Perl-Praxis

Dateien und Daten

Jörn Clausen
joern@TechFak.Uni-Bielefeld.DE

Übersicht

- Kommandozeilen-Parameter
- Informationen über Dateien
- Daten aus Dateien lesen
- Daten in Dateien schreiben
- mit pipes lesen und schreiben

Parameter

- Parameter an Skript übergeben:

```
$ linecount.pl brief.txt dipl.tex
```

- Argumente in @ARGV
- Skriptname nicht enthalten
- typische Verwendung:

```
foreach $name (@ARGV) {
  print "reading file $name\n";
  ...
}
```

Aufgaben

- Schreibe ein Skript, das eine Folge von Zahlen als Argument erwartet und die kleinste und die größte Zahl wieder ausgibt.

- Minimum und Maximum bestimmen:

```
$min = $max = shift(@ARGV);
foreach $num (@ARGV) {
    $min = $num if $num < $min;
    $max = $num if $num > $max;
}
```

Die Verwendung von `while`, z.B.

```
while ($num = shift(@ARGV)) {
```

empfiehlt sich an diesem Fall nicht. Die Schleife terminiert, wenn die Zahl „0" eingelesen wird. Dieses Problem kann mit Hilfe von `defined` umgangen werden:

```
while (defined($num = shift(@ARGV))) {
```

Die `foreach`-Schleife ist in diesem Fall aber einfach besser geeignet und sollte bevorzugt werden.

Datenverarbeitung mit Perl

- nachträgliche Auflösung des Akronyms „Perl“:

  Pratical Extraction and Report Language

- . . . oder auch:

  Pathologically Eclectic Rubbish Lister

- Verarbeitung von Dateien, vor allem Textdateien
- Einlesen und Parsen
- Integration mit Unix-Tools (`cat`, `more`, `head`, `tail`, `wc`, . . .)

Dateitestoperatoren

- Eigenschaften/Existenz von Dateien überprüfen
- von Shells übernommen

```
if (-f "/etc/passwd") { ... }  # is a file
if (-r "/etc/shadow") { ... }  # ooops, passwords readable?
if (-w "/etc/shadow") { ... }  # OOOPS, even writeable????
if (-x "/bin/ls") { ... }      # is executable
if (-d "/var/tmp") { ... }     # directory exists
```

- weitere Tests

Aufgaben

- Überprüfe die Eigenschaften der folgenden Dateien:

```
@files = ( "/etc/motd",
           "/tmp",
           "/kernel",
           "/dev/null",
           "/" );
```

- Teste Dateieigenschaften:

```
foreach $file (@files) {
    print "$file is\n";
    print "  a file\n" if -f $file;
    print "  a directory\n" if -d $file;
    print "  is readable\n" if -r $file;
    print "  is writeable\n" if -w $file;
    print "  is executable/accessible\n" if -x $file;
}
```

mit Dateien arbeiten

- Abstraktion: *filehandle*
- Datei lesen:

```
open(DATA, 'data.txt');
```

- Rückgabewert zeigt Fehler an
- typisches Idiom mit *short-circuit operator*:

```
open(DATA, 'data.txt') || die "can't open file\n";
```

- Fehlercode/Fehlermeldung in Variable `$!`:

```
open(DATA, 'data.txt') || die "can't open file: $!\n";
```

- Datei schließen:

```
close(DATA);
```

Großbuchstaben Konvention

Dateien lesen

- Zeilenweise aus Datei lesen:

```
open(DATA, 'data.txt') || die "can't open file: $!\n";
while ($line = <DATA>) {
  print "got line: $line";
}
close(DATA);
```

- *angle operator*: `<...>`
- liefert am Dateiende `undef`
- Datei komplett lesen:

```
@lines = <DATA>;
foreach $line (@lines) { ... };
```

Die vom *angle operator* zurückgelieferte Zeile enthält üblicherweise ein Newline. Dieses kann mit `chomp` entfernt werden.

Aufgaben

- Schreibe ein Programm, das eine Text-Datei einliest und die enthaltenen Zeilen in umgekehrter Reihenfolge wieder ausgibt. Reagiere auf mögliche Fehler (Datei nicht vorhanden, Datei nicht lesbar) mit geeigneten Fehlermeldungen.
- Schreibe ein Programm, das einen oder mehrere Dateinamen auf der Kommandozeile erwartet und die Anzahl der Zeilen in den Dateien ermittelt.

```
}
    print "$filename has $linecount lines\n";
    close(FILE);
  for ($linecount = 0; <FILE>; $linecount++) {}
  open(FILE, $filename) || next;
foreach $filename (@ARGV) {
```

- Zeilen zählen:

```
print reverse(@lines);
close(FILE);
@lines = <FILE>;
```

Etwas kürzer geht es, indem man die Datei auf einmal in ein Array einliest und es dann mit der Funktion `reverse` umdreht:

- Lies jeweils eine Zeile der Datei ein und füge diese vorne an ein Array an. Am Ende enthält das Array die Datei in umgekehrter Reihenfolge.

```
$filename = shift(@ARGV);
open(FILE, $filename) || die "error opening $filename: $!\n";
@lines = ();
while ($line = <FILE>) {
   unshift(@lines, $line);
}
close(FILE);
print @lines;
```

Ausgaben weiterverarbeiten

- Unix-Stil: Programme kombinieren:

```
$ ps -ef | grep mozilla | sort | less
```

- *pipe*: |
- Standardeingabe: `STDIN`

```
while ($line = <STDIN>) { ... }
```

- kein `open` oder `close`

Aufgaben

- Was macht das folgende Programm `reader.pl`

```
while ($line = <>) {
  $i++;
  print "$i: $line";
}
```

wenn man es folgendermaßen aufruft:

```
reader.pl /etc/motd /etc/resolv.conf
ls -l | reader.pl
ls -l | reader.pl /etc/motd /etc/resolv.conf
ls -l | reader.pl /etc/motd -
```

- Der *diamond operator* <> interpretiert die Argumente auf der Kommandozeile als Dateinamen, öffnet die Dateien nacheinander und liest sie zeilenweise ein. Falls kein Dateiname angegeben wurde, liest er stattdessen von der Standardeingabe. Man sollte den diamond operator also immer dann verwenden, wenn man ein filter-Programm schreiben will, das sich wie die klassischen Unix-Tools `cat`, `head`, `tail` usw. verhält.

  Falls sowohl Dateinamen als Parameter angegeben wurden, als auch Daten über STDIN bereitstehen, werden die Dateien gelesen und die Standardeingabe ignoriert.

  Der Dateiname „`-`" wird, je nach Kontext, als STDIN oder STDOUT interpretiert. Dies ist eine Konvention, die verschiedene Unix-Programme verstehen. Sie ist sinnvoll, wenn ein Programm unbedingt einen Dateinamen erwartet, man aber trotzdem Daten per pipe übergeben will.

Interaktion

- `STDIN` für Benutzereingaben:

```
print "please answer the following questions:\n";
print "name:  ";  $name = <STDIN>;   chomp($name);
print "age:   ";  $age = <STDIN>;    chomp($age);
print "email: ";  $email = <STDIN>;  chomp($email);
```

- `chomp` schneidet LF/CR-LF ab

Verzeichnisse lesen

- Verzeichnis ist auch nur Datei
- spezielle Befehle zur Interaktion:

```
opendir(DIR, "/") || die "can't open root directory\n";
while ($file = readdir(DIR)) {
  ...
}
closedir(DIR);
```

- oder auch wieder

```
@dirs = readdir(DIR);
```

- `readdir` liefert Dateinamen, nicht filehandle

Aufgaben

- Ändere das Programm mit den Dateitestoperatoren so, daß alle Dateien eines Verzeichnisses, das auf der Kommandozeile übergeben wird, auf ihre Eigenschaften untersucht werden.

```
closedir(DIR);

}
    print "   is executable/accessible\n" if -x $fullname;
    print "   is writeable\n" if -w $fullname;
    print "   is readable\n" if -r $fullname;
    print "   a directory\n" if -d $fullname;
    print "   a file\n" if -f $fullname;
    print "$fullname is\n";
    $fullname = "$dir/$file";
foreach $file (readdir(DIR)) {

opendir(DIR, $dir) || die "error opening directory $dir: $!\n";
$dir = shift(@ARGV);
```

- Teste Eigenschaften von mehreren Dateien:

Dateien schreiben

- filehandle zum Schreiben öffnen:

```
open(RESULT, '> results.txt') || die "can't open ...";
```

- > Ausgabeumlenkung in Shells
- filehandle mit `close` schließen
- Ausgabe in Datei:

```
print RESULT "$i zum Quadrat ist $i**2\n";
```

- eventuell vorhandene Datei wird überschrieben
- existierende Datei erweitern:

```
open(RESULT, '>> results.txt');
```

Falls die Datei hinter >> noch nicht existiert, wird sie einfach angelegt. Es wird keine Fehlermeldung ausgegeben.

Aufgaben

- Erweitere das Programm zum Zählen der Zeilen so, daß das Ergebnis in eine Datei „`linecount.out`“ geschrieben wird.
- Wann wird der Inhalt einer bereits vorhandenen Datei gelöscht? Wenn die Funktion `open` ausgeführt wird, oder wenn Daten mit `print` geschrieben werden?

- Die Datei wird bereits beim open gelert.

```
close(OUTFILE);

}
    print OUTFILE "$filename has $linecount lines\n";
    close(FILE);
    for ($linecount = 0; <FILE>; $linecount++) {}
    open(FILE, $filename) || next;
foreach $filename (@ARGV) {

open(OUTFILE, "> linecount.out") || die "error opening result file: $!\n";
```

- Zeilen zählen und Ergebnis in Datei schreiben:

Aufgaben

- Was macht das folgende Programm?

```
foreach $i (1..1000) {
  print "$i  ";
  print "\n" unless $i % 5;
  sleep(1);
}
```

- Füge folgende Zeile am Anfang ein:

```
$| = 1;
```

- Das Programm gibt Zahlen in fünfer-Gruppen pro Zeule aus. Nach jeder Zahl wird eine Sekunde gewartet. Es wird aber immer nur eine gesamte Zeile ausgegeben, d.h. alle fünf Sekunden werden fünf Zahlen auf einmal ausgegeben.
- Nun werden die Zahlen im Sekundentakt ausgegeben, auch wenn die Zeile noch nicht zu Ende geschrieben ist.

buffering

- `$|` beeinflußt aktuelles Ausgabe-filehandle
- `select` wählt Ausgabe-filehandle
- ungepufferte Ausgabe in Dateien:

```
open(OUT, "> numbers.txt") || die "can't open output\n";
select(OUT);
$| = 1;
foreach $i (1..1000) { ... }
close(OUT);
```

- Programm starten, dann

```
$ tail -f numbers.txt
```

- Was passiert bei gepufferter Ausgabe?

- Mit dem `tail`-Befehl kann man verfolgen, wie die Zahlen im Sekundentakt in die Datei geschrieben werden.
- Wenn man die Ausgabe wieder gepuffert erzeugt, sieht man lange Zeit nichts. Insbesondere wird nicht alle fünf Sekunden eine komplette Zeile angezeigt, sondern es dauert sehr viel länger, bis mehrere Zeilen auf einmal ausgegeben werden.

Die Größe des Puffers richtet sich nach der Art der Ausgabe. Bei Ausgabe nach STDOUT enthält der Puffer immer genau eine Zeile. Bei Ausgabe in eine Datei ist der Puffer mehrere KByte groß.

filehandles in/aus pipes

- seitenweise Ausgabe:

```
open(PAGER, '| more') || die "can't open pipe\n";
foreach $i (1..400) {
    print PAGER "$i^2 = $i**2\n";
}
close(PAGER);
```

- Ausgabe eines Befehls lesen:

```
open(WHO, 'who |') || die "can't open pipe\n";
while ($line = <WHO>) { ... }
close(WHO);
```

backtick evaluation

- noch ein Shell-Erbe:

```
@files = ‘ls‘;

$machine = ‘uname -a‘;
```

- Vorsicht vor unportablen Skripten

Aufgaben

- Der Befehl

  ```
  ps -ef
  ```

  zeigt alle laufenden Prozesse an. Filtere Deine eigenen Prozesse aus der Liste und gib das Ergebnis als PostScript-Datei aus. Verwende dazu das Programm `a2ps`. Das Ergebnis kannst Du mit Ghostview (`gv`) überprüfen.

- Liste von Prozessen als PostScript-Datei ausgeben:

```
$me = 'whoami'; chomp($me);
@procs = `ps -ef`;
open(A2PS, "| a2ps -1") || die "can't open pipe: $!\n";
foreach $line (@procs) {
    if (index($line, $me) != -1) {
        print A2PS $line;
    }
}
close(A2PS);
```

Mit der Funktion `grep` lassen sich die eigenen Prozesse etwas besser herausfiltern. Dann kann das Array mit einer `print`-Anweisung geschrieben werden:

```
@procs = grep(index($_, $me)!=-1, `ps -ef`);
print A2PS @procs;
```

Noch eleganter geht es mit einem Regulären Ausdruck:

```
@procs = grep(/^\s*$me/, `ps -ef`);
```