Perl-Praxis

Subroutinen

Jörn Clausen
joern@TechFak.Uni-Bielefeld.DE

Übersicht

- Subroutinen
- Sichtbarkeitsbereiche, scoping
- Parameter
- Modularisierung

Subroutinen

- Warum Funktionen/Prozeduren/Methoden?
  - keine Code-Duplikation
  - Faktorisierung
  - Strukturierung
  - kein Spaghetti-Code
- in Perl: Subroutinen

Subroutinen, cont.

- Beispiel:

```
&usage if $#ARGV < 0;

sub usage {
  die "usage: $0 file [file ...]\n";
}
```

- alternativer Aufruf: `usage()`
- keine Vorwärtsdeklaration nötig

`$0` enthält den Namen des Perl-Skripts

Rückgabewert

- Block evaluiert zu letztem Ausdruck:

```
print "time: ", clock(),"\n";

sub clock {
  ($s, $m, $h) = localtime(time());
  $time = "$h:$m:$s";
}
```

- besser: expliziter Rückgabewert

```
sub clock {
  ($s, $m, $h) = localtime(time());
  return("$h:$m:$s");
}
```

Rückgabewert, cont.

- Liste/Array als Rückgabewert möglich:

```
return($h, $m, $s);
```

- schon gesehen: Listenzuweisung

```
($hour, $min, $sec) = clock();
```

- Rückgabe als Skalar oder Array vom Kontext abhängig

```
$time = clock();
```

- Abfrage des Kontextes:

```
sub clock {
  ($s, $m, $h) = localtime(time());
  if (wantarray) { return($h, $m, $s); }
  else           { return("$h:$m:$s"); }
}
```

Aufgaben

- Der Befehl `uname` liefert Informationen über den Rechner und sein Betriebssystem (siehe auch `uname(1)`).

  Schreibe eine Subroutine `hwinfo`, die im skalaren Kontext den Namen des Rechners zurückliefert und im Array-Kontext eine Liste mit dem Rechnernamen, dem Betriebssystem und dessen Versionsnummer.

  ```
  $name = hwinfo();
  ($name, $os, $ver) = hwinfo();
  ```

- Welche Ausgabe erzeugt

  ```
  print hwinfo(),"\n";
  ```

  In welchem Kontext wird die Subroutine also aufgerufen? Wende die Funktion `scalar` auf `hwinfo()` an.

Sichtbarkeit

- Blöcke durch geschweifte Klammern: `{ ... }`
- Sichtbarkeit von Variablen?

```
$a = 10;
{
  $a = 20;
  print "inside block: a is $a\n";
}
print "outside block: a is $a\n";
```

- Variablen in Perl global

```
inside block: a is 20
outside block: a is 20
```

lexical scoping

- lexikalische Variable in Block überdeckt globale Variable:

```
$a = 10;
{
  my $a = 20;
  print "inside block: a is $a\n";
}
print "outside block: a is $a\n";
```

- „Reichweite“: schließende Klammer des aktuellen Blocks

```
inside block: a is 20
outside block: a is 10
```

lokale Variablen

- temporäres Überschreiben globaler Variablen:

```
$a = 10;
$b = 11;
{
  my $a = 20;
  local $b = 21;
  print "inside block: a is $a, b is $b\n";
  printvars();
}
print "outside block: a is $a, b is $b\n";

sub printvars {
    print "in sub: a is $a, b is $b\n";
}
```

```
inside block: a is 20, b is 21
in sub: a is 10, b is 21
outside block: a is 10, b is 11
```

Parameterübergabe

- Parameter an Subroutine übergeben:

```
$teiler = ggT(15, 35);
```

- Parameter stehen in @_:

```
sub ggT {
  my $num1 = $_[0];
  my $num2 = $_[1];
  ...
}
```

- typisches Idiom:

```
sub ggT {
  my ($num1, $num2) = @_;
```

Aufgaben

- Die Folge der Fibonacci-Zahlen

  $$(1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots)$$

  ist folgendermaßen definiert:

  $$f_0 = 1$$
  $$f_1 = 1$$
  $$f_n = f_{n-1} + f_{n-2} \qquad \text{für } n \geq 2$$

  Implementiere die Subroutine `fib($n)` einmal rekursiv und einmal iterativ.

- Berechne die Fibonacci-Zahlen f_0 bis f_{25}. Wie unterscheidet sich die Laufzeit der beiden Implementierungen?

variable Parameterzahl

- in anderen Sprachen problematisch
- in Perl der Normalfall:

```
$sum1 = sum(5, 8, 4, 12, 7, 3);
$sum2 = sum(3, 9, 6, 1);

sub sum {
  my $sum = 0;
  foreach $num (@_) { $sum += $num; }
  return($sum);
}
```

- wieder „formale Parameter“:

```
my (@nums) = @_;
```

Aufgaben

- Das arithmetisches Mittel $\bar{x}$ und die Standardabweichung s einer Folge von Zahlen $x_1, x_2, \ldots, x_n$ berechnet sich durch

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

  Schreibe eine Subroutine `mean`, die im skalaren Kontext den Mittelwert und im Array-Kontext Mittelwert und Standardabweichung zurückliefert:

```
        $avg = mean(4, 7, 12, 9, 32, 19);
($avg, $dev) = mean(4, 7, 12, 9, 32, 19);
```

```
mean value: 13.8333333333333, standard deviation: 10.2648266749452
```

- Mittelwert und Standardabweichung:

```
sub mean {
    my (@xs) = @_;
    my $n = @xs;
    my $msum = 0;
    foreach $x (@xs) { $msum += $x; }
    my $xbar = $msum/$n;

    if (wantarray) {
        my $dsum = 0;
        foreach $x (@xs) { $dsum += ($x - $xbar)**2; }
        my $s = sqrt($dsum / ($n - 1));
        return($xbar, $s);
    } else {
        return($xbar);
    }
}
```

Parameterübergabe, cont.

- Was sind hier die Probleme?

```
@mix = zipper(@a, @b);  # (a0, b0, a1, b1, a2, b2, a3, b3)

(@even, @odd) = untangle(5,2,7,8,10,3,4,9,7,1);
```

- Wie lassen sie sich lösen?

```
@mix = zipper(\@a, \@b);

($even_ref, $odd_ref) = untangle(5,2,7,8,10,3,4,9,7,1);
```

- Implementiere die beiden Subroutinen `zipper` und `untangle`

```
}
    return(\@even, \@odd);
}
    else { push(@even, $num); }
if ($num % 2) { push(@odd, $num); }
foreach $num (@nums) {
my @even = (); my @odd = ();
my (@nums) = @_;
sub untangle {

}
return(@zip);
push(@zip, @a, @b);  # shift rest of longer list
}
    push(@zip, shift(@a), shift(@b));
while (@a and @b) {
my @a = @$a_ref; my @b = @$b_ref;
my @zip = ();
my ($a_ref, $b_ref) = @_;
sub zipper {
```

- Bei der Zuweisung der Arrays als Parameter in der Subroutine (`zipper`) bzw. als Rückgabewerte (`untangle`) werden die Listen von Arrays zu einem Array zusammengefaßt und "flachgeklopft". Es läßt sich nicht mehr feststellen, wo das eine Array endete und das zweite Array anfing.
- Mit Hilfe von Referenzen als Parameter lassen sich die beiden Subroutinen verwirklichen:

Aufgaben

- Miß die Laufzeit dieses Programms

```
@a = (1..100000);
foreach (1..100) { nop(@a); }

sub nop {
  my (@params) = @_;
}
```

mit Hilfe von `/usr/bin/time`.

Ändere das Programm so, daß das Array als Referenz übergeben wird. Wie ändert sich das Laufzeitverhalten des Programms? Erkläre das Ergebnis.

- Laufzeiten:

```
$ /usr/bin/time passcopy.pl
real       11.1
user       10.8
sys         0.1
$ /usr/bin/time passref.pl
real        0.4
user        0.3
sys         0.1
```

Parameterübergabe revisited

- Welche Ausgabe erzeugt dieses Programm? Wieso?

```
$p1 = $p2 = $p3 = 'call by value';
sub1($p1); sub2($p2); sub3(\$p3);
print "$p1\n$p2\n$p3\n";

sub sub1 {
    $_[0] = 'call by reference';
}
sub sub2 {
    my ($param) = @_;
    $param = 'call by reference';
}
sub sub3 {
    my ($param_ref) = @_;
    $$param_ref = 'call by reference';
}
```

```
call by reference
call by value
call by reference
```

Modularisierung

- probiere folgendes Programm aus:

```
say('Hello World');

package shouter;
sub say { print uc($_[0]),"\n"; }

package whisperer;
sub say { print lc($_[0]),"\n"; }
```

- ersetze Aufruf durch

```
shouter::say('Hello World');
```

bzw.

```
whisperer::say('Hello World');
```

```
Undefined subroutine &main::say called at ./say.pl line 3.
```

packages

- `package` definiert *Namensraum*:

```
package shouter;
$a = 20;
sub say {
    print uc($_[0]),"\n";
    print "a is $a\n";
}

package main;
$a = 10;
shouter::say('Hello World');
print "a is $a\n";
```

- `$a` ist Paket-global

```
HELLO WORLD
a is 20
a is 10
```

Module

- wiederverwendbaren Code in eigene Datei auslagern
- Datei `shouter.pm`:

```
package shouter;
1;
sub say { print uc($_[0]),"\n"; }
```

- Datei `say.pl`:

```
use shouter;
shouter::say('Hello World');
```

- Versionierung:

```
shouter.pm:    $VERSION = '1.3';
say.pl:        use shouter 1.2;
```

Module, cont.

- Namen exportieren

```
package shouter;
require Exporter;
@ISA = ("Exporter");
@EXPORT = ("say");
```

- in `say.pl`

```
say('Hello World');
```

- `require` ähnlich zu `use`
- Vererbung durch `@ISA`
- Export von Symbolen sparsam einsetzen

das andere Perl-Motto

- Tugenden eines Programmierers:

Laziness	**Faulheit**
Impatience	**Ungeduld**
Hubris	**Überheblichkeit**