

Perl-Praxis
Kontrollstrukturen

Jörn Clausen
joern@TechFak.Uni-Bielefeld.DE

Übersicht

- Vergleichsoperatoren
- Bedingungen
- Wahrheitswerte
- bedingte und unbedingte Schleifen

Vergleichsoperatoren

- korrekten Operator wählen:

numerisch:	==	!=	<	>	<=	>=	<=>
lexikographisch:	eq	ne	lt	gt	le	ge	cmp

- Arrays sortieren:

```
@sorted = sort(@disney);
```

- Sortierkriterium angeben:

```
@bsorted = sort( {$b <=> $a} @primes);
```

- default: { \$a cmp \$b } (lexikographisch aufsteigend)

Aufgaben

- Was passiert, wenn Du das Array `@disney` numerisch und das Array `@primes` lexikographisch sortierst?

```
@disney = ("Mickey", "Donald", "Goofy", "Scrooge",  
          "Daisy", "Pluto", "Huey", "Dewey", "Louie");
```

```
@primes = (2,3,5,7,11,13,17,19,23,29);
```

- Sortierung der Arrays:

```
print join(" ", sort({$a cmp <=> $b} @disney)), "\n";  
print join(" ", sort({$a cmp $b} @primes)), "\n";
```

Ergebnisse:

```
Mickey, Donald, Goofy, Scrooge, Daisy, Pluto, Huey, Dewey, Louie  
11, 13, 17, 19, 2, 23, 29, 3, 5, 7
```

Die Strings haben den numerischen Wert "0", sind aus Sicht von `sort` also gleichwertig. Die Reihenfolge wird bei der Sortierung beibehalten.

Zufall

- Zufallszahl mit `rand` erzeugen:

```
$num = rand();           # im Interval [0,1[  
$lotto = int(rand(49))+1; # aus der Menge 1,2,...,49
```

- Zufallsgenerator mit `srand` initialisieren:

```
srand();
```

- wird bei der ersten Verwendung von `rand` aufgerufen
- fester *seed*:

```
srand(12345);
```

- zum Testen von Software
- Initialisierung aufgrund von Zeit, PID, usw. nicht sicher

Entscheidungen

- mehrere Möglichkeiten der Entscheidung:
 - ternärer Operator
 - if-then-else-Konstrukt
 - case/switch-artige Strukturen
 - *compound statements vs. modifiers*
 - *short-circuit operators*
- TIMTOWTDI at work

Ternärer Operator

- *Test ? Wahr-Ausdruck : Falsch-Ausdruck*

- Münzwurf:

```
$num = int(rand(2));  
$face = ($num == 1) ? "Kopf" : "Zahl";  
print "$face gewinnt\n";
```

- echter Operator:

```
$year = ($digits>70 ? 1900 : 2000) + $digits;
```

if-then-else

- klassische Verzweigung:

```
if ($num == 1) {  
    print "Kopf gewinnt\n";  
    $head++;  
} else {  
    print "Zahl gewinnt\n";  
    $tail++;  
}
```

- geschweifte Klammern { . . . } kennzeichnen *Block*
- müssen verwendet werden, auch bei einzeiligem Block
- else-Zweig optional

Case/Switch

- kein echtes case/switch in Perl

- Abfolge von Bedingungen:

```
if ($nephew eq 'Huey') {  
    print "nephew no. 1\n";  
} elsif ($nephew eq 'Dewey') {  
    print "nephew no. 2\n";  
} elsif ($nephew eq 'Louie') {  
    print "nephew no. 3\n";  
} else {  
    print "no nephew of Donald\n";  
}
```

- beliebige Bedingungen, Vorsicht vor „logischen Lücken“
- kein break o.ä.

Aufgaben

- Ermittle eine ganze Zufallszahl im Bereich von 1 bis 1000. Gib aus, durch welche der Zahlen 2, 3, 4, 5, 7 oder 10 sie teilbar ist. Die Ausgabe soll folgende Form haben:

Die Zahl 42 ist durch 2 und 3 und 7 teilbar.

Tip 1: Sammle die passenden Teiler in einem Array.

Tip 2: Verwende `join` zur Ausgabe der Teiler.

- Das Problem läßt sich auf unterschiedliche Weisen lösen. Man kann Abfragen kombinieren, da die Zahl z.B. nur durch 10 teilbar ist, wenn sie auch durch 5 teilbar ist.

Variante 1:

```
if ($num % 2 == 0) {
  push(@teiler, 2);
}
if ($num % 4 == 0) {
  push(@teiler, 4);
}
if ($num % 3 == 0) {
  push(@teiler, 3);
}
if ($num % 5 == 0) {
  push(@teiler, 5);
}
if ($num % 10 == 0) {
  push(@teiler, 10);
}
if ($num % 7 == 0) {
  push(@teiler, 7);
}
}
```

Variante 2:

```
if ($num % 4 == 0) {
  push(@teiler, 2, 4);
} elsif ($num % 2 == 0) {
  push(@teiler, 5, 10);
} elsif ($num % 5 == 0) {
  push(@teiler, 5);
}
if ($num % 3 == 0) {
  push(@teiler, 3);
}
if ($num % 7 == 0) {
  push(@teiler, 7);
}
}
```

Was ist Wahrheit?

- ... oder was Perl dafür hält.
- kein Bool'scher Datentyp
- Welche Werte werden zugewiesen?

```
$eq1 = (5 == 5);  
$eq2 = ('X' eq 'U');
```

- Perls Sicht der Dinge:

falsch	0, '0'	(Zahl/String „Null“)
	''	(leerer String)
	()	(leere Liste/Array/Hash)
	undef	(undefinierter Wert)
wahr	alles andere	

```
$eq1 == 1  
$eq2 == '' # leerer String
```

logische Operatoren

- „Und“: &&
- „Oder“: ||
- Negation: !
- mit geringster Präzedenz: and, or, not, xor
- im Zweifelsfall klammern:

```
if ( ($login eq 'juser') && ($passwd eq 'geheim') ) { ... }
```

- Bit-Operatoren: & | ^ ~ >> <<

```
$debug = $options & ($VERBOSE | $LOG);
```

Wahrheitstests

- typisches Idiom:

```
if (!$infile) { die "no input file specified\n"; }
```

- Unterschied zwischen „nicht wahr“ und „nicht definiert“:

```
# $a = 0;
if (!$a) { print "nicht wahr\n"; }
if (!defined($a)) { print "nicht definiert\n"; }
```

- polymorphe Unwahrheit:

```
$res = ('X' eq 'U');
if ($res == 0) { print "gleich Null\n"; }
if ($res eq '') { print "leerer String\n"; }
if (!$res) { print "unwahr\n"; }
if (!defined($res)) { print "undefiniert\n"; }
```

Variationen über Bedingungen

- Gegenteil von `if`: `unless`

```
unless ($name eq 'Joe User') {  
    print "I don't know you\n";  
}
```

- bisher: *compound statements*, jetzt: *modifiers*

```
print "I don't know you\n" unless $name eq 'Joe User';  
$color='red' if $choice == 3;
```

- Code wird „lesbarer“
- Aufmerksamkeit auf Aktion, nicht auf Bedingung

short-circuit operators

- logische Operatoren und *lazy evaluation*
- logisches „Und“: `&&` und `and`
- logisches „Oder“: `||` und `or`
- Datei öffnen oder Fehlermeldung:

```
open(...) || die "can't open file\n";
```

- Datei öffnen und Erfolgsmeldung:

```
open(...) && print "file opened\n";
```

bedingte Schleifen

- Schleife, solange Bedingung wahr ist:

```
$num = 1;
while ($num <= 10) {
    print "$num\n";
    $num++;
}
```

- Zuweisung evaluiert zum zugewiesenen Wert:

```
@nephews = ("Huey", "Dewey", "Louie");
while ($name = shift(@nephews)) {
    print "Hi $name\n";
}
```


bedingte Schleifen, cont.

- Gegenteil von while:

```
until ($num > 10) { ... }
```

- Bedingung nach einem Schleifendurchlauf:

```
@names = ("Brian", "Judith", "Jehova", "Loretta");  
do {  
    $name = shift(@names);  
    print "Hi $name\n";  
} until ($name eq 'Jehova');  
print "SHE SAID JEHOVA!\n";
```

- ebenfalls *modifier*
- funktioniert auch mit while, if und unless

Aufgaben

- Erzeuge Zufallszahlen im Bereich von 1 bis 1000 und gib sie aus. Brich das Programm ab, wenn die Zahl 666 generiert wird.
- Erweitere das Programm so, daß maximal 100 Zahlen erzeugt werden, auch wenn die Zahl 666 noch nicht darunter war.

```
count = 0;
while (($num != 666) and ($count < 100)) {
    $num = int(rand(1000))+1;
    $count++;
    print "$count. Versuch: $num\n";
}
```

- Schleife, bis 666 gezogen wird, maximal 100 Durchläufe:

```
while ($num != 666) {
    $num = int(rand(1000))+1;
    print "$num\n";
}
```

- Schleife, bis 666 gezogen wird:

for-Schleifen

- klassische dreiteilige for-Schleife:

```
for ($i=0; $i<=10; $i++) {  
    print $i,"**2 = ",$i**2,"\n";  
}
```

- for (*Initialisierung ; Bedingung ; Modifikation*) { *Block* }

- mehrere Anweisungen durch , trennen:

```
for ($i=1,$j=2**20; $i<$j; $i+=1,$j/=2) {  
    print "$i, $j\n";  
}
```

- unendliche Schleife:

```
for (;;) { ... }
```

```
1, 1048576  
2, 524288  
3, 262144  
4, 131072  
5, 65536  
6, 32768  
7, 16384  
8, 8192  
9, 4096  
10, 2048  
11, 1024  
12, 512  
13, 256  
14, 128  
15, 64  
16, 32
```

unbedingte Schleifen

- Iteration über Liste/Array:

```
foreach $duck (@nephews) {  
    print "$duck is a nephew of Donald\n";  
}
```

- falls Array veränderbar ist (aus *lvalues* besteht):

```
@prices = (1..100); # prices in DEM  
foreach $price (@prices) {  
    $price = $price / 1.95583;  
}  
# prices now in EUR
```

- Schleifenvariable *alias* für Array-Element

loop control

- Steuerung des Schleifenablaufs
- Überspringen des aktuellen Schleifenblocks:

```
while ($num = shift(@nums)) {  
    next if $num == 0; # avoid division by zero  
    push(@recip, 1/$num);  
}
```

- Beenden der Schleife:

```
while ($name = shift(@names)) {  
    last if $name eq 'Jehova';  
    ...  
}
```

- Wiederholen des aktuellen Schleifenblocks: `redo`

Aufgaben

- Löse die letzte Aufgabe (maximal 100 Zufallszahlen, oder 666) erneut, diesmal mit Hilfe einer `for`- oder `foreach`-Schleife.
- Gib das kleine Einmaleins (alle Produkte von 1×1 bis 10×10) aus.
- Implementiere den Bubble-Sort-Algorithmus, der ein Array von beliebig vielen Zufallszahlen *in-place* sortiert.

```
for ($i=0; $i<$len; $i++) {
    print "have now: ", join(" ", @a), "\n";
    for ($j=$len-1; $j>=$i; $j--) {
        if ($a[$j] < $a[$j+1]) {
            ($a[$j], $a[$j+1]) = ($a[$j+1], $a[$j]);
        }
    }
}
```

- Bubblesort:

```
foreach $count (1..100) {
    $num = int(rand(1000))+1;
    print "$count. Versuch: $num\n";
    last if $num == 666;
}
```

- Schleife, bis 666 gezogen wird, maximal 100 Durchläufe: