```
# perl -e '$aref = [ 1 .. 4 ]; print "$aref->{1}\n" '  
Can not coerce array into hash at -e line 1.
```

```
# perl -e '$aref = [ 1 .. 4 ]; print "$aref->[1]\n" '  
2
```

- Es gibt (noch) keinen Mechanismus zum Typ-Casting

# Erzeugen von Referenzen

---

- Die Referenz auf eine Variable wird durch Voranstellen von “\” erzeugt
- Diese Referenz wird einer skalaren Variable zugewiesen

```
$var = 3;           # skalare Variable
$sref = \ $var;    # $sref ist Skalar-Referenz
@var = (1 .. 3);   # Array-Variable
$aeref = \@var;   # $aeref ist Array-Referenz
%var = (uid => "F23", # Hash-Variable
        qty => 1);
$href = \%var;     # $href ist Hash-Referenz
```

- Referenzen auf existierende Datenstrukturen sind sinnvoll, wenn an einen Funktionsaufruf die Referenz übergeben werden soll (Pass-by-Reference).

# Anonyme Referenzen

---

- Explizites Anlegen von Referenz-Variablen kann hinderlich sein beim dynamischen Erweitern von Datenstrukturen um neue Arrays und Hashes. Anonyme Referenzen umgehen dies.

```
$aref = [1 .. 3];           # Array-Referenz
$href = {uid => "F23", qty => 1}; # Hash-Referenz
$fref = sub { CODE };      # Code-Referenz
```

- Erzeugen von benannten und anonymen Referenzen

| Referenz auf | benannt                    | anonym                        |
|--------------|----------------------------|-------------------------------|
| Scalar       | <code>\\$scalar</code>     | <code>\do{my \$scalar}</code> |
| Array        | <code>\@array</code>       | <code>[ LIST ]</code>         |
| Hash         | <code>\%hash</code>        | <code>{ LIST }</code>         |
| Code         | <code>&amp;function</code> | <code>sub { CODE }</code>     |

# Zugriff auf Referenten – Dereferenzieren

---

- Dereferenzieren erfolgt durch Voranstellen des Typ-Symbols

`$scalar = $$sref;`

`@array = @$aref;`

`%hash = %$href;`

- Referent kann auch in { } eingeschlossen werden

`$scalar = ${$sref};`

`@array = @{$aref};`

`%hash = %{$href};`

# Zugriff auf Referenten – Dereferenzieren

---

- Zugriff auf Elemente des referenzierten Arrays oder Hashes durch infix Zeiger-Operator

```
$a1 = $aref ->[1];  
$uid = $href->{uid};
```

- Aufruf einer indirekt referenzierten Funktion ebenfalls durch infix Zeiger-Operator

```
$code_ref ->(" arg1" , " arg2" );
```

# Zugriff auf Referenten – Dereferenzieren

---

```
print $sref , "\n";           # SCALAR(0x8105490)
print $$sref , "\n";         # 3
print ${$sref} , "\n";      # 3
print "$$sref\n";           # 3
print $aref , "\n";          # ARRAY(0x81054d8)
print @{$aref} , "\n";      # 123
print "@{$aref}\n";         # 1 2 3
print $href , "\n";          # HASH(0x81055a4)
print $href->{uid} , "\n";   # F23
print "$href->{qty}\n";      # 1
```

# Testen von Referenzen

---

- Mit der Funktion `ref(EXPR)` kann geprüft werden, ob ein Ausdruck eine Referenz ist.
- Ergebnis ist Wahrheitswert
- Rückgabewert hängt von Art der Referenz ab (SCALAR, ARRAY, HASH, CODE, REF, GLOB, LVALUE)

```
if (ref($value) eq "HASH") {  
    print "\$value is a reference to a hash.\n";  
}  
unless (ref($value)) {  
    print "\$value is not a reference at all.\n";  
}
```

- Bei Objekt-Referenzen wird der Package-Name zurück gegeben
- Wenn `ref` ohne Argument aufgerufen wurde, erfolgt die Prüfung für `$_`

# Verschachtelte Datenstrukturen

---

- In Perl werden Referenzen vor allem verwendet, um die Beschränkung zu umgehen, dass Arrays und Hashes “nur” Skalare enthalten dürfen.
- Um ein mehrdimensionales Array (List of Lists) zu erstellen, wird ein Array von Array-Referenzen angelegt.
- So können leicht beliebige Datenstrukturen, wie Lists of Lists (LoL), List of Hashes (LoH), Hashes of Lists (HoL) und Hashes of Hashes (HoH) und beliebig komplexere Datenstrukturen abgebildet werden.

# Beispiel: eine Person

---

```
$person = { uid => "hmeier" ,  
            name => { name => "Hans" , sname => "Meier" } ,  
            place => { city => "Rostock" , zip => "18055" ,  
                      street => "Goethestrasse_18" } ,  
            birthday => [24, 12, 1967 ] ,  
};  
printf " _ _ _ %s _ %s \n _ _ _ %s \n %s _ %s \n _ _ _ [%02d.%02d.%04d]" ,  
    $person->{name}->{name} , $person->{name}->{sname} ,  
    $person->{place}->{street} ,  
    $person->{place}->{zip} , $person->{place}->{city} ,  
    $person->{birthday}[0] , $person->{birthday}->[1] ,  
    $person->{birthday}[2]; # "->" optionally omitted!
```

# Beispiel: eine Personendatenbank

---

```
$addr{" hmeier"} = { name => "Hans" ,  
    sname => "Meier" , city => "18055_Rostock" ,  
};  
$addr{" smueller"} = { name => "Steffen" ,  
    sname => "Mueller" , city => "18107_Rostock" ,  
};  
foreach my $person (sort keys %addr) {  
    printf "%s , %s\n_%s_%s\n\n" ,  
        $addr{$person}->{sname} , $addr{$person}->{name} ,  
        $addr{$person}->{city} ;  
    $addr{$person}{processed} = 1; # "->" might be omitted  
}
```

# Beispiel: eine Matrix

---

```
# eine zwei-dimensionale Matrix
$matrix = [[1, 2], [2, 3]];
print $matrix ->[0][1], "\n";    # 2
$matrix ->[1][1] = 25;
print "@{$matrix ->[1]}" , "\n"; # 2 25
```

# Manipulation von verschachtelten Datenstrukturen

---

- Die für die originären Datentypen anwendbaren Operationen können ebenso auf verschachtelte Datenstrukturen angewendet werden.

```
$bar = $$scalarref;
```

```
push(@$arrayref, $filename);
```

```
$$arrayref[0] = "Januar";
```

```
$$hashref{KEY} = "VALUE";
```

```
&$coderef(1,2,3); # Funktionsaufruf
```

```
$coderef->(1,2,3); # identisch!
```

```
$val = (defined $$href{KEY}) ? $$href{KEY} : 0;
```

```
&{ $dispatch{$index} }(1,2,3); # Funktionsaufruf
```

---

# Packages, Module und Pragmas

- Packages
- Module
- Pragmas

# Packages

---

- Package ist der Mechanismus zur Definition von Sichtbarkeitsbereichen (Partitionierung des globalen Namespace)
- Package ist Basis für Module und objektorientierte Klassen
- Sichtweite einer Package-Deklaration reicht
  - von der Deklaration
  - bis zum Ende des einschließenden Blocks oder
  - bis zur nächsten Package-Deklaration
- In den Packages definierte Variablen und Funktionen haben Gültigkeit nur in dem aktuellen Package (jedes Package hat eigene Symboltabelle)

# Packages

---

- Bei unqualifizierten Variablen haben lexikalische Variablen Vorrang vor globalen Variablen
- Ein Programm kann beliebig viele Package-Deklaration enthalten
- Package-Bezeichner beginnen mit einem Großbuchstaben (per Konvention)
- Eine Package-Deklaration kann an einer beliebigen Stelle eingefügt werden, an der eine Anweisung stehen darf
- Initiales aktuelles Package ist `main`, es ist implizit definiert
- Variablen ohne Package-Bezeichner gehören automatisch zum Package `main`

# Packages

---

- Auf globale Variablen in den Packages kann durch vorangestellten Package-Bezeichner `$package::var` zugegriffen werden
- Auf lokale Variablen eines Packages kann nicht von außerhalb zugegriffen werden
- Eingebaute Variable (wie `$_` und `$.`) sowie die Bezeichner `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC` und `SIG` haben ihre Sonderbedeutung nur im Package `main`.
- So meint `%ENV` immer `%main::ENV`.
- Die Variable `%MyPackage::ENV` hat keine Sonderbedeutung.

# Packages

---

```
package Oder;
$var =5;
package So;
$var =23;
print " [$Oder::var , _$So::var , _$var]\n"; # [5 , 23 , 23]
package main;
$var ="Hallo";
print "\$var=$var\n"; # Hallo
print "\$Oder::var=$Oder::var\n"; # 5
$Oder::var=7;
print "\$Oder::var=$Oder::var\n"; # 7
print "\$So::var=$So::var\n"; # 23
```

# Module

---

- Module sind wiederverwendbare Software-Einheiten
- Sie enthalten eine Sammlung verwandter Funktionen, organisiert in einem Package
- Das Package wird in einer Bibliotheks-Datei mit demselben Namen wie das Package und der Endung `.pm` abgelegt
- Jedes Modul hat ein öffentliches Interface – öffentlich ist die Funktionalität, die vom Implementor dokumentiert wurde
- Auf undokumentierte Funktionen und Bezeichner sollte nicht von außerhalb des Modules zugegriffen werden (aber es ist möglich)

# Module

---

- Thematisch zusammengehörige Module können strukturiert abgelegt werden

`Text::RTF`, `Text::HTML`, `Text::Plain::ASCII`, `Text::Plain::UTF8`

- Die Modul-Strukturen werden auf Verzeichnisse im Dateisystem abgebildet: `Text/RTF.pm`, `Text/HTML.pm`, `Text/Plain/ASCII.pm`, `Text/Plain/UTF8.pm`
- Modulnamen sollten immer mit einem Großbuchstaben (in Unterscheidung zu Pragmas) beginnen
- Module sollten immer mit POD-Kommentaren dokumentiert werden – Anwender können mit `pod2text Modulname` die öffentliche Schnittstelle des Moduls studieren (siehe `perldoc perlpod`).

# Laden von Modulen

---

- Ein Programm fordert Module mit `use` oder `require` an

```
use Text::plain::ASCII;  
require Fcntl;
```
- `require` lädt Modul zur Laufzeit und prüft, ob das Modul bereits geladen wurde
  - Funktions-Prototypen werden nicht sichtbar für den Compiler wegen Ladens zur Laufzeit. Nur der Compiler berücksichtigt Prototypen, nicht der Interpreter!
- `use` lädt Modul zur Compile-Zeit und importiert gleichzeitig freigegebene Bezeichner
  - Programm startet nicht, wenn ein Modul nicht geladen werden kann, da Compilation fehl schlägt
  - Funktions-Prototypen werden sichtbar für den Compiler

# Laden von Modulen

---

- Module werden in den in @INC angegebenen Verzeichnissen gesucht.

```
# perl -e 'print "@INC\n";'  
  /etc/perl /usr/local/lib/perl/5.8.8  
  /usr/local/share/perl/5.8.8 /usr/lib/perl5  
  /usr/share/perl5 /usr/lib/perl/5.8  
  /usr/share/perl/5.8 /usr/local/lib/site_perl .
```

- use ist wegen seines Compile-Zeit-Verhaltens nutzbar für Compiler-Anweisungen (Pragmas)

# Aufbau eines Modules

---

```
package Text::HTML;
use strict;
use warnings;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(func1);
@EXPORT_OK = qw(func3);
%EXPORT_TAGS = (extra => [qw/func2/]);
$var1 = "Wo"; @array2 = (" bist", "Du");
END { unlink ".tmp" if ( -e ".tmp" && -f _ ) }
sub func1 { ... }
sub func2 { ... }
sub func3 { ... }
1;      # Modul muss immer 'true' evaluieren!!
```

# Aufbau eines Modules

---

- Bedeutung der vordefinierten Variablen

@ISA                    beschreibt Vererbungshierarchie

@EXPORT                implizit zu exportierende Bezeichner (autoexport)

@EXPORT\_OK            auf Anforderung exportierbare Bezeichner

%EXPORT\_TAGS        Symbol-Sets definieren

- Variablen sollten NIE zur API eines Moduls gehören, sondern immer über eine Funktion erreichbar/modifizierbar sein
- Ein geladenes Modul muss erfolgreich evaluiert werden, deshalb sollte die letzte Anweisung eines jeden Moduls dies garantieren, z.B.:  

```
1;
```
- Package-Name sollte immer synchron zum Datei-Namen sein, Abweichungen davon werden erst bemerkt, wenn auf die Bezeichner nicht zugegriffen werden kann  $\Rightarrow$  schwer zu finden und zu debuggen.

# Import und Export von Bezeichnern

---

- Mit Import wird im aktuellen Package ein Alias auf den importierten Bezeichner angelegt.
- Importierter Bezeichner muss nicht mehr voll qualifiziert oder mit `use subs` bzw. `use vars` vordeklariert werden.
- `@EXPORT`: Bezeichner werden implizit geladen.
- `@EXPORT_OK`: Bezeichner können bei Bedarf geladen werden.
- `@EXPORT_TAGS`: Sets von Bezeichner können bei Bedarf geladen werden.

```
use Module;                # @EXPORT
use Module qw(&function);  # @EXPORT_OK
use Module qw(:SET);       # @EXPORT_TAGS
```

# Import und Export von Bezeichnern

---

- Zu importierende Bezeichner können auch kombiniert werden

```
use Module qw(:SET %hash); # %EXPORT_TAGS @EXPORT_OK
```

- Gültige (Mindest-)Version eines Moduls kann geprüft werden

```
use Module 1.26; # fail if ($VERSION < 1.26)
```

- Mit dem Import-Mechanismus können auch eingebaute Funktionen überladen werden

```
use PreciseTime qw(time); # time() wird reimplementiert
```

```
my $start = time(); # PreciseTime::time()
```

# Fehler beim Modul-Import abfangen

---

- Nicht-Ladbarkeit eines Modules erzeugt Exception
- Um darauf reagieren zu können, wird das Laden in BEGIN{ } evaluiert

```
BEGIN {  
    unless (eval "require_$module") { # ohne Import  
        warn "Couldn't load_$module:_$@" ;  
    }  
    unless (eval "use_$module") { # mit Import  
        warn "Couldn't load_$module:_$@" ;  
    }  
}
```

- BEGIN{ } stellt sicher, dass eval() zur Compile-Zeit statt zur Laufzeit ausgeführt wird

# Laden eines Modules aus einer Reihe von Alternativen

---

- Bei Bedarf kann die konkrete, geladene Implementierung eines Modules im Programm beeinflusst werden

```
BEGIN {  
    my $found;  
    my @MODS = qw( Module1 Module2 Module::Mod3 );  
    for my $mod (@MODS) {  
        if (eval "require _$module") {  
            $mod->import();  
            $found = 1;  
            last;  
        }  
    }  
    die "Couldn't load _@MODS" unless $found;  
}
```

# Automatisieren von Modul-Initialisierung und -Aufräumung

- Initialisierung erfolgt durch Anweisungen, die außerhalb von Funktionsdefinitionen codiert sind.
- Modul wird beim Laden (mit `use` oder `require`) evaluiert.
- Funktion `END{ }` wird beim Beenden des Programms ausgeführt.
- Die einzelnen `END{ }`-Funktionen werden in der umgekehrten Reihenfolge der Modul-Initialisierung ausgeführt.
- `END{ }` wird ausgeführt, wenn Perl Programmende entdeckt (auch bei `die()`, `exit()` und Laufzeitfehlern).
- Bei nicht-abgefangenen Signalen wird `END` nicht ausgeführt.  

```
use sigtrap qw(die normal-signals error-signals);
```
- Anmerkung: `BEGIN{ }` wird beim Compilieren ausgeführt, nicht bei Initialisierung!

# Automatisieren von Modul-Initialisierung und -Aufräumung

---

- Beispiel: Initialisieren und Aufräumen eines Logger-Modules

```
package Logger ;
# ...
BEGIN { $logfile = "/var/log/logfile.log" ; }
our $log ;
sub msg { print {$log}, shift }
open($log, '>', $logfile) or
    die "Couldn't open $logfile : $!\n" ;
END {
    msg(" Closing Logfile" );
    close $log ;
}
```

# Eigene Modul-Verzeichnisse integrieren

---

- Perl sucht nach Modulen in allen Verzeichnissen, die in @INC enthalten sind
- Weitere Verzeichnisse können bei Ausführen des Programmes hinzugefügt werden:

- Perl's Kommandozeilen-Option -I

```
perl -I/apps/lib/perl myprog.pl
```

- Umgebungsvariable PERL5LIB

```
export PERL5LIB=$PERL5LIB:/apps/lib/perl # sh
```

# Eigene Modul-Verzeichnisse integrieren

---

- Im Programm können weitere Bibliothekspfade definiert werden:

- Mit dem Pragma `lib`

```
#!/usr/bin/perl  
use lib "/apps/lib/perl";
```

- Wenn ein Pfad relativ zum ausgeführten Skript eingebunden werden soll, dann hilft das Modul `FindBin`:

```
use FindBin qw($Bin);  
use lib "$Bin/../lib";
```

# Ein neues Projekt erstellen

---

- Mit `module-starter` aus dem Modul `Module::Starter` werden Templates für Projekte generiert

```
# module-starter --verbose --mb --force \  
  --module Text::HTML  
  --author=Dirk\ Ruediger --email=dirk@niebegeg.net  
Created Text-HTML  
Created Text-HTML/lib/Text  
Created Text-HTML/lib/Text/HTML.pm  
Created Text-HTML/t  
Created Text-HTML/t/pod-coverage.t  
Created Text-HTML/t/pod.t  
Created Text-HTML/t/boilerplate.t  
Created Text-HTML/t/00-load.t  
Created Text-HTML/.cvsignore
```

# Ein neues Projekt erstellen

---

```
Created Text-HTML/Build.PL
Created Text-HTML/Changes
Created Text-HTML/README
Created Text-HTML/MANIFEST
Created starter directories and files
```

- Inhalt der Distribution im Verzeichnis Text-HTML/
  - lib/ Verzeichnis enthält alle Module
  - Build.PL Buildskript zum Bauen, Testen, Packen und Distributieren des Moduls
  - README allgemeine Informationen für Modul-Anwender
  - t/ Verzeichnis mit Testfällen für das Modul, ausführbar mit `perl Build test` oder `prove -r`
  - Changes Protokoll zur Projektgeschichte und -entwicklung
  - MANIFEST Liste aller Projektdateien (wird zum Zusammenstellen der Distribution benötigt)
- Modul kann leicht erweitert und portabel implementiert werden.

# Ein neues Projekt erstellen

---

- Buildskript kann erweitert werden
  - Hinzufügen von benötigten Modulen (beim Bau und zur Laufzeit)
  - Anmelden von Testfällen
- Buildskript steuert den Umgang mit den Projektdateien

```
perl Build.PL # Generieren des ausfuehrbaren Buildskripts
./Build      # Erzeugen der Projektdateien (Libs , POD, ...)
./Build test # Ausfuehren der Tests
./Build install # Installieren der Anwendung
./Build dist  # Erzeugen einer Distributionsdatei
```

- Siehe `Module::Build` für die Möglichkeiten

# Pragmas

---

- Pragmas dienen zum Definieren von Anweisungen für den Compiler
- Sie sind eine besondere Art von Modulen
- Pragma-Namen beginnen immer mit Kleinbuchstaben
- Wichtige definierte Pragmas sind:
  - diagnostics verstärkte Diagnose-Meldungen anfordern
  - lib Manipulation von @INC zur Compile-Zeit
  - locale POSIX-Locales nutzen
  - overload Perl-Operationen überladen
  - sigtrap einfache Signal-Behandlung ermöglichen
  - strict unsichere Konstrukte einschränken
  - subs Vordeklarieren von Funktionen
  - vars Vordeklarieren von globalen Bezeichnern

# Pragmas: strict

---

- Mit `strict` können unsichere Sprach-Konstrukte eingeschränkt werden

`use strict; # Anwenden aller moeglichen Restriktionen`

- Die Restriktionen haben Gültigkeit bis zum Ende des einschließenden Blocks oder bis zur aufhebenden Anweisung

`no strict; # Aufheben aller definierten Restriktionen`

- Verstöße gegen die Restriktionen werden mit Compiler-Fehlermeldungen quittiert

# Pragmas: strict

---

- bei `use strict` werden drei Kategorien unterschieden:
  - `subs` beschränkt unsicheren Gebrauch von Funktionen. Ein Compiler-Fehler wird ausgegeben beim Gebrauch eines Wortes ohne Namensraum-Bezeichner (Bareword), welches nicht als Funktion identifiziert werden kann
  - `vars` beschränkt unsicheren Gebrauch von Variablen. Ein Compiler-Fehler wird ausgegeben, wenn eine Variable ohne vorherige Deklaration gebraucht wird
  - `refs` beschränkt unsicheren Gebrauch von Referenzen. Ein Compiler-Fehler wird ausgegeben, wenn symbolische Referenzen benutzt werden

# Pragmas: strict

---

- Einzelne Kategorien können selektiv aktiviert

```
use strict 'vars';
```

oder deaktiviert

```
use strict;  
no strict 'refs';
```

werden

- Deklaration von Bezeichnern erfolgt mit der Anweisung

```
use vars qw($maus $katze %zoff);
```

- Deklaration von Funktionen erfolgt mit der Anweisung

```
use subs qw(func1 func2);
```

# Pragmas: strict

---

Ein Beispiel:

```
use strict 'vars';
use vars qw/$maus @kaese/;# Deklariert $maus und @kaese
                                # im aktuellen Package
$maus = 'klein';                # OK, global deklariert
@kaese = qq/Edam Brie/;        # via Pragma
my $falle = 'offen';          # OK, lokal deklariert via "my"
$ergebnis = 'leer';           # Compiler-Fehler: $ergebnis
                                # nicht deklariert!
{
    no strict 'vars';          # Ausschalten in einem Block
    $hallo = "Welt";           # $hallo ist nicht vordeklariert
}
```

# Pragmas: warnings

---

- Das Pragma `warnings` ist ein Ersatz des Kommandozeilenschalters `-w`
- Mit `use warnings;` werden alle Warnungen aktiviert, mit `no warnings;` werden alle Warnungen deaktiviert.
- `warnings` ist begrenzt aus den umschließenden Block (`-w` gilt global)
- So können Warnungen lokal deaktiviert werden, z.B. wenn man auf den Grund für die Warnung keinen Einfluss hat

```
use warnings;  
# etwas Programmtext  
{  
    no warnings; # gilt bis zum Ende des Blocks  
# etwas mehr Programmtext  
}
```

---

# Dokumentation des Perl-Programmtextes

- POD ist eine einfache Auszeichnungssprache
- POD = Plain Old Documentation
- Entwickelt für die Dokumentation von Perl, Perl-Programmen und Perl-Modulen
- Übersetzer für die Konvertierung von POD in die verschiedensten Formate (plain text, HTML, man pages, und mehr) sind vorhanden.
- Dokumentation kann in den Programmtext eingebunden werden  $\Rightarrow$  die Dokumentation und der Programmtext entstehen gemeinsam
- Dokumentation wird bei Anzeige direkt aus dem Programmtext extrahiert, keine Vorverarbeitung notwendig

- 
- POD markup besteht aus drei Absatztypen:

**einfach (ordinary)** Einfacher Text, der Hauptanteil der Dokumentation.

Der Text beginnt in der ersten Spalte und wird begrenzt durch Leerzeile vor und nach dem Absatz. Er kann eine Vielzahl von Texthervorhebungen enthalten (fett, kursiv, ...).

**wortgetreu (verbatim)** Der Text wird so dargestellt wie eingegeben, z.B.

Code-Blöcke. Alle Zeilen des Absatzes beginnen mit mindestens einem Leerzeichen oder Tab. Alle Zeichen werden unverändert ausgegeben.

**Anweisung (command)** In dem Text enthält Befehle zur Verarbeitung größerer Textabschnitte („Überschriften, Aufzählungen, ...“)

# Anweisungs-Absätze

---

| Aweisung                                                                                                     | Bedeutung                                                |
|--------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| <code>=head1 text</code><br><code>=head2 text</code><br><code>=head3 text</code><br><code>=head4 text</code> | Überschriften                                            |
| <code>=over</code><br><code>=item text</code><br><code>=back</code>                                          | Aufzählungen                                             |
| <code>=begin format</code><br><code>=end format</code><br><code>=for format text</code>                      | Direkte Anweisungen für den Übersetzer in Ausgabeformate |
| <code>=pod</code><br><code>=cut</code>                                                                       | Beginn- und Endmarken für POD-Prozessor                  |

# Anweisungen: Überschriften

---

=head1 Program dokumentation

=head2 Getting started

This section ...

=head2 Program window

The main window ...

=head3 Navigation view

The navigation view is located in the left sidebar ...

# Anweisungen: Aufzählungen

---

=over

=item Text

=over

=item Plain

=item XML

=back

=back

# Anweisungen: Direktausgabe für Übersetzer

---

```
=begin html
```

```
E-Mail an
```

```
<a href="mailto:dirk@niebegeg.net">Dirk Ruediger</a>  
schreiben.
```

```
=end html
```

```
=for docbook <author><affiliation>
```

```
  <address> <email>dirk@niebegeg.net</email></address>
```

```
  </affiliation></author>
```

- =for format text wirkt nur auf den unmittelbar folgenden Absatz!

# Anweisungen: POD-Beginn- und POD-Endmarken

---

```
=pod
```

```
=item unique_hash_value()
```

`unique_hash_value()` calculates for an array of arguments a unique hash value.

```
=cut
```

```
sub unique_hash_value {  
    ...  
}
```

# Formatierungscode

---

<code>I&lt;text&gt;</code>	kursiver Text
<code>B&lt;text&gt;</code>	fetter Text
<code>C&lt;code&gt;</code>	"Schreibmaschinenschrift", z.B. fuer Sourcecode
<code>L&lt;name&gt;</code>	Ein Verweis auf eine andere Manpage
<code>L&lt;name/sec&gt;</code>	
<code>L&lt;text name/sec&gt;</code>	
<code>L&lt;http://xmpl.com/&gt;</code>	ein Verweis auf eine Webseite
<code>E&lt;entity&gt;</code>	Ein Entity, ähnlich wie in HTML (z.B. <code>E&lt;lt&gt;</code> für <code>&lt;</code> , <code>E&lt;gt&gt;</code> für <code>&gt;</code> , <code>E&lt;verbar&gt;</code> für ein <code> </code> , <code>E&lt;htmlname&gt;</code> und <code>E&lt;number&gt;</code> fuer HTML-Entities)
<code>F&lt;filename&gt;</code>	ein Dateiname
<code>S&lt;text&gt;</code>	Text mit Leerzeichen, an denen nicht umgebrochen wird

# Einbau in die Hilfeausgabe eines Programmes

---

```
use Getopt::Long;
use Pod::Usage;

my $help = 0;
my $man = 0;
my $result = GetOptions(
    'help|?'      => \$help,
    'usage'       => \$man,
    # any other settings come here ...
) or pod2usage(2);

pod2usage(1) if $help;
pod2usage(-exitstatus => 0, -verbose => 2) if $man;
```

ergibt ...

# Einbau in die Hilfeausgabe eines Programmes

---

- Kurzhilfe

```
# myprog -h
```

```
Usage:
```

```
myprog [-h] [--usage] [...]
```

- Ausführliche Hilfe

```
# myprog --usage
```

```
Usage:
```

```
myprog [-h] [--usage] [...]
```

```
# ... komplette Hilfe folgt im Pager
```

- Weitere Infos in der Dokumentation zu `Getopt::Long` und `Pod::Usage` – natürlich im POD-Format eingebettet in den Modulen

# Anzeige und Übersetzung der Dokumentation

---

- Anzeigen der Dokumentation zu einem Modul

```
perldoc Text::Manip
```

- Anzeigen der Dokumentation zu einer Datei

```
perldoc example.pl
```

- Übersetzen der Dokumentation in Ausgabeformate

```
pod2man example.pl
```

```
pod2html example.pl
```

```
pod2latex example.pl
```

```
pod2text example.pl
```

- Syntaktische Überprüfung der Dokumentation

```
podchecker example.pl
```

# Anzeige und Übersetzung der Dokumentation

---

- Anzeige des Quelltextes und der Dokumentation anderer Module

`perldoc -m Modul::Name`      # z.B.: `perldoc -m FindBin`

---

# Testen in Perl

- Tests, Testfälle und Testsuiten
- Schreiben von Tests
- Was kann getestet werden?
- Untestbaren Code testen
- Exkurs: Testgetriebene Softwareentwicklung

# Tests, Testfälle und Testsuiten

---

- Tests sind eine ausführbare, selbst-verifizierende Spezifikation des Verhaltens einer Software
- Zu jedem Zeitpunkt kann geprüft werden, dass der Programmcode die Spezifikation erfüllt.
- Tests erhöhen Sicherheit beim Schreiben korrekten Codes
- Tests formulieren Anweisungen zur Ausführung von Programmblöcken und vergleichen ein erwartetes Ergebnis mit dem tatsächlichen Ergebnis
- Tests ermöglichen eine reproduzierbare Prüfung des Sourcecodes – man kann nicht vergessen, etwas getestet zu haben
- Wird ein Fehler gefunden, so wird er erst durch einen Testfall reproduziert und dann behoben

# Tests, Testfälle und Testsuiten

---

- Tests werden in Testfällen organisiert
- Ein Testfall
  - fasst alle Tests zu einem bestimmten Testgegenstand (Modul, inhaltlicher Kontext, Funktionsbaustein) zusammen
  - ist ein gewöhnliches Perlskript
  - Hat den Inhalt des Testziels im Namen
  - Hat üblicherweise die Dateiendung `.t`
  - befindet sich in einem Unterverzeichnis `t/`.
- Die Summe aller zusammengehörigen Testfälle werden Testsuite genannt
- Es können einzelne Testfälle oder eine gesamte Testsuite (siehe `Test::Harness`) ausgeführt werden

```
perl t/00_load.t # Einzelnen Testfall ausfuehren
prove -r         # Alle Tests einer Testsuite ausfuehren
```

# Getestet werden kann ...

---

- das Laden von Modulen
- das Ergebnis eines Funktionsaufrufs
  - Skalare Werte (Zahlen, Zeichenketten, etc.)
  - Komplexe Datenstrukturen (LoL, HoL, etc.)
- die Klassenzugehörigkeit von Objekten
- die syntaktische Korrektheit der Dokumentation
- die Kommunikation mit externen Systemen
- der Inhalt von Webseiten
- die Vollständigkeit der Tests (Testabdeckung)
- die Vollständigkeit einer Modul-Distribution
- ...

# Testwerte streuen

---

- Um die Stabilität der Software gegen “schlechtes Verhalten” zu erhöhen, sollten in Tests wahrscheinliche **und** unwahrscheinliche Werte getestet werden, z.B.
  - Minimaler und maximaler möglicher Wert sowie ein wenig Mehr und Weniger als diese Werte
  - Positive und negative Zahlen sowie 0, auch ganz Kleine
  - Leere Zeichenketten und mehrzeilige Zeichenketten
  - Zeichenketten mit Steuerzeichen, Nicht-ASCII- und UNICODE-Zeichen
  - undef und Listen von undef
  - Leere Listen, Arrays und Hashes
  - ...

# Ein einfacher Test

---

```
use Test::Simple tests => 1; # Wie viele Tests?
```

```
    # unsere zu testende Methode
```

```
sub getTwenty() {  
    return 20;  
}
```

```
ok(getTwenty() == 20, 'getTwenty() ist wirklich 20');  
    # ok() erwartet einen Wahrheitswert
```

Wird ausgeführt:

```
# perl examples/simple_test.t  
1..1  
ok 1 - getTwenty() ist wirklich 20
```

# Erste Prüfung: Anzahl der Tests

---

- Bei der Ausführung eines Testfalls wird geprüft, wie viele Tests geplant waren und wie viele ausgeführt wurden

```
use Test::Simple tests => 1; # Wie viele Tests?
```

- Ergebnisse:

- Korrekte Anzahl: Keine Ausgabe

- Mehr Tests geplant als ausgeführt:

```
# Looks like you planned 2 tests but only ran 1.
```

- Weniger Tests geplant als ausgeführt:

```
# Looks like you planned 1 test but ran 1 extra.
```

- Wenn nicht bekannt ist, wie viele Tests durchgeführt werden:

```
use Test::Simple tests => 'no_plan';
```

# Bessere Testvergleiche

---

- Mehr Möglichkeiten mit `Test::More`
- Tatsächliche Werte werden mit erwarteten Werten verglichen
  - `use_ok('Module::Name')` rüft das Laden von Modulen
  - `isa_ok($var, 'Module::Name')` prüft die Zugehörigkeit eines Objekts zu einer Klasse
  - `is($value, $expected)` vergleicht einen tatsächlichen Wert mit einem erwarteten Wert
  - `like($value, qr/regex/)` vergleicht einen tatsächlichen Wert mit einem regulären Ausdruck
  - `isnt($value, $expected)` vergleicht einen tatsächlichen Wert mit einem erwarteten Wert (Negation von `is()`)
  - `unlike($value, qr/regex/)` vergleicht einen tatsächlichen Wert mit einem regulären Ausdruck (Negation von `like()`)

# Bessere Testvergleiche

---

```
use Test::More tests => 6;
use_ok('My::Module') or exit; # Kann Modul geladen werden?

my $m = My::Module->new();
isa_ok($m, 'Parent::Type'); # Prüft OO-Vererbung

is($m->get_value(), 23, '$m is 23'); # Wertevergleich
like($m->get_name(), qr /^[A-Z]+$/); # Prüfung gegen Regex

isnt($m->get_value(), 0, '$m is not 0'); # Wertevergleich
unlike($m->get_name(), qr /^[^A-Z]+$/); # Prüfung gegen Regex
```

## Weitere Tests

---

Das CPAN stellt eine Vielzahl von Modulen für weitere Tests zur Verfügung, unter anderem:

<code>Test::Warn</code>	Testen auf das Auftreten von Warnings
<code>Devel::Cover</code>	Prüfen der Vollständigkeit von Tests (Testabdeckung)
<code>Test::Pod</code>	Syntaktische Prüfung der POD-Dokumentation
<code>Test::Distribution</code>	Prüfung der Modul-Distribution
<code>Test::Kwalitee</code>	Prüft die Qualität des Sourcecodes anhand von <i>Best Practices</i> , die die <i>Perl QA group</i> aufgestellt hat.

# Untestbaren Code testen

---

- Ein Test soll immer von der “Außenwelt” isoliert ausgeführt werden, so dass ausschließlich das Verhalten des zu testenden Codes bewertet wird
- Somit müssen z.B.:
  - Systemaufrufe abgefangen werden,
  - Netzwerkverbindungen emuliert werden,
  - externe Systeme (z.B. Datenbanken) emuliert oder durch interne Equivalente ersetzt werden,
  - Bestimmte eingebaute Perl-Funktionen ersetzt werden,

# Module (teilweise) durch Attrappen ersetzen

---

- Wenn in einem zu testenden Modul Funktionen aufgerufen werden, die mit der (nicht vorhersehbaren) Umgebung interagieren, dann können diese Funktionen im Test modifiziert werden
- `Test::MockModule` erlaubt es, im Test den Modulcode gezielt zu ändern
- Beispiel: Aufruf eines Systemkommandos in einem Medienabspieler-Modul
  - Methode `play()` ruft den externen Abspieler auf und liefert das Ergebnis des Systemaufrufs als Rückgabewert
  - Installationsort und Eigenschaften des konkreten Abspielers werden bei Installation der Medienabspieler-Moduls konfiguriert  $\Rightarrow$  im automatischen Test wird `play()` modifiziert

# Module (teilweise) durch Attrappen ersetzen

---

```
my $module = 'Media::Player';
use_ok($module) or exit;
my $player = $module->new();
isa_ok($player, $module);
# Some more tests
{
    # No some tests with mocked $player methods
    my $player = Mock::Module->new('Media::Player');
    $player->mock(play => sub($) { $uri = shift; 1});

    my $result = $player->play('/home/drue/mysong.ogg');
    is($result, 1, 'Could play ogg song');
}
# Further tests with unmodified $player
```

# Exkurs: Testgetriebene Softwareentwicklung

---

- Testgetriebene Softwareentwicklung (Test driven development) besagt, die Funktionalität eines Softwarebausteins und das gewünschte Ergebnis erst in einem Test zu definieren und dann in dem Softwarebaustein zu implementieren
- Ziele:
  - 100% Testabdeckung der Modul-API (!!!)
  - API des Moduls funktioniert genau wie bei Nutzung erwartet
  - Es wird nur die Funktionalität implementiert, die von darauf aufbauenden Anwendungen benötigt wird

# Exkurs: Testgetriebene Softwareentwicklung

---

- Vorgehensweise:
  - (1) Einen Testfall anlegen, den Funktionsaufruf im Modul und einen Test zu dem Funktionsaufruf definieren
  - Den Testfall ausführen
  - Der Compiler meldet einen Fehler, da der Funktionsaufruf in dem Modul nicht implementiert ist
  - (2) Die Funktion im Modul deklarieren (aber noch nicht implementieren)
  - Der Code wird fehlerfrei übersetzt und der Test ausgeführt
  - Test schlägt fehl, da der Rückgabewert der Funktion nicht mit dem erwarteten Ergebnis übereinstimmt
  - (3) Die Funktion in dem Modul implementieren
  - Der Code wird fehlerfrei übersetzt und der Test ausgeführt
  - Test läuft erfolgreich durch, da die Funktion nun implementiert wurde

# Exkurs: Testgetriebene Softwareentwicklung

---

- Nächste Funktion ...
- Jeder dieser Zyklen sollte möglichst kurz (wenige Minuten) dauern
- Spätere Änderungen am Programmcode zerbrechen nicht vorhandene Funktionalität

---

# Programmieren in Perl – ein Styleguide

- Perl stellt keine Anforderungen an Sourcecode-Strukturierung
- Einige Gestaltungsregeln haben sich – informell – etabliert
- Sie helfen, Programme leichter zu lesen, zu verstehen und zu pflegen

# Grundregeln

---

- Programme immer mit dem Pragma `use warnings` erstellen
- das Pragma `use strict` benutzen
- die Pragmas
  - `use sigtrap` zum Behandeln von Signalen
  - `use diagnostics` zum Erzeugen von Warnungen bei der Kompilationsind hilfreich

# Sourcecode-Formatierung

---

- 4 Zeichen Einrückungstiefe
- Bei mehrzeiligen Anweisungsblöcken ist die schließende Klammer } nach dem eröffnendem Schlüsselwort ausgerichtet
- Öffnende Klammer { in der gleichen Zeile wie Anweisung
- Leerzeichen vor { bei mehrzeiligen Anweisungsblöcken
- einzeilige Anweisungen kommen inklusive { } in eine Zeile
- Kein Leerzeichen vor Semikolon
- Kein Semikolon in “kurzen” einzeiligen Anweisungen
- Leerzeichen um Operatoren

# Sourcecode-Formatierung

---

- Leerzeichen um komplexe Ausdrücke innerhalb [ ]
- Leerzeile zwischen Code-Blöcken
- Kein Leerzeichen zwischen Funktionsname und öffnender Klammer der Argumentliste
- Leerzeichen nach jedem Komma
- Lange Zeilen nach Operatoren umbrechen (außer and und or)
- Korrespondierende Elemente vertikal ausrichten
- Automatisierte Unterstützung: <http://perltidy.sourceforge.net/>

# Variablen und Bezeichner

---

- Das Modul `English` bietet lesbare Bezeichner für Perls eingebaute Variablen
- Verständliche, erklärende Bezeichner wählen
- mehrere Wörter im Bezeichner durch `_` trennen
- Benennung von Variablen:
  - `$ALL_CAPS_HERE`      Konstanten
  - `$Some_Caps_Here`      Package-weite globale Variablen (OO: static)
  - `$no_caps_here`      Lokale Variablen

# Anweisungen

---

- Anweisungen inhaltlich sinnvoll konstruieren

```
open(my $file , $foo) || die "Can't open_$foo:_$!";    # ++  
die "Can't open_$foo:_$!" unless open($file , $foo);# --
```

und

```
print "Starting_analysis\n" if $verbose;              # ++  
$verbose && print "Starting_analysis\n";              # --
```

- In größeren Programmen nicht auf implizite Vorgabewerte von Funktionen vertrauen (z.B. bei `split ()` und `join ()`)
- Klammern ( ) können die Lesbarkeit von Funktionsaufrufen erhöhen
- Komplexe reguläre Ausdrücke mit `/x` kommentieren

# Anweisungen

---

- `grep()` und `map()` nicht im `void`-Kontext nutzen
- Zum Iterieren über Listen dienen `for` und `foreach`
- Bei größeren Anweisungsblöcken genügt eine Anweisung je Zeile
- Unsicheren Code (auch Betriebssystem-abhängigen Code) in `eval { }` ausführen

# Funktionen und Operatoren

---

- Alle eingebauten Funktionen liefern auswertbare Rückgabewerte!
- Eigene Funktionen sollten das auch tun!
- return ohne Argument signalisiert dem Aufrufenden, dass Funktion fehlgeschlagen ist
- Rückgabewerte von Systemaufrufen prüfen
- Funktionsnamen immer mit Kleinbuchstaben
- and und or sind bequemer als || und &&
- eigene Funktionen wie eingebaute Funktionen aufrufen

```
&myfunc $arg1 , $arg2 ;      # --  
myfunc( $arg1 , $arg2 );    # ++
```

# Funktionen und Operatoren

---

- HERE-Documents sind besser lesbar als wiederholte `print ()`-Anweisungen

```
print <<"END_OF_MSG" ;  
Hallo Welt , auf dem Rechner läuft $^O! :-)  
END_OF_MSG;
```

- Erzeugte Fehlermeldungen nach `STDERR` ausgeben sowie schlüssig und inhaltlich aussagekräftig sein
- Eigene Fehlermeldungen sollten deren ursprüngliche Fehlermeldung mit ausgeben

```
opendir(D, $dir)  
or die "can't opendir \"$dir:$!"; # So einfach!
```

# Packages, Module und Objekte

---

- Mindestanforderung an Perl-Version definiert

```
use 5.006001;
```

- Mindestanforderung an Modul definiert

```
use Text::HTML 2.40;
```

- Version eines Moduls definiert

```
package Text::HTML;  
$VERSION = 2.40;
```

- Package- und Modul-Namen beginnen mit Großbuchstaben
- Nur Pragmas beginnen mit Kleinbuchstaben
- Package-Name stimmt mit dem Pfadnamen des Moduls überein  
(:: wird zu /)

# Packages, Module und Objekte

---

- private Funktionen und Variablen eines Packages mit `_` beginnen (informell)
- `carp()` und `croak()` aus dem Carp-Modul statt `warn()` und `die()` anwenden
- Module dokumentieren (siehe `perldoc perlpod`) – damit wird die öffentliche Schnittstelle des Moduls (API) publiziert
- Schreib Tests für jeden Funktionsbaustein! (siehe `perldoc Test`)

# Test::Kwalitee - Best Practices nutzen

---

- Das Perl QA team hat eine ganze Reihe von Best Practices zusammengestellt, die befolgt werden sollten, um stabilen, portable, wartbaren Sourcecode zu erstellen
- Einstiegspunkt: <http://cpants.perl.org/kwalitee.html>
- Es wurden 32 *Kwalitee Indicators* aufgestellt, sie umfassen
  - Bestandteile der Distribution - Readme, Buldskript, Lizenz, ...
  - Nutzung und Vermeidung von bestimmten Sprachkonstrukten
  - Korrektheit der Dokumentation und Tests
  - und viel mehr!
- Diese Qualitätsaussagen werden erstklassig in dem Buch "*Perl Best Practices*" von Damian Conway aus dem O'Reilly-Verlag beschrieben (ISBN: 0596001738)

## Kwalitee Indicators Overview

Available Kwalitee: 23 (including optional metrics: 32)

### Core Indicators

- buildtool\_not\_executable
- extractable
- extracts\_nicely
- has\_buildtool
- has\_changelog
- has\_humanreadable\_license
- has\_manifest
- has\_meta\_yaml
- has\_proper\_version
- has\_readme

- has\_tests
- has\_version
- has\_working\_buildtool
- manifest\_matches\_dist
- metayml\_conforms\_to\_known\_spec
- metayml\_has\_license
- metayml\_is\_parsable
- no\_cpants\_errors
- no\_generated\_files
- no\_pod\_errors
- no\_symlinks
- proper\_libs
- use\_strict

## Optional Indicators

- `has_example`
- `has_test_pod`
- `has_test_pod_coverage`
- `has_tests_in_t_dir`
- `is_prereq`
- `metayml_conforms_spec_current`
- `no_stdin_for_prompting`
- `prereq_matches_use`
- `use_warnings`

---

# Perl Tipps

- Das Carp-Modul
- Fehlersuche in Perl
- Hilfe im Internet
- Buch-Tipps

# Das Modul Carp

---

- Carp-Routinen agieren wie `warn()` und `die()`, aber sie geben an, wo im Code der Fehler passierte.
- Für Debugging-Zwecke kann der Stacktrace (die rekursive Liste der Funktionsaufrufe) mit ausgegeben werden

Standard-Funktion	Carp-Funktion	Carp-Funktion mit Stacktrace
<code>die()</code>	<code>croak()</code>	<code>confess()</code>
<code>warn()</code>	<code>carp()</code>	<code>cluck()</code>

- `cluck()` muss explizit importiert werden
- Für CGI-Programmierung existiert ein analoges Modul `CGI::Carp`

# Das Modul Carp

---

- Stacktrace-Funktionen können direkt aufgerufen werden, bzw. durch Importieren des Symbols `verbose`

- im Skript

```
use Carp qw(:DEFAULT verbose);  
# importiert croak confess carp verbose
```

- durch Übergabe an der Kommandozeile

```
perl -MCarp=verbose script.pl
```

- durch Einfügen von "MCarp=verbose" in die Umgebungsvariable `PERL50PT`.

# Das Modul Carp

---

```
use Carp;  
croak "No arguments given!" unless @ARGV;  
carp "Only one Argument— that's bad!"  
    if (scalar @ARGV == 1);
```

# Fehlersuche in Perl

---

- Aufruf von Perl mit Warnmeldungen

```
perl -w perlskript.pl
```

oder Angabe im She-Bang (erste Programmzeile – nur bei Unix)

```
#!/usr/bin/perl -w
```

- Aufruf des Perl-Debuggers (siehe Hilfe-Seite `perldebug`)

```
perl -d perlskript.pl
```

- Einbauen von Fehlermeldungen im Programm

```
print STDERR "$0: _Probleme!" if $PROBLEM;
```

# Fehlersuche in Perl

---

- Nutzen von Logging-Frameworks (Siehe CPAN)
  - Log-Log4perl – Log4J-Implementierung für Perl
  - Log::Logger – OO Interface für benutzerdefiniertes Logfile
- Berechtigungen des Skripts prüfen
  - Unix: `chmod 0755 perlskript.pl`
  - Win32: `PATHEXT=.pl;.com;.exe;.bat`
- Ist das Script Standard-Perl

```
perl -wc perlskript.pl
#      ^ nur kompilieren
```
- Sind die eingebunden Module in der erforderlichen Version vorhanden

```
perl -MText::HTML -e 'print Text::HTML->VERSION'
2.40
```

# Hilfe zu Perl

---

- Perl besitzt ein umfangreiches Hilfe-System
- Einzelne Sektionen beschreiben Sachverhalte zu Perl, z.B.
  - perl – Allgemeine Informationen
  - perlsyn – Syntax von perl, Kontrollstrukturen, Schleifen
  - perlop – Alles über Operatoren
  - perlfunc – Perl's eingebaute Funktionen
  - perldata – die verschiedene Datentypen
  - perlsub – Arbeiten mit benutzerdefinierten Funktionen
  - perlre – reguläre Ausdrücke (Perl's großer Trumpf ;-)
  - perllocal – lokal installierte Packages

# Hilfe zu Perl

---

- Aufruf der Hilfe an der Kommandozeile

```
perldoc perlsektion
```

- und außerdem: Aufruf der Hilfe-Seiten unter Unix

```
man perlsektion
```

- oder: Aufruf unter Windows (ActivePerl) durch Browsen der HTML-Seiten im Unterverzeichnis `html` der Perl-Installation

- Aufruf der Hilfe zu einer Perl-Funktion

```
perldoc -f function
```

# Hilfe im Internet

---

- <http://www.perl.com> – die Zentrale
- <http://www.perl.org> – Sammlung von Links
- <http://www.cpan.org> – Software-Archiv
- <http://www.perl.de> – die deutschsprachige Perl-Community
- [news:comp.lang.perl.\\*](news:comp.lang.perl.*) – News-Foren

# TTLO – Wofür kein Platz im Zeitplan war

---

- Perl und XML
- Objekt-Orientierte Programmierung in Perl
- Web-technologien, Webservices
- GUI-Programmierung (Gtk, Qt, Tk, ...)
- Bindings zu anderen (System-)Bibliotheken
- Anbindung von externen Informationssystemen (Datenbanken, LDAP-Server)
- ...

# Lesestoff

---

- Damian Conway: *Perl Best Practices*. O'Reilly, 1. Auflage, 2005.
- Sriram Srinivasan: *Advanced Perl Programming*. O'Reilly, 2005.
- Larry Wall, Tom Christiansen, Jon Orwant: *Programming Perl*. 3. Auflage, O'Reilly, 2000.
- Ian Langwoth, chromatic: *Testing Perl – A Developer's Notebook*. O'Reilly, 1. Auflage, 2005.
- Jon Orwant, Jarkko Hietaniemi, John Macdonald: *Algorithmen mit Perl*. O'Reilly, 1. Auflage, 2000.
- Johan Vromans: *Perl 5 kurz und gut*. O'Reilly, 4. Auflage, 2003.
- Tom Christiansen, Nathan Torkington: *Perl Cookbook*. O'Reilly, 2. Auflage, 2003.
- The Perl Journal, <http://www.tpj.com/>