Perl Einführung

Inhaltsverzeichnis

1 Skalare Variablen

1.1 Skalare Variablen : Strings

Variablenzuweisung

$str = "mein string"	$str = qq(mein string)
$str = 'mein string'	$str = q(mein string)

Expansion von Variablen

$var = "mein string"; $str = "das ist $var";
=> $str = "das ist mein string"
Expansion mit "

$var = "mein string"; $str = 'das ist $var';
=> $str = "das ist $var"
KEINE Expansion mit '

Zeichen mit Sonderbedeutung

$ @ % # \ & ;

Diese Zeichen müssen mit führendem \ in einem String angegeben werden, wenn sie als darstellbares Zeichen verwendet werden. Darüber hinaus gibt es spezielle Steuerzeichen gemäß ANSI.

Konkatenation

Strings können zusammengeführt werden mit dem "**.**"-Operator (sprich : Punkt-Operator)
`$var = "das"; $str = "Leben"; $erg = $var.$str;` => $erg = "dasLeben"

Kombinationen sind möglich : $erg = "das".$str;

Der Wiederholungsoperator x

Beispiel : `$line = "#" x 5;` => $line = "#####"
Der W. dient zur abkürzenden Schreibweise und wird z.B. bei der formatierten Ausgabe verwendet (zur Erzeugung von Linien u.ä.).

HERE - Dokumente

HERE - Dokumente bieten die Möglichkeit, lange Strings zu erzeugen ohne daß man sich um Zeilenumbrüche kümmern muß. Der String wird einfach mehrzeilig aufgeschrieben und durch eine Dokumentendung abgeschlossen. Diese Endung muß alleine und am Anfang der letzten Zeile stehen. Beispiel :

```
$adressat = "Mustermann";
 $brief = <<"ENDE_VON_BRIEF";
Sehr geehrter Herr $adressat !
In sogenannten HERE-Dokumenten findet eine Expansion von Variablen
stattfindet, wenn die Dokumentendung ( ENDE_VON_BRIEF ) in " " steht.
ENDE_VON_BRIEF
```

1.2 Funktionen auf Strings

chomp(*string*)

chomp($str) : entfernt Zeilenendezeichen ("\n") Beispiel : chomp($str = "geheim\n") => $str = "geheim"

length(*string*)

$anz = **length**($str) : liefert die Länge des Strings Beispiel : $str = "arbeiten"; $anz = length($str) => $anz = 8

index(*string,substring,offset)*

$pos = **index**($str1,$str2,$off) : liefert Position von $str2 in $str1 ab $ind $str1 = "Bananas"; $str2 = "na"; $off = 3; Beispiel 1 : $pos = index($str1,$str2); => $pos = 2
Beispiel 2 : $pos = index($str1,$str2,$off); => $pos = 4 *index liefert -1, wenn $str2 nicht in $str1 enthalten ist*

substr(*string,offset,length*)

$teil = **substr**($str,$off,$len)
liefert Teilstring von $str der Länge $len ab Position $off
substr($str,$off,$len) = $ins_str
verändert $str ab Position $off um $len Zeichen $str = "Schiff"
Beispiel 1 : $teil = substr($str,4) => $teil = "Schi"
Beispiel 2 : $teil = substr($str,2,3) => $teil = "hif"
Beispiel 3 : substr($str,3,0) = "l" => $str = "Schliff" , $len = 0 : einfügen
Beispiel 3a : substr($str,3) = "l" => $str = "Schl" , ab $off=3 alles ersetzen
Beispiel 4 : substr($str,3,1) = "l" => $str = "Schlff"
Beispiel 5 : substr($str,3,2) = "l" => $str = "Schlf"
Beispiel 6 : substr($str,3,3) = "l" => $str = "Schl"

lc , lcfirst uc , ucfirst

lc : **l**ower **c**ase, Kleinschreibung	**uc** : **u**pper **c**ase, Großschreibung
$str = **lc**("HALLO"); => $str = "hallo"	$str = **uc**("hallo"); => $str = "HALLO"
$str = **lcfirst**(WILLY); => $str = "wILLY"	$str = **ucfirst**("willy"); => $str = "Willy"

eval "*string*"

eval $str : $str enthält gültigen Perlcode.
eval ruft den Perlinterpreter mit diesem Code auf.
Damit ist es möglich, Perlcode *zur Laufzeit* zu erzeugen und auszuführen.
Beispiel :
$op = "**";
$rec = **eval** "3 $op 2";
=> $rec=9

1.3 Skalare Variablen : Zahlen

Variablenzuweisung

$dig = 4711;

Automatische Konvertierung

$str = "7"; $erg = 5 + $str; => $erg = 12
$str = "a"; $erg = 8 + $str; => $erg = 8, denn "a" wird zu 0 konvertiert

Rechenoperationen

Operation	Operator	Beispiel	Ergebnis
Addition	**+**	$dig = 1 + 1;	=> $dig = 2
Subtraktion	**-**	$dig = 5 - 2;	=> $dig = 3
Multiplikation	*	$dig = 5 * 2;	=> $dig = 10
Division	/	$dig = 5 / 3;	=> $dig = 1.6666...
Potenzieren	**	$dig = 6 ** 2;	=> $dig = 36
Modulo"ganzzahliger Rest"	**%**	$dig = 13 % 4;$dig = 18 % 5	=> $dig = 1=> $dig = 3

Inkrementierung

Das automatische Erhöhen eines Wertes ist eine häufig angewandte Operation (Zähler)
$i = 0;

$i**++**;	erhöht $i um 1 *nach* Ausführung dieser Anweisung
++$i;	erhöht $i um 1 *vor* Ausführung dieser Anweisung

Beispiel : print $i++," ",$i; => 0 1
Beispiel : print ++$i," ",$i; => 1 1

Kurzformen

$zaehl = 24 :

Operation	Kurzform	Ergebnis
$zaehl = $zaehl + 3	$zaehl **+=** 3	=> $zaehl = 27
$zaehl = $zaehl - 5	$zaehl **-=** 5	=> $zaehl = 19
$zaehl = $zaehl * 2	$zaehl ***=** 2	=> $zaehl = 48
$zaehl = $zaehl / 4	$zaehl **/=** 4	=> $zaehl = 6
$zaehl = $zaehl."abc"	$zaehl **.=** "abc"	=> $zaehl ="24abc"

Zahlensysteme

Hexadezimal	$hexadz = 0x7e;	führend : **0x**
Oktal	$oktalez = 072;	führend : **0**
Binär	$binaer = 0b1011;	führend : **0b**

1.4 Funktionen auf Zahlen

Arithmetische Funktionen

Funktion	Syntax	Beispiel	Ergebnis
Absolutwert	**abs**($zahl)	$dig = abs(-100);	=> $dig = 100
ArcusTangens	**atan2**($x,$y)	$dig = atan2(3.14,1);	=> $dig = 1.26...
Cosinus	**cos**($zahl)	$dig = cos(3.14/4);	=> $dig = 0.707...
Sinus	**sin**($zahl)	$dig = sin(3.14/2);	=> $dig = 0.9999996...
Logarithmus	**log**($zahl)	$dig = log(2.718);	=> $dig = 0.999896...
Exponent	**exp**($zahl)	$dig = exp(1);	=> $dig = 2.718...
Wurzel	**sqrt**($zahl)	$dig = sqrt(9);	=> $dig = 3
Integer	**int**($zahl)	$dig = int(2.542);	=> $dig = 2
Zufallszahl	**rand**($zahl)	$dig = rand(10);	=> 0 <= $dig < 10

Konvertierung

$zahl wird immer zuerst in eine Dezimalzahl umgerechnet, dann wird konvertiert !

Funktion	Syntax	Beispiel	Ergebnis
Hexadezimal	**hex**($zahl)	$dig = hex(0x1e)$dig = hex(30)	=> $dig = 48=> $dig = 48
Oktalzahl	**oct**($zahl)	$dig = oct(0x1e)$dig = oct(30)	=> $dig = 24=> $dig = 24
ASCII-Zeichen	**chr**($zahl)	$var = chr(65)	=> $var = "A"
ASCII-Wert	**ord**($zeichen)	$var = ord(B)	=> $var = 66

2 Strukturblöcke

2.1 Bedingungen

Die Werte true und false in Perl

false : "" , "0" , 0, *undef*, () true : sonst (u.a. "00", "7", "a", 5) Insbesondere können Rückgabewerte von Funktionen als Boolesche Ausdrücke verwendet werden.

Vergleichsoperatoren für Strings

Vergleich	**Operator**	**Beispiel**	**Ergebnis**
Gleichheit	**eq**	("hallo" eq "hallo")	true
Ungleichheit	**ne**	("hallo" ne "hallo")	false
Kleinergleich	**le**	ASCII-Wert Vergleich	unsinnig
Größergleich	**ge**	ASCII-Wert Vergleich	unsinnig
Kleiner	**lt**	ASCII-Wert Vergleich	unsinnig
Größer	**gt**	ASCII-Wert Vergleich	unsinnig

Vergleichsoperatoren für Zahlen

Vergleich	**Operator**	**Beispiel**	**Ergebnis**
Gleichheit	**==**	(7 == 8)	false
Ungleichheit	**!=**	(9 != 10)	true
Kleinergleich	**<=**	(56 <= 55)	false
Größergleich	**>=**	(22 >= 22)	true
Kleiner	**<**	(5 < 6)	true
Größer	**>**	(8 > 8)	false

Warnung

Bei Verwendung von Variablen in Vergleichen ist auf die automatische Konvertierung zu achten :

$str = "abc";(0 == $str) ? print „OK“ : print „NOK“; OK, Zahl:"abc" => 0

$str = "abc";(0 eq $str) ? print „OK“ : print „NOK“; NOK, String: 0 => "0"

Logische Operatoren

and&&	true	false
true	true	false
false	false	false

or||	true	false
true	true	true
false	true	false

xor^	true	false
true	false	true
false	true	false

not!	true	false
	false	true

2.2 Verzweigungen

if - Verzweigungen

```
if (Bedingung)
{ Perlcode, wenn Bedingung true}
else { Perlcode, wenn Bedingung false}
```

Erweiterte Bedingungsabfrage (*select-case*)

```
if (Bedingung 1)
{ Perlcode, wenn Bedingung 1 true}
elsif (Bedingung 2) { Perlcode, wenn Bedingung 1 false, 2 true}
elsif (Bedingung 3) { Perlcode, wenn Bedingung 1 und 2 false, 3 true}
else { Perlcode, wenn keine Bedingung true }
```

Kurzformen

Besteht der auszuführende Perlcode im *true*-Zweig aus genau einer Anweisung, dann kann verkürzt geschrieben werden

Anweisung **if (***Bedingung***)**

Beispiel :

```
print $ergebnis if ( $str1 eq $str2 );
```

+

Besteht der auszuführende Perlcode im *true* **und** im *false*-Zweig aus genau einer Anweisung, dann kann verkürzt geschrieben werden

(*Bedingung***) ?** *Ausdruck(true)* **:** *Ausdruck(false)*;

Beispiel :

```
( $pass eq "geheim" ) ? print "OK" : exit 0;
```

Bedingungsabfrage mit unless

unless negiert die if-Logik. `unless (Bedingung) { ... }`
ist identisch mit `if !(Bedingung) { ... }`
Das gilt auch für die Kurzform.

Hinweise

Die **if** - Abfrage leitet einen eigenen Codeblock ein, der seperat vom übrigen Code betrachtet werden kann. Optisch macht man das kenntlich, indem man den Codeblock im Quellcode einrückt:

```
Anweisung
if ( ... ){
    Anweisungen im true-Zweig
}
else {
    Anweisungen im false-Zweig
}
Anweisungen ...
```

2.3 Schleifen

for (*Startwert;Abbruchbedingung;Inkrement*) { ...}

Der Schleifenblock { ... } wird so oft abgearbeitet, bis die Abbruchbedingung *nicht* mehr erfüllt ist.

- Startwert : Zuweisung eines Wertes an eine Variable (*Laufvariable*)
- Abbruchbedingung : Vergleich von Zahlen, bei *false* erfolgt Abbruch
- Inkrement : Anweisung zur Veränderung der Laufvariablen bei jedem Schleifendurchlauf
- Die Steuerung der Schleife wird vollständig durch die drei Angaben bestimmt

Beispiel :

```
for ($i = 0;$i <= 10;$i++)
{ print $i,"\n" }
```

Hinweise : Im Schleifenblock sollte die Laufvariable *nicht* mehr verändert werden.
Mit for (1..10) { print $i,"\n" } erfolgt die gleiche Ausgabe wie im Beispiel.
Um den Schleifenblock auch optisch im Code hervorzuheben, bietet sich das Einrücken an :

```
Anweisung
Anweisung
for ($i=0;$i<=100;$i+=2) {
 Anweisung
 Anweisung
 mach was mit $i
}
Anweisungen ...
```

while (*Abbruchbedingung*) { ... }

Der Schleifenblock **{ ... }** wird solange abgearbeitet, bis die Abbruchbedingung *false* ergibt. Die erste Bedingungsprüfung erfolgt *vor dem ersten Abarbeiten des Schleifenblockes* (Kopfschleife)

Beispiel :

```
while ($i<=10) { print $i++,"\n" }
```

Hinweise :
Die Laufvariable *muß* im Schleifenblock verändert werden, wenn die Abbruchbedingung erreicht werden will.
Wie bei der *for*-Schleife ist ein Einrücken des Schleifenblockes zweckmäßig. Der Schleifenblock wird mit "}" beendet, ein folgendes ";" ist nicht nötig.

2.4 Andere Schleifen

do { ... } while (*Abbruchbedingung*)

Die erste Bedingungsprüfung erfolgt *nach einmaligem Abarbeiten des Schleifenblockes* (Fußschleife) d.h., diese Schleife wird *immer einmal* durchlaufen. Es gelten die gleichen Hinweise wie zur while-Schleife.

Minischleifen

Besteht der Schleifenblock aus genau einer Anweisung, dann kann verkürzt geschrieben werden:

Anweisung ***while*** *Bedingung*

Beispiel 1: `print $i++ while $i <= 12;`
Beispiel 2: `print $line = <STDIN> while $line ne "\n";`

foreach $lauf (*Liste*) { ... } [obsolet,im Hinblick auf Perl6]

Die foreach-Schleife arbeitet nur auf Listen.
In jedem Schleifendurchlauf wird ein Listenelement der Laufvariable zugewiesen.
Wird die Laufvariable nicht explizit angegeben, so wird die perlinterne Laufvariable **$_** verwendet.

3 Datenstrukturen

3.1 Listen

Definition

Eine Liste ist eine geordnete Folge von skalaren Werten. Bemerkung : Variablen, die Listen enthalten, heißen Arrays

Eigenschaften

- Schreibweise : **(** *1.Element*,*2.Element*,...**)**
  Die Elemente einer Liste werden durch "**,**" voneinander getrennt.
- Polymorphie : Eine Liste kann unterschiedliche Datentypen enthalten
  Beispiel : ("string",1,3,1.54,"gamma")
- Keine Schachtelung : (1,4,(13,5),8) identisch (1,4,13,5,8)

Anwendungsgebiete

- Parameterübergabe an Funktionen/Subroutinen
- Komprimierte Zuweisung : ($a,$b,$c) = (1,2,3);
- Häufigste Anwendung in Zusammenhang mit Arrays

Zugriff auf Elemente

Beispiel 1	$w = (5,6,7,8)**[**2**]**	=> $w = 7
Beispiel 2	($v,$w) = (5,6,7,8)**[**1,3**]**	=> ($v,$w) = (6,8)
Beispiel 3	($u,$v,$w) = (5,6,7,8)**[**0..2**]**	=> ($u,$v,$w) = (5,6,7)
Beispiel 4	($u,$v,$w) = (5..7)**[**0..2**]**	=> ($u,$v,$w) = (5,6,7)

Bereichsoperator "**..**" steht für *von .. bis*

3.2 Arrays

Definition

Arrays sind Variablen, die Listen speichern Bemerkung : Die Elemente eines Arrays sind Skalare

Zuweisung

@array = *Liste*;
@array = (6,8,9)
$array[2] = 9; ***Skalar*** *!*
@array[1,2] = (8,9);

Beispiele für Listen : eine Zahlenfolge, die Zeilen einer Datei, Rückgabewerte einer Funktion/Subroutine, eingelesene Pufferinhalte

Dimension (Anzahl der Elemente)

Der Ausdruck **$#array** liefert den höchsten Index von @array
Beispiel : @array = (5,4,6,1) => $#array = 3
Die Zuweisung **$zahl = @array** liefert die Anzahl der Elemente (= $#array + 1)

ACHTUNG : Das Array wird hierbei im *skalaren Kontext* gelesen und hat nichts mehr mit den Elementen des Arrays zu tun ! Skalare Variablen beginnen immer mit $

3.3 Funktionen auf Arrays 1

sort *Liste*

Man unterscheidet Sortierung nach ASCII-Zeichen und nach Zahlen :

@sort_list = **sort** (@unsort_list)	ASCII-Sortierung, Normalfall
@sort_list = **sort** { $a cmp $b } @ul	ASCII-Sortierung
@sort_list = **sort** { $a <=> $b } @ul	Numerische Sortierung
@sort_list = **sort** { $b cmp $a } @ul	ASCII-Sortierung, absteigend
@sort_list = **sort** { $b <=> $a } @ul	Numerische Sortierung, absteigend

reverse *Liste*

@a = **reverse** qw(a b c); => @a = ("c","b","a")

split *Trennzeichen* , String*

Mit der Funktion **split** wird ein String an allen Positionen eines *Trennzeichens* zerschnitten und in mehrere Strings aufgeteilt. Die neuen Strings bilden eine Liste. Das Trennzeichen taucht nicht mehr auf.

Beispiel : $line = "hugo:*:507:101:Hugo Hugendubel:/home/hugo:/bin/sh"; @pw = **split**(":",$line); => @pw = ("hugo","*","507","101", ...)

Die Elemente des Arrays sind Skalare

join *Trennzeichen* , Liste*

Mit der Funktion **join** werden die Elemente einer Liste zu *einem* String zusammengefügt. Zwischen die einzelnen Elemente kann dabei ein Trennzeichen eingefügt werden.

Beispiel : @user_items = ("willy","*","508","101","Willy Wunz","/home/willy","/bin/bash"); $new_user = **join**(":",@user_items); => $new_user = "willy:*:508:101:Willy Wunz:/home/willy:/bin/bash";

- : Als Trennzeichen sind alle ASCII-Zeichen erlaubt, auch Steuerzeichen wie z.b. "\n"

3.4 Funktionen auf Arrays 2

push *Array , Liste*

Mit der Funktion **push** wird ein oder mehrere Elemente an ein Array angefügt.

Beispiel :
```
@dig = (1,2,3);
push(@dig,4);
```
=> @dig = (1,2,3,4)

pop *Array*

Mit der Funktion **pop** wird das letzte Element aus einem Array entfernt.

Beispiel :
```
@dig = (1,2,3);
$snap =pop(@dig);
```
=> @dig = (1,2) *und* $snap = 3

shift *Array*

Mit der Funktion **shift** wird das erste Element aus einem Array entfernt.

Beispiel :
```
@dig = (1,2,3);
$snip = shift(@dig);
```
=> @dig = (2,3) *und* $snip = 1

unshift *Array , Liste*

Mit der Funktion **unshift** wird einem Array ein oder mehrere Elemente vorangestellt.

Beispiel :
```
@dig = (1,2,3);
unshift(@dig,4,5);
```
=> @dig = (4,5,1,2,3)

splice *Array Offset Length Newarray*

Die Funktion **splice** entfernt aus *Array* den Bereich *Array[Offset] - Array[Offset+Length]* und ersetzt diesen Bereich duch *Newarray*. Der entfernte Bereich wird zurückgeliefert.

Beispiel :
```
@array = (1,2,3,4,5,6);
@inarr = qw(a b c);
@remov = splice(@array,2,2,@inarr);
```

=> @array = (1,2,"a","b","c",5,6)
und @remov = (3,4)

Bemerkung : *Array* muß ein Array sein, *Newarray* kann auch eine Liste sein.

3.5 Funktionen auf Arrays 3

grep : Extraktion

- ... extrahiert bestimmte Elemente eines Arrays
- ... wendet auf jedes Element eine Funktion an
- ... extrahiert ein Element ($_), wenn die Funktion *true* liefert
- ... liefert eine Liste mit den extrahierten Elementen
- ... verändert **nicht** die Elemente des Origanal-Arrays
- ... verändert **nicht** das Original-Array

Beispiel :

```
@orig_array = (1..10);
@new_array = grep { $_ % 2 } @orig_array;
```

=> @orig_array = (1,2,3,4,5,6,7,8,9,10)
=> @new_array = (1,3,5,7,9)

allgemein gilt $#orig_array >= $#new_array

map : Modifikation

- ... modifiziert jedes Element einer Liste
- ... liefert eine Liste mit den verarbeiteten Elementen
- ... verändert **nicht** die Elemente des Original-Arrays
- ... verändert **nicht** das Original-Array

Beispiel :

```
@orig_array = (1..10);
@modi_array = map { $_ % 2 } @orig_array;
```

=> @orig_array = (1,2,3,4,5,6,7,8,9,10)
=> @modi_array = (1,0,1,0,1,0,1,0,1,0)

allgemein gilt $#orig_array = $#new

3.6 Hashes

Definition

Ein Hash ist ein string-indiziertes Array.

Bezeichnungen

Der **Index eines Hashes** wird als **key** oder deutsch als **Schlüssel** bezeichnet
der **Wert eines Hashes zu einem key** wird als **value** bezeichnet.

Zuweisung

```
%hash = ( "key1" => "value1","key2" => "value2" ...);
```

Auch die Listenzuweisung ist möglich :

```
%hash = ("key1","value1","key1","value1");
%hash = @array;
```

Dabei werden die Elemente der Liste paarweise gelesen (*key-value-pair*)

einzelne Elemente zuweisen (Skalare):

```
$hash { "key17" } = "achtzehn";
$hash { "hallo" } = "wie geht's?";
$hash { "From" } = "willy@hash.pipe.net"
```

=> %hash = ("key17"=>"achtzehn","hallo"=>"wie
geht's?","From"=>"willy@hash.pipe.net")

Dabei ist die Eingabereihenfolge nicht mehr gewährleistet (s.u.)

Zugriff auf Elemente

$var = $hash**{**"key1"**}**; *#Skalar !* => $var = "value1"

Reihenfolge der Elemente

Die Elemente eines Hashes werden nicht in der Reihenfolge abgespeichert wie sie definiert wurden. Besonders beim Durchlauf durch alle key-value-pairs ist die Reihenfolge scheinbar beliebig.

.

3.7 Funktionen auf Hashes

reverse *Hash*

Die Funktion **reverse** vertauscht *key* und *value* für jeden Hasheintrag.

Beispiel :

```
%dig_to_spell = ( 1 => "eins",2 => "zwei" );
%spell_to_dig = reverse %dig_to_spell;
```

=> $spell_to_dig{"eins"} = 1 *und* $spell_to_dig{"zwei"} = 2

keys *Hash*

Die Funktion **keys** liefert eine Liste aller keys eines Hashes.

Beispiel :

```
%dig_to_spell = ( 1 => "eins" , 2 => "zwei" , 3 => "drei" );
@schluessel = keys %dig_to_spell;
```

=> @schluessel = (1,2,3)

values *Hash*

Die Funktion **values** liefert eine Liste aller values eines Hashes.

Beispiel :

```
%dig_to_spell = ( 1 => "eins" , 2 => "zwei" , 3 => "drei" );
@werte = values %dig_to_spell;
```

=> @werte = ("eins","zwei","drei")

each *Hash*

Die Funktion **each** liefert im *Listenkontext* eine zweielementige Liste (key,value)
Die Funktion **each** liefert im *skalaren Kontext* den nächsten Schlüssel analog zu **keys**:

Beispiel
Listenkontext :

```
while ( ($schl,$wert) = each %hash )
{ mach was mit $schl und $wert ...}
```

sort keys *Hash*, sort values *Hash*

Eine Sortierung von Hashkeys oder -values ist nicht direkt möglich. Die keys bzw. values

werden durch die Funktionen in einem Array gesammelt und dann sortiert.

Beispiel : `foreach (sort keys %hash ) { tu was mit $_ sortiert ... }`

3.8 Referenzen

Der allokierte Speicherbereich für eine Variable (Skalar, Array, Hash) kann durch den Varibalennamen angesprochen werden. Dieser Varablenname kann lokal oder global gültig sein. Ist der Varablenname lokal gültig (z.B. in einer Funktion) so stirbt der Varablenname nach Abarbeitung der Funktion, der Speicherbereich wird wieder freigegeben.

Eine Referenz auf eine Variable bietet die Möglichkeit, auf den allokierten Speicherbereich zuzugreifen (vgl. Call by Reference bei Parameterübergabe). Wird eine globale Variable an ein Unterprogramm übergeben, so wird der Speicherbereich in den Code hineinkopiert, wird eine Referenz übergeben, dann wird nur die Speicheradresse des Objekts übergeben.

Wird nun eine Referenz auf eine lokale Variable definiert, so stirbt der Variablenname, wenn das Programm den Gültigkeitsbereich der Variablen verläßt (i.A. nach Beendigung der Funktion). Da aber eine Referenz existiert, wird der Speicherbereich nicht freigegeben. Dies geschieht erst mit dem Sterben der letzten Referenz auf das Hauptspeicherobjekt. Damit ist der Varablenname nicht mehr verwendbar, der Speicherbereich über die Referenz sehr wohl.

Wird nun eine Referenz aus einer Funktion heraus zurückgegeben, so spielt der ursprüngliche Name der Variablen keine Rolle. Ebensogut kann man für das erzeugte Speicherobjekt in der Funktion eine Anonyme Variable verwenden.

Syntaxbeispiele für Referenzen

Benannte Variablen :

```
$scalar_ref = \$scalar      $array_ref = \@array
$hash_ref = \%hash          $func_ref = \&function
```

Anonyme Variablen :

```
$scalar_ref = \42                          $array_ref = [ "eins","zwei","drei" ]
$hash_ref = { "key1" => "val1",...}        $func_ref = sub { print "bla" }
```

Zugriff auf den Speicherinhalt

Syntaxbeispiele zum Dereferenzieren von Referenzen :

Anonyme Hashes

```
%hash = %$hash_ref                         %hash = %{$hash_ref}
$my_hash_value1 = ${$hash_ref}{'key1'}     $my_hash_value1 = $hash_ref->{'key1'}
```

Essenz

Den Speicherinhalt eines anonymen Hashes kann man mit

```
$my_hash_value1 = $hash_ref->{'key1'}
```

auslesen.

4 Unterprogramme - Libraries - Module

4.1 Subroutinen

Was sind Subroutinen ?

Subroutinen sind wiederverwendbare Codeblöcke. Es gibt perleigene Funktionen (siehe *man perlfunc*) und Selbstdefinierte. Die Begriffe *Subroutine, Funktion ,Prozedur* können synonym verwendet werden.

Eigenschaften

- Subroutinen können überall im Programm definiert werden.
  Zur besseren Übersicht werden sie häufig am Ende des Quelltextes definiert.
- Subroutinen akzeptieren Parameter, in Perl ist die Anzahl variabel.
- Subroutinen können einen oder mehrere Werte zurückliefern.
- Subroutinen können rekursiv aufgerufen werden.

Syntax

```
sub name_der_funktion {
 $parameter1 = shift;
 Anweisungen...
 return $rueck;
}
```

Der Aufruf einer Funktion erfolgt alternativ mit

- name_der funktion;
- name_der_funktion();
- &name_der_funktion()

Wobei in () die Parameter eingegeben werden können : *name_der_funktion($var,$str)*. Die Parameter werden in der Subroutine sukzessive mit **shift** eingelesen.

Veränderung der Aufrufparameter : my

Mit der Parameterübernahme in der Form *$parameter1 = shift* und anschließender Veränderung von *$parameter* wird auch der Wert des Parameters aus dem Funktionsaufruf geändert. (*Call by reference*).
Mit der Parameterübernahme in der Form ***my*** *$parameter1 = shift* wird eine Kopie des Übergabeparameters angelegt. Eine Veränderung der Kopie ändert das Original nicht. (*Call by value*).

Rückgabewerte mit return

Mit **return $rueck** kann die Funktion *sofort* verlassen werden, **return** kann dabei an beliebiger Stelle im Funktionsblock { ... } stehen. Mit $rueck wird dann der Rückgabewert festgelegt.

Globale und Lokale Variablen

Variablen sind üblicherweise überall gültig (Hauptprogramm/Subroutinen). Sie heißen dann globale Variablen Werden in einer Funktion Variablen verwendet, die nur dort verwendet werden, dann werden sie mit my $var; deklariert. Sie heißen dann lokale Variablen, sie sind nur in dem jeweiligen Anweisungsblock gültig, ihr Speicherplatz wird mit Verlassen des Blockes freigegeben (sie "sterben"). Dadurch wird vermieden, daß Variablen im Hauptprogramm versehentlich überschrieben werden (z.B. Laufvariablen !)

Funktionen auslagern

Funktionen können in Perl ausgelagert werden. D.h., sie werden in einer seperaten Datei gespeichert. In ein Perl-Script werden sie eingebunden durch `require datei_name.extension`

Voraussetzung dafür ist, daß sich die Datei in Verzeichnissen befindet, die in einem Array namens **@INC** enthalten sind. In @INC befindet sich insbesondere das aktuelle Verzeichnis sowie unter Linux z.B. /usr/lib/perl5/5.6.0 u.a.m. @INC ist die Suchpfadliste für Funktionsbibliotheken (zunächst einmal).

@INC ist erweiterbar.

Die Funktionen können im Script verwendet werden, als wären sie direkt im Script definiert worden. Der Speicherbereich des Scripts[1] ohne *require* wurde expandiert um den Speicherbedarf der Funktionen. Der Vorteil liegt also nicht in effizienterer Speicherverwaltung, sondern lediglich in der Übersichtlichkeit des Quelltextes und der Möglichkeit, **Funktionsbibliotheken in mehreren Scripten gleichzeitig zu nutzen**

Zwischenbemerkung

Nicht nur Funktionen können in Funktionsbibliotheken abgelegt werden. Auch Variablen, global wie lokal (mit *my $var*) können dort definiert werden. Lokal definierte Variablen können nicht über *require* in ein Script importiert werden. Funktionen können in Perl nicht lokal definiert werden, im Gegensatz zu Java (*private*) . Allerdings kann in der Funktionsbibliothek auf diese Variablen Bezug genommen werden.

Wie wäre es nun, wenn man nicht den gesamten Speicherbereich der Funktionsbibliothek in das Script kopiert (denn nichts anderes passiert durch *require*), sondern lediglich **die Speicheradressen der Funktionen und Variablen dem Script übergibt** ? Vordergründig gibt man die Mehrfachnutzung wieder preis, da dann nicht mehrere Scripten *zugleich* die Lib nutzen können. Jedoch kann man es dem Script überlassen, welche Funktionen und damit Speicherbereiche in das Script zur Laufzeit hineinkopiert werden.

Welche Funktionalitäten müssen dafür zur Verfügung gestellt werden ?

1. Die Funktionsbibliothek stellt eine Liste der Adressen ihrer Funktionen und Variablen zur Verfügung.
2. Das Script nimmt die Liste entgegen und allokiert sich bei Bedarf für die mit den Adressen verknüpften Elemente der Lib Speicherbedarf und kopiert die Struktur dort hinein.

Zu 1 : Welche Speicherstruktur kommt in Frage ? Geleistet werden muß die unmittelbare Zuordnung einer Speicheradresse zu einem Funktionsnamen (allgemeiner : zu einem Elementnamen). Das leistet ein Hash. Dieser Hash braucht keinen Namen, wenn ich der Lib einen Namen gebe, über die ich sie ansprechen kann. Also ein anonymer Hash, der über eine Funktion (bless) mit dem Namen der Lib verknüpft wird. Die Namensgebung (Objektbezeichnung) muß daher innerhalb der erweiterten Lib erfolgen und erfolgt durch **package name** in der Lib, die jetzt als Modul bezeichnet wird. (Perl : Modul, Java : Klasse, zuweilen synonym verwendet)

Nun muß dieser dem packagename verknüpfte anonyme Hash noch einem Script zur Verfügung gestellt werden. Als Schnittstelle dient eine Funktion (new)

Daraus entsteht folgender Aufbau eines Perl-Moduls :

```
package my_modul
sub new {
          bless {};
}
sub func1 { ... }
sub func2 { ... }
```

[1] : Das ist natürlich nicht exakt gesprochen, denn Speicherbereich wird erst zur Laufzeit allokiert, da Perl eine interpretierte Sprache ist.

4.2 Packages

Packages und Funktionsbibliotheken

Subroutinen werden zur wiederholten Verwendung *innerhalb* eines Programms definiert. Zweckmäßig ist die Definition von Subs in eigenen Dateien (Bibliotheken), die bei Bedarf in ein Programm eingebunden werden können. Diese Bibliotheken können dann von mehreren Programmen verwendet werden.

Name einer Datei, die ein Package enthält

z.B. **pack1.pm** (**.pm** erforderlich)

Ort der Datei im Verzeichnis

@INC oder das gleiche Verzeichnis wie das aufrufende Programm

Packagename

wie Dateiname ohne Extension (hier : *pack1*)

Syntax

Im pack1.pm wird in der ersten Zeile das Package definiert : **package pack1;** Die letzte Zeile eines Packages enthät einen wahren Ausdruck : **1;**

Die im Package verwendeten Funktionsnamen müssen bekannt gemacht (exportiert) werden :

```
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(name_of_fkt1 name_of_fkt2 ...);
```

Danach erfolgt die Definition der Subroutinen Im Programm kann das Package mit **use pack1;** eingebunden werden. Die Funktionsnamen können dann verwendet werden, als wenn die Funktionen im Script selbst definiert wurden.

Beispiel

Datei /home/willy/bin/script.pl	Datei /home/willy/bin/give_uc_name.pm

`#!/usr/bin/perl`
 	
`use give_uc_name;`
 	
`$myname = "Willy Wunz";`
`$ucname = do_it_now($myname);`
 	
`print $ucname,"\n";`
	`package give_uc_name;`
	`require Exporter;`
	`@ISA = qw(Exporter);`
	`@EXPORT = qw(do_it_now);`
	
	`sub do_it_now {`
	`  my $name = shift;`
	`  return uc($name);`
	`}`
	`1;`

4.3 Module

Pragmatischer Ansatz

Wir widmen uns folgenden Fragen:

- Was bieten Module ?
- Wo liegen die Module ?
- Wie werden Module angewendet ?
- Wo gibt es Informationen über Module ?
- Wo gibt es neue Module ?

Was bieten Module ?

Module sind fertige Lösungen von speziellen Programmieraufgaben. Sie bieten Funktionen (**Methoden** genannt), und Variablen (**Eigenschaften** genannt) die verwendet werden können, ohne daß die Einzelheiten der Lösung bekannt sein müssen.

Beispiel : Das Modul Cwd bietet u.a. die Methode cwd, die das aktuelle Verzeichnis liefert. Das geschieht unabhängig vom jeweiligen Betriebssystem, auf dem das Programm abläuft.

Wo liegen die Module ?

Perl bringt standardmäßig eine Menge von Modulen mit. Sie werden als **.pm**-Dateien in einem Verzeichnis abgelegt. Die Verzeichnisse, in denen der Perlinterpreter nach Modulen sucht, sind in der Perlvariable @INC gespeichert. Beispiel : SuSE Linux 7.3, Perlversion 5.6.1 , *perl -e 'print join "\n",@INC'* liefert

```
/usr/lib/perl5/5.6.1/i586-linux
/usr/lib/perl5/5.6.1
/usr/lib/perl5/site_perl/5.6.1/i586-linux
/usr/lib/perl5/site_perl/5.6.1
/usr/lib/perl5/site_perl
```

.Zu beachten ist das "**.**"-Verzeichnis. Das ist das Verzeichnis, in dem das Programm abläuft. (Genauer : das aktuelle DVZ zum Zeitpunkt der Anwendung von **use** *Modulname*)
Unterhalb der DVZ' aus @INC gibt es weitere Verzeichnisse, wie z.B. Net unter .../site_perl/5.6.1/Net/.

Wie werden Module angewendet ?

1. Einbinden in Programme

Module werden in ein Programm mit **use** *Modulname;* eingebunden. Dann hat man Zugriff auf die **Methoden** dieses Moduls.

Liegt das Modul in einem Unterverzeichnis von @INC, so wird das Modul mit **use** *Verzeichnisname***::***Modulname;* eingebunden.

Beispiel : **use** Net**::**FTP;

Mit der Funktion **new** wird eine Kopie (Instanz) des Moduls gebildet.

Beispiel : $ftp_instanz = **new** Net::FTP("host.domain.com")

Wie werden Module angewendet ?

2. Verwenden von Funktionen

Module stellen Methoden und Eigenschaften zur Verfügung. Diese Methoden werden nicht einfach durch Aufruf des Methodennamens ausgeführt. Eine Funktion des Moduls ruft man mit der Instanz auf

Beispiel : $ftp_instanz**->**login("$username","password");

Wo gibt es Informationen über Module ?

In den Man-pages, die zu den Modulen geliefert werden. Dabei werden die Man-pages genauso aufgerufen, wie die Module in den Scripten eingebunden werden.

Beipiel : man Cwd man Net::FTP man File::Copy

Wo gibt es neue Module ?

Auf dem CPAN (Comprehensive Perl Archive Network), ein FTP-Archiv, das weltweit gespiegelt wird. Einstieg über www.perl.com Mirror-sites worldwide Deutscher Mirror auf ftp.gmd.de Eine Newsgroup **comp.lang.perl.modules** kümmert sich um Module eine Newsgroup **comp.lang.perl.misc** kümmert sich um Perl allgemein.

Etwas Theorie

Zunächst wird für das Modul mit seinen Methoden und Eigenschaften ein eigener Speicherplatz reserviert. Das geschieht, indem man eine Kopie aller Methoden/Eigenschaften aus dem Modul erstellt und diese Kopie benennt. Beispiel : $ftp = new Net::FTP; eine Kopie wird mit der Methode **new** erstellt, die Variable $ftp ist dann ein Bezeichner, der auf diese Kopie verweist. In der Terminologie der OOP ist das Modul ein Objekt, die Kopie eine Instanz dieses Objekts und die skalare Variable, der die Instanz zugewiesen wird, eine Referenz auf diese

5 Dateiverarbeitung

5.1 Dateizugriff

Mit **open(FH,"<*dateiname*")** wird eine Datei zum Lesen geöffnet.
Mit **open(FH,">*dateiname*")** wird eine Datei zum Schreiben geöffnet.
Mit **open(FH,">>*dateiname*")** wird eine Datei zum Anhängen geöffnet.
Mit dem sogenannte *Filehandle* **FH** kann die Datei angesprochen werden kann.
Mit **close FH** wird die Datei wieder geschlossen.

Verzeichniszugriff

Mit **opendir(DH,"*verzeichnisname*")** wird ein Verzeichnis göffnet.
Mit **readdir(DH)** werden die Dateinamen des DVZ in eine Liste gelesen (incl. Unter-DVZ)

Mit **closedir DH** wird das Verzeichnis wieder geschlossen. Dateitestoperatoren *

-r (-R)	Datei ist durch effektive (reale) UID/GID lesbar
-w (-W)	Datei ist durch effektive (reale) UID/GID schreibbar
-x (-X)	Datei ist durch effektive (reale) UID/GID ausführbar
-o (-o)	Datei gehört der effektiven (realen) UID/GID
-f -d -l -p -S	Datei ist reguläre Datei, Verzeichnis, Sym-Link, Pipe, Socket
-b -c -t	Gerätedatei ist ein *block special file*, *character special file*, ein tty
-e -z -s	Datei existiert, hat Länge Null Byte, hat Länge in Byte
-u -g -k	Datei hat setuid-Bit,setgid-Bit, Sticky-Bit
-T -B	Datei ist Textdatei, Binärdatei
-M -A -C	Modifikationszeit, Zugriffszeit, Inode-Veränderungszeit in Tagen seit Programmstart

Die stat()-Funktion *

0	dev	Gerätenummer des Dateisystems
1	ino	Inode-Nummer innerhalb des Dateisystems
2	mode	Modus: Zugriffsrechte und Dateityp
3	nlink	Anzahl Hardlinks
4	uid	UID des Besitzers
5	gid	GID des Besitzers
6	rdev	Systemweite Raw-Gerätenummer des Dateisystems
7	size	Logische Größe der Datei in Byte
8	atime	Letzte Zugriffszeit in Sekunden seit der Epoche
9	mtime	Letzte Modifikationszeit in Sekunden seit der Epoche
10	ctime	Letze Inode-Ã„nderung in Sekunden seit der Epoche
11	blksize	Ideale Blockgröße für Dateisystem-Ein/Ausgabe
12	blocks	Anzahl aktuell zugewiesener Blöcke

- : Quelle : Perl Einführung,Anwendung, Referenz. Farid Hajji, Addison Wesley 2. Auflage

5.2 Reguläre Ausdrücke 1

Der Operator =~

Gegeben sei ein String $str und eine Pattern-Matching-Ausdruck z.B. /A/ . Der Ausdruck $str =~ m/^A/ kann dann als Bedingung verwendet werden. Fragestellung : kommt die Zeichenkette "A" (*Pattern*) in $str vor ?. Dabei kann der Pattern-Matching-Ausdruck nach den Regeln der Regulären Ausdrücke gestaltet werden. Siehe auch **man perlop, man perlre**

Suchen in Strings : m

```
$var =~ m/PATTERN/flag
```

Anwendung : Erzeugen von Bedingungen

Bei Verwendung von $_ kann geschrieben werden : */PATTERN/*

Anstelle von "/" können auch andere Zeichen verwendet werden, z.B. ";" oder "'".

Der Suchalgorithmus kann mit Flags modifiziert werden

Suchposition nicht zurücksetzen bei Fehler (bei g)	
g	Globale Suche, d.h. finde alle Vorkommen
m	Case-insensitive
i	Strings können aus mehreren Zeilen bestehen
o	Suchmuster nur einmal kompilieren
s	Strings nur als eine einzige Zeile betrachten
x	Erweiterte Syntax aktivieren

Suchen und ersetzen : s

$var =~ **s**/*PATTERN*/*replace-string*/*flag*

Beispiel :

```
$str = "Hans,Harries,Hamburg";
$str =~ s/,/:/g;
```

=> $str = "Hans:Harries:Hamburg";

Ersetzen von Zeichenketten : tr bzw. y

$str =~ tr/*VONLISTE*/*ZULISTE*/*yank-flags*
Beispiel : $str = "abcdef"; $str = tr/abc/xyz/; => Str = "xyzdef"

c	Komplement von *VONLISTE*
d	Entfernt nichtersetzte Zeichen
s	Aus der Ersetzung entstandene Duplikate entfernen
U	Konvertiert von/nach UTF-8
C	Konvertiert von/nach 8-Bit-Zeichen (Oktet)

5.3 Reguläre Ausdrücke 2

```
$str = "From:michael.kalinka\@gmx.de";
```

Einfaches Matching : /*PATTERN*/

Prüfung, ob ein String in einem zweiten vorkommt.

Beispiel : `if ($str =~ /From/) { print "From in $str" }`

Matching am Anfang : /^*PATTERN*/

Prüfung, ob ein String **am Anfang** eines zweiten vorkommt.

Beispiel : `if ($str =~ /^From/) { print "From am Anfang $str" }`

Matching am Ende : /*PATTERN*$/

Prüfung, ob ein String **am Anfang** eines zweiten vorkommt.

Beispiel : `if ($str =~ /de$/) { print "From am Anfang $str" }`

Matching : ein beliebiges Zeichen mit ".

Prüfung, ob ein Zeichen in einem zweiten vorkommt.

Beispiel : `if ($str =~ /./) { print 'Immer wahr, wenn $str != "" ' }`
Beispiel : `if ($str =~ /mi./) { print "Gibt es" }`
Beispiel : `if ($str =~ /de./) {} else { print "nach de kommt kein Zeichen" }`

Wiederholungen : Keinmal, Einmal, Mehrfach

***** bedeutet **keinmal, einmal oder mehrfach**.

Beispiel : `if ($str =~ /de.*/) { print "Gibt es" }`
Grund : in $str gibt es **de** mit *keinmal* ein beliebiges Zeichen.

+ bedeutet **einmal oder mehrfach**, also *mindestens einmal*

Beispiel : `if ($str =~ /ka+/) { print "Gibt es" }`
Grund : in $str gibt es **ka** mindestens einmal.
Beispiel : `if ($str =~ /de.+/) {} else { print "Gibt es nicht" }`
Grund : in $str gibt es **de** nicht mit mindestens einem beliebigen Zeichen.

Zeichenmengen : Whitespace (Blank), Wort, Ziffer, Wortgrenze

\s \S : Whitespace - kein Whitespace
\d \D : Ziffer - keine Ziffer
\w \W : Wort - kein Wort
\b \B : Wortgrenze - keine Wortgrenze

Beispiel : `if ($str =~ /\s/) {} else { print "Kein Whitespace" }`
Beispiel : `if ($str =~ /\w/) { print "Ein Wort gefunden" }`
Beispiel : `if ($str =~ /\D/) { print "Keine Ziffer" }`
Beispiel : `if ($str =~ /\b/) { print "Wortgrenze gefunden" }`

5.4 Reguläre Ausdrücke 3

$str = "From:michael.kalinka\@gmx.de";

Bereiche einzelner Zeichen mit [ ]

Mit [abc] kann geprüft werden, ob "a" *oder* "b" *oder* "c" vorkommt.

Beispiel : `if (`**`$str =~ /[yZa]/`**`) { print "y|Z|a in $str" }`
Beispiel : `if (`**`$str =~ /[yZ]/`**`) {} else { print "weder y noch Z in $str" }`

Bemerkung : Das gleich gilt für Ziffern.

Bereichsangaben : **[a-e]** steht für **[abcde] [0-9]** steht für alle Ziffern **[0123456789]** Beispiel : if (**$str =~ /[0-9]/**) {} else { print "keine Ziffer in $str" } **[a-zA-Z0-9]** steht für alle alphanumerischen Zeichen

Negierung in Bereichen : **[^abc]** steht für nicht "a" und nicht "b" und nicht "c"

Beispiel : if (**$str =~ /[^Z]/**) { print "Es gibt ein Zeichen, das kein Z ist" }

Beispiel : if (**$str =~ /[^Za]/**) { print "Es gibt ein Zeichen, das kein Z *oder* kein a ist" }

Beispiel : if (**$str =~ /[^a]/**) { print "Es gibt ein Zeichen, das kein a ist" }

Extraktion mit ( )

Durch Klammerung können bestimmte Matches extrahiert werden.

Beispiel : $wort = ($str =~ /^**(**\w**)**/);

==> $wort = "From"

Beispiel : ($keyword,$text) = ($str =~ /^**(**\w**)**:**(**.***)**/);

==> $keyword = "From" , $text = "michael.kalinka@gmx.de"

Beispiel : $str =~ /^**(**.***)**:**(**.***)**/;

==> $1 = "From" , $2 = "michael.kalinka@gmx.de"

ACHTUNG : Beispiel : $str =~ /^**(**\w**)**:**(**.***)**/; ==> kein Matching (Grund unbekannt, Hinweise an obige mail) Beispiel : $str =~ /**(**\w**)**:**(**.***)**/; ==> $1 = "m" (Grund unbekannt, Hinweise an

5.5 Escapesequenzen

Formatierung von ASCII-Text

Sonderzeichen	\"	Das Anführungszeichen selbst
	\\	Der Backslash selbst
Steuerzeichen	\n	Newline : Sprung in die nächste Zeile
	\r	Cursor return : "Wagenrücklauf"
	\t	Tabulator
	\f	Form feed : Seitenvorschub
	\v	Vertikaler Tabulator

ANSI Steuerzeichen : Escape-Sequenzen

Cursor	**\033[**x;y**H**	C. auf x. Spalte, y. Zeile
	\033[x;y**f**	C. auf x. Spalte, y. Zeile
	\033[n**A**	C. n Zeilen nach oben
	\033[n**B**	C. n Zeilen nach unten
	\033[n**C**	C. n Spalten nach rechts
	\033[n**D**	C. n Spalten nach links
	\033[s	Speichert C.-Position
	\033[u	Setzt C. auf gespeicherte C.-Pos.
Bildschirm	**\033[**2J	Löschen des Bildschirms (clear)
	\033[K	Löscht alle Zeichen rechts vom C.

Farben und Attribute

Attribute		Maske : **\033[**n**m**		Standard : **\033[m**	
Textattribute		Vordergrundfarben		Hintergrundfarben	
n	Wert	n	Wert	n	Wert
0	Attribute aus	30	schwarz	40	schwarz
1	Fettdruck	31	rot	41	rot
4	unterstreichen	32	grün	42	grün
5	blinken	33	gelb	43	gelb
7	invertierte Darstellung	34	blau	44	blau
8	unsichtbar	35	magentarot	45	magentarot
		36	cyanblau	46	cyanblau
		37	weiß	47	weiß
Syntax bei Mehrfachauswahl : **\033[**i**;**j**;**...**;**n**m**					