

HowTo

Perl in einer Stunde

Oliver Wilkening

Inhaltsverzeichnis

1	Einleitung	4
2	Hello World	4
3	(Skalare) Variablen	5
4	Bedingungen	6
5	Arrays, Listen	7
5.1	Hash(array)	8
6	Dateiverarbeitung	9
7	Unterprogramme	11
8	Here-Document	12

Abbildungsverzeichnis

1	Ausgabe hello.pl	4
2	Ausgabe var.pl	5
3	Ausgabe bed.pl	6
4	Ausgabe arr.pl	7
5	Ausgabe hash.pl	9
6	Ausgabe data.pl	10

1 Einleitung

Dieses Tutorial soll dazu dienen, einen Einstieg in die Programmiersprache Perl zu finden. Voraussetzung zum Verständnis ist das grundlegende Wissen um Programmstrukturen, also die Erfahrung in mindestens einer Programmiersprache. Alle Beispiele sind unter Linux getestet und sollten auch unter Betriebssystemen funktionieren, die einen Perl-Interpreter bereitstellen.

Die Geschichte von Perl o.ä. soll hier nicht erläutert werden. Dazu sei auf den entsprechenden Wikipedia-Artikel¹ verwiesen.

2 Hello World

Das obligatorische „Hello World“ erläutert wohl am Besten die grundsätzlichen Strukturen einer Programmiersprache. Wir generieren die Datei `hello.pl` mit folgendem Inhalt:

```
#!/usr/bin/perl
print "Hello World\n";
```

Und machen die Datei ausführbar mit `chmod 0755 hello.pl`. Danach führen wir die Datei aus: `./hello.pl`. Die Ausgabe sollte sich dann wie folgt darstellen:

A screenshot of a terminal window titled 'mc - /'. The prompt is 'oliver@server:~/webperl>'. The user enters './hello.pl'. The output is 'Hello World'. The prompt returns to 'oliver@server:~/webperl>'.

```
mc - /
oliver@server:~/webperl> ./hello.pl
Hello World
oliver@server:~/webperl>
```

Abbildung 1: Ausgabe hello.pl

¹<http://de.wikipedia.org/wiki/PERL>

Zwei Dinge sollten auffallen:

- Zeile 1 enthält die so genannte *Shebang*: `#!/usr/bin/perl`. Die *Shebang* gibt den Ort zum Perl-Interpreter an (hier `/usr/bin/perl`). **Der Gartenzaun # läutet ansonsten im Perl-Code Kommentare ein!**
- Eine Ausgabe auf der Standardausgabe erfolgt mit dem `Print`-Befehl, Befehlszeilen werden mit einem Semikolon abgeschlossen. Das `\n` bewirkt einen einfachen Zeilenumbruch.

Damit hätten wir die Grundlagen gelegt und können jetzt richtig loslegen!

3 (Skalare) Variablen

Skalare Variablen werden in Perl ähnlich wie z.B. in PHP gebildet:

`$name`. Skalare Variablen sind immer typenlos!

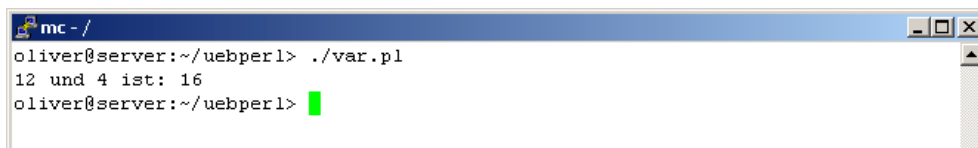
Hervorzuheben ist hier Zeile 2 „`use strict;`“. Mit `strict` wird eine Variable mit dem Schlüsselwort `my` initialisiert. Das bedeutet, dass ohne diese explizite Initialisierung Kompilierfehler auftauchen werden, dadurch Flüchtigkeitsfehler durch falsche Benennung minimiert werden². Als Beispiel legen wir jetzt die Datei `var.pl` mit folgendem Inhalt an:

```
#!/usr/bin/perl
use strict;

my $zahl1 = 12;
my $zahl2 = "4";
my $summe = $zahl1 + $zahl2;

print $zahl1." und " . $zahl2." ist: " . $summe . "\n";
```

Die Datei machen wir wieder ausführbar und sollten nach der Ausführung Folgendes zu sehen bekommen:



```
mc - /
oliver@server:~/uebperl1> ./var.pl
12 und 4 ist: 16
oliver@server:~/uebperl1>
```

Abbildung 2: Ausgabe var.pl

Die Variable `$zahl2` wurde hier wie eine String-Variablen belegt. Dem richtigen Rechenresultat tut das keinen Abbruch. `$zahl2` wird automatisch in den richtigen Zahlentyp konvertiert. Weiterhin sieht man, dass der Punkt zuständig für die

²<http://wiki.perl-community.de/bin/view/Wissensbasis/UseStrict>

Zusammensetzung von Zeichenketten ist.

4 Bedingungen

If, else, elsif sollten aus anderen Programmiersprachen bekannt sein und funktionieren in Perl genau so. Als kurzes Beispiel dient hier die Datei bed.pl:

```
#!/usr/bin/perl
use strict;

my $zahl1 = 12;

if ($zahl1 > 10)
{
    print $zahl1." ist groesser als 10\n";
}
else
{
    print $zahl1." ist kleiner als 10\n";
}
```

Die Ausgabe sieht hier wie folgt aus:



Abbildung 3: Ausgabe bed.pl

Bei der Interpretation vorhandener Perl-Programme werden die logischen Operatoren oft auch in der althergebrachten Fortran-Schreibweise vorkommen:

- größer → gt
- kleiner → lt
- größer oder gleich → ge
- kleiner oder gleich → le
- gleich → eq
- ungleich → ne

Eine Besonderheit bei Perl ist `unless` und ist eine Schreibweise für `if (!(<Bedingung>))`. Stellt also sozusagen das „Nicht-Wahr“ dar.

5 Arrays, Listen

Für Arrays/Listen bedient sich Perl bei dem @-Zeichen. Die Zuweisung sieht also wie folgt beispielhaft aus:

```
@blumen = ("Rosen", "Tulpen", "Begonien");
```

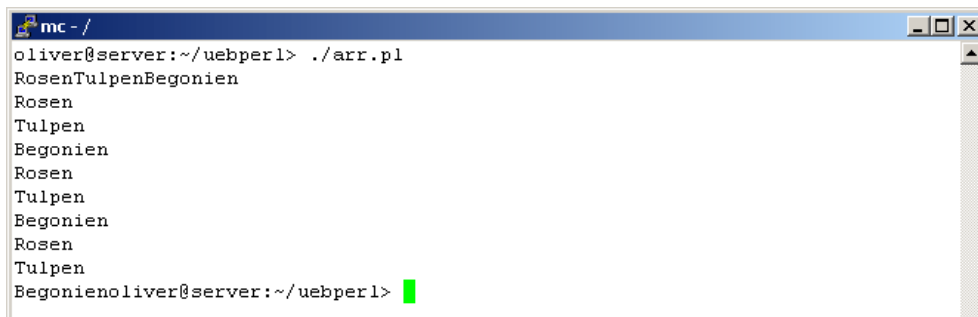
Die Verarbeitung einer solchen Liste geschieht mit einer for- oder einer foreach-Schleife. Zur beispielhaften Darstellungsweise, legen wir dazu die Datei arr.pl mit folgendem Inhalt an:

```
#!/usr/bin/perl
use strict;

my @blumen = ("Rosen", "Tulpen", "Begonien");
print @blumen;

for (my $i=0; $i<@blumen; $i++)
{
    print "\n" . $blumen[$i];
}
foreach my $value (@blumen)
{
    print "\n" . $value;
}
foreach (@blumen)
{
    print "\n" . $_;
}
```

Die Ausgabe sieht wie folgt aus:



```
mc - /
oliver@server:~/uebperl1> ./arr.pl
RosenTulpenBegonien
Rosen
Tulpen
Begonien
Rosen
Tulpen
Begonien
Rosen
Tulpen
Begonienoliver@server:~/uebperl1>
```

Abbildung 4: Ausgabe arr.pl

Wir sehen, dass `print @blumen;` die Liste einfach ausgibt. Um eine Formatierung hineinzubekommen, benötigen wir eine Schleife.

Bei der For-Schleife sehen wir, dass die Anzahl der vorhandenen Elemente nicht erst durch eine zusätzliche Funktion abgefragt werden muss. `@blumen` liefert uns die Anzahl zurück. Der Perl-Interpreter ist also so „schlau“, dass er weiß, was

wir vorhaben. Bei der Print-Anweisung (`@blumen`) gibt er die Liste aus, in der For-Schleife liefert er die Anzahl der Elemente zurück mit `@blumen`.

In der ersten foreach-Schleife, weisen wir der Variable `$value` bei jedem Durchlauf den entsprechenden Wert aus der Liste zu. Wenn wir nicht auf die Weiterverarbeitung des Wertes angewiesen sind, also nicht unbedingt einen Variablenamen benötigen, können wir auf die kurze und knappe Schreibweise der zweiten foreach-Schleife zurückgreifen:

Jeder Listenwert steht im entsprechenden Durchlauf hier automatisch in der Variable `$_`.

5.1 Hash(array)

Eine Hasharray bietet die Möglichkeit, die Listenwerte zu kennzeichnen, sie zu indexieren. Man kann sich das wie bei einem Telefonbuch vorstellen, wo einem Namen eine Nummer zugeordnet ist. Ein Hasharray wird mit dem Prozentzeichen wie folgt gebildet:

```
%telefon = ("Mr. Foo", "12345", "Ms. Bar", "345345"); alternativ
```

```
%telefon = ("Mr. Foo"=>"12345", "Ms. Bar"=>"345345");
```

Der Zugriff kann im Rahmen einer Schleife erfolgen aber auch direkt:

```
print $telefon{"Mr. Foo"};
```

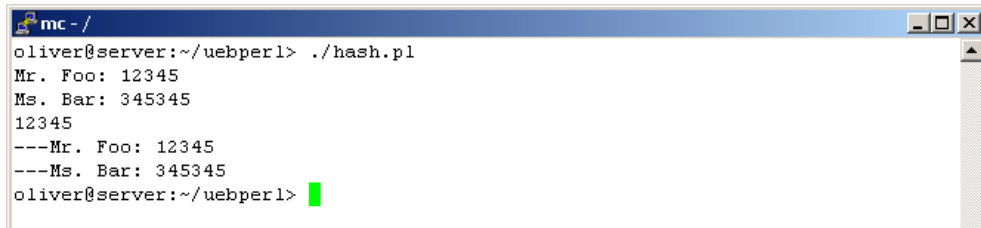
Zum testen legen wir die Datei `hash.pl` mit folgendem Inhalt an:

```
#!/usr/bin/perl
use strict;

my %telefon = ("Mr. Foo", "12345", "Ms. Bar", "345345");
# Abarbeitung in einer While-Schleife
my $key;
my $value;
while ( ($key, $value) = each(%telefon) )
{
    print $key . " : " . $value . "\n";
}

# Alternative
%telefon = ("Mr. Foo"=>"12345", "Ms. Bar"=>"345345");
# Direkter Zugriff -> $telefon
print $telefon{"Mr. Foo"} . "\n";
# Abarbeitung in einer While-Schleife
while ( ($key, $value) = each(%telefon) )
{
    print "---" . $key . " : " . $value . "\n";
}
```

Die Ausgabe sieht wie folgt aus:



```
mc - /
oliver@server:~/uebperl1> ./hash.pl
Mr. Foo: 12345
Ms. Bar: 345345
12345
---Mr. Foo: 12345
---Ms. Bar: 345345
oliver@server:~/uebperl1>
```

Abbildung 5: Ausgabe hash.pl

Die Abarbeitung erfolgt mit einer While-Schleife. Man sieht dabei, dass man sowohl Zugriff auf den Schlüssel (`$key`), als auch natürlich auf den Wert (`$value`) hat.

6 Dateiverarbeitung

Rudimentär sei hier die Arbeit mit Dateien erklärt und zwar das Schreiben und das Lesen. Dazu legen wir die Datei `data.pl` mit folgendem Inhalt an:

```
#!/usr/bin/perl
use strict;

my $datei = "daten.txt";
open(DATEI, "<".$datei);
my @zeilen = <DATEI>;
close(DATEI);

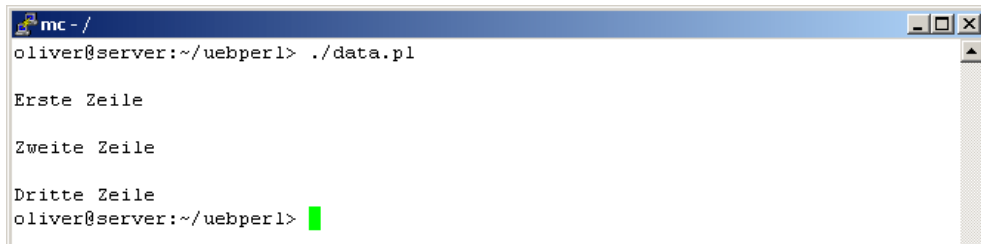
foreach (@zeilen)
{
    print "\n"._;
}

# Schreiben in eine Datei
open(DATEI2, ">".$datei . ".bak");
print DATEI2 @zeilen;
close(DATEI2);
```

Weiterhin legen wir im gleichen Ordner die Text-Datei `daten.txt` mit folgendem Inhalt an:

```
Erste Zeile
Zweite Zeile
Dritte Zeile
```

Die Ausgabe sieht wie folgt aus:



```
mc - /
oliver@server:~/uebperl> ./data.pl

Erste Zeile

Zweite Zeile

Dritte Zeile
oliver@server:~/uebperl> █
```

Abbildung 6: Ausgabe data.pl

Weiterhin sehen wir, dass die Datei `daten.txt.bak` existiert.

Der Programmablauf ist wie folgt:

1. Mit `open` wurde die in `$datei` spezifizierte Text-Datei über den Operator „<“ in das sogenannte Handle DATEI eingelesen.
2. Der Liste `@zeilen` wurde das Handle DATEI zugewiesen. D.h. jede Zeile der Datei ist jetzt als Element in dieser Liste repräsentiert.
3. Ausgabe der Liste mit Hilfe einer Foreach-Schleife und Schließen des Handle mit `close`.
4. Beim Schreiben wurde wieder der Befehl `open` benutzt. Allerdings ist der Operator diesmal „>“ um schreiben zu können. Der Name des Handle ist hier DATEI2.
5. Mit dem Print-Befehl wird eine Liste direkt in die Datei (das Handle) geschrieben.
6. Schließen des Handle.

Dieser exemplarische Ablauf ist wirklich nur rudimentär, zeigt aber die grundsätzliche Dateibehandlung.

7 Unterprogramme

Wir sind jetzt eigentlich schon so weit, dass wir jede Menge Zeilen Code untereinander schreiben könnten für durchaus sinnvolle Programme. Allerdings wird wohl nur das visuelle Ausgabeergebnis in Ordnung sein...

Wir sollten dem Spaghetti-Code einen Riegel verschieben und Code in Unterprogramme auslagern. Im folgenden Beispiel werden wir eine Addition durch ein Unterprogramm vornehmen lassen:

```
#!/usr/bin/perl
use strict;

my $zahl1 = 10;
my $zahl2 = 20;

my $ergebnis = &addiere($zahl1, $zahl2);
print $ergebnis."\n";

sub addiere
{
    my $lok1 = $_[0];
    my $lok2 = $_[1];
    my $erg = $lok1 + $lok2;
    return $erg;
}
```

Das Unterprogramm wird mit dem Schlüsselwort `sub` eingeleitet. Zu übergebende Parameter müssen nicht explizit nach dem Namen des Unterprogramm angegeben werden. Sie stehen im Unterprogramm gemäß der Reihenfolgen als Elemente im Array `$_` zur Verfügung. Die Rückgabe von Werten aus dem Unterprogramm, erfolgt mit `return`.

Der Aufruf des Unterprogramm geschieht mit `&addiere($zahl1, $zahl2)`. Da in Klammern Parameter übergeben werden, könnte man sogar auf das vorangestellte „&“ verzichten also, `addiere($zahl1, $zahl2)` schreiben.

8 Here-Document

Früher oder später wird man einmal vor dem Problem stehen, dass einem die Finger vom Tippen zu vieler Print-Anweisungen weh tun. Wenn man einmal mehr ausgeben will als zwei, drei Zeilen, bietet es sich an, statt der Print-Anweisung das sogenannte „Here-Document“ zu verwenden.

Aus

```
#!/usr/bin/perl
use strict;

print "Hallo Systembetreuer!\n";
print "Ein Rechner hat einen Plattenschaden\n";
print "Wir hoeren Geraeusche aus dem Rechner\n";
```

wird

```
#!/usr/bin/perl
use strict;

print <<MARKE;
Hallo Systembetreuer!
Ein Rechner hat einen Plattenschaden
Wir hoeren Geraeusche aus dem Rechner
MARKE
```

Der Effekt ist der gleiche, die Tipperei war aber bestimmt angenehmer. Man setzt mit `print <<MARKE;` eine Marke. So lange, bis die Marke im haargenau gleichen Wortlaut wieder auftaucht, kann man mehr oder weniger frei schreiben. Wichtig ist, dass nach `MARKE` in der Zeile kein Zeichen, d.h. auch kein Leerzeichen, mehr kommen darf!