I/O-Grundlagen

Inhalt

Standard-Eingabe

- `<STDIN>` **liest** von der Standard-Eingabe:
  - Im skalaren Kontext **eine einzelne Zeile**.
  - Ist keine Zeile mehr vorhanden, gibt es `undef` zurück.
- **Beispiel**

```
while (defined($zeile = <STDIN>)) {
    print "Ich las $zeile";
}
```

Abkürzung (*genau so zu schreiben!*)

```
while (<STDIN>) {
    print "Ich las $_";
}
```


Standard-Eingabe

- `<STDIN>` **liest** von der Standard-Eingabe:
  - Im Listenkontext **alle Zeilen** bis zum Dateiende (*braucht evtl. viel Speicherplatz!*).
- **Beispiel**

```
chomp(@zeilen = <STDIN>);

foreach (<STDIN>) {
    chomp;
    print "$_ gelesen\n";
}
```


Diamant-Operator

- Der **Diamant-Operator** <> (**diamond**) simuliert das Verhalten von UNIX-Kommandos ("**Filtern**"):
  - Falls **Parameter** beim Aufruf eines Skripts angegeben wurden, das <> benutzt, werden sie als Dateinamen interpretiert und der Reihe nach zeilenweise eingelesen.

```
pgm.pl file1 file2 file3
```

(**program**)

  - Falls **keine Parameter** beim Aufruf dieses Skripts angegeben wurden, wird von der Standard-Eingabe gelesen (*auch durch explizite Angabe von "-" erreichbar*).

```
pgm.pl    oder    pgm.pl -
```


Diamant-Operator

- **Vorteil**: Erst zum Aufrufzeitpunkt eines Programms wird entschieden, **woher** seine Daten kommen:

Standard-Input:	`pgm.pl`	(*Tastatur*)
+ Umlenkung:	`pgm.pl < file1`	(*Datei*)
+ Pipe:	`grep "err" file* | pgm.pl`	(*Prozess*)
Argumente:	`pgm.pl file*`	(*Dateien*)

- <> entspricht `<STDIN>`, allerdings ist der Daten-Ursprung **beim Aufruf wählbar**.
  - Die Dateien werden **nacheinander** eingelesen, am Schluß wird `undef` zurückgegeben.

Diamant-Operator

Beispiel (*liest Zeile für Zeile*)

```
while (defined($zeile = <>)) {
    print "Zeile $zeile";
}
```

Beispiel (*liest alle Zeilen auf einmal*)

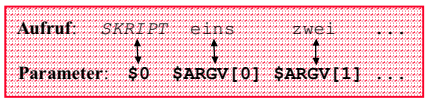
```
foreach (<>) {
    chomp;                          (Warum?)
    print "$_ gelesen\n";
}
```

Aufrufparameter

- Falls <> eine Datei **nicht findet**:
  - Gibt er eine **Fehlermeldung** aus.
  - Macht er mit der nächsten Datei weiter (*bricht nicht ab!*).
- Das Array **@ARGV** (**argument vector**) enthält die **Liste der Aufruf-Parameter**, die <> verwendet.
  - **$ARGV** enthält die aktuell von <> gelesene `Datei`.
  - Wie normale Arrays bearbeitbar (`shift`, `foreach`, ...), z.B. erst **Optionen** entfernen (*beginnen mit "-"*).
  - Ist `@ARGV` **leer** oder hat den Wert `("-")`, dann liest <> von der **Standard-Eingabe**.

Aufrufparameter

- Beim Skript-Aufruf auf der Kommandozeile können **beliebige Texte** als **Argumente** übergeben werden.
  - Der **Programmname** steht in `$0`.



- Mit `$ARGV[N]` wird auf die Argumente zugegriffen.
  - **Argument-Anzahl** erhält man durch **`scalar`** `@ARGV`.

Aufrufparameter

Beispiel (*Angabe der Parameter im Programm*)

```
@ARGV = qw/ a.dat b.dat c.dat /;
while (<>) {
    chomp;
    print "$_ gelesen\n";
}
```

- **Hinweis**: Die beim Aufruf angegebenen Argumente werden in diesem Beispiel ignoriert (*überschrieben*)!

Aufrufparameter

Beispiel (*Skript* `param.pl`):

```
#!/usr/bin/perl -w
print "Alle: @ARGV\n";
print "$0 $ARGV[0] $ARGV[1]\n";
print "$#ARGV ", scalar @ARGV, "\n";
```

Aufruf:

```
param.pl aa bb cc dd
```

Ergebnis:

```
Alle: aa bb cc dd
param.pl <aa> <bb>
3 4
```


Standard-Ausgabe

- `print` gibt eine Werte-Liste auf der **Standard-Ausgabe** aus, ohne etwas hinzuzufügen:
  - Zwischen der Ausgabe eines Arrays und eines interpolierten Arrays besteht ein kleiner Unterschied:

```
print @namen;    → TomHarryRick
print "@namen"; → Tom Harry Rick
```

  - **Trennzeichen** wird durch `$"` festgelegt (***Std**: Leerzeichen*).
  - Steht in jedem Element ein **Zeilenvorschub** `\n`, dann ist die 1. Variante sinnvoller (*warum?*).
  - Die 2. Variante sollte ein `\n` als Abschluß enthalten.

Standard-Ausgabe

- Die Ausgabe wird meist **gepuffert**, d.h. erst wird eine bestimmte Menge an Zeichen gesammelt, bevor etwas ausgegeben wird.
  → Performance-Gewinn.
- In bestimmten Fällen kann es sinnvoll sein, dieses Verhalten abzuschalten (→ *Performance-Verlust*).
  - Um z.B. **Fehler- oder Logmeldungen** sofort ausgeben.
  - Hierzu ist die Variable `$|` auf `1` zu setzen.

```
$| = 1;
```

Standard-Ausgabe

- Mit `print` und <> ist sehr einfach ein **Ersatz** für die 2 UNIX-Kommandos `cat` und `sort` erstellbar:

```
print <>;          (cat)
print sort <>;     (sort)
```

  – *So wird allerdings evtl. sehr viel Speicherplatz verbraucht!*
- `print` wertet seine Argumente im Listenkontext aus.
- `print` gibt **wahr** zurück, wenn die Ausgabe erfolgreich war, sonst **falsch**:

```
$ok = print "Hallo Welt!\n";
```


Standard-Ausgabe

- `print` kann auch mit **Klammern** aufgerufen werden und sieht dann wie eine **Funktion** aus:

```
print("Hallo Welt!\n");
```

- **Vorsicht**

```
print (2+3)*4;      Gibt 5 aus, nimmt den Rückgabe-
                    wert mal 4 und verwirft ihn.
print ((2+3)*4);    Macht das Richtige.
```

  – Ohne Klammern arbeitet `print` als **Listenoperator**.
  – Mit Klammern arbeitet es als **Funktion** und bearbeitet *nur* den Klammerinhalt.

Standard-Ausgabe

- **Alle Perl-Listenfunktionen** (*mit mehr als einen Argument, z.B.* `print`, `sort`) verhalten sich als Funktion anders als als Listenoperator:

```
print sort @namen, "\n";      (sortiert \n mit!)
print (sort @namen, "\n");    (sortiert \n mit!)
print (sort @namen), "\n";    (ignoriert \n!)
print ((sort @namen), "\n");  (arbeitet richtig!)
print (sort(@namen), "\n");   (arbeitet richtig!)
```

*Wenn man sich **nicht sicher** ist, lieber eine **Klammer zuviel** als zuwenig verwenden!*

Formatierte Ausgabe

- `printf` hat zusätzlich zu `print` als 1. Argument eine **Formatangabe**, die festlegt, wie die weiteren Werte auszugeben sind (**print format**, *analog C*).

```
printf "%s ist %d Jahre alt",
    $name, $alter{$name};
```

- Sie enthält **Formatelemente** (**Konversionen**), die mit "`%`" beginnen und mit einem **Buchstaben** enden.
  – **Sinnvoll**: Anzahl Konversionen = Anzahl der folgenden Listenelemente (*im obigen Beispiel 2*).
  – Soll ein `%` ausgegeben werden, ist es zu verdoppeln `%%`.

Formatierte Ausgabe

- **Konversionen** (*Auswahl*)

`%s`	**S**tring
`%d`	Ganze Zahl (**d**igit)
`%o`	Ganze Zahl **o**ktal
`%x`	Ganze Zahl **he**x**a**dezimal
`%f`	Fließkommazahl (**f**loat)
`%g`	Fließkommazahl (*autom. optimales Format*: **g**enerell)

  – Bei `%s` ist eine **Minimalbreite** angebbar (***rechtsbündig*** *bzw. negativ* ***linksbündig***): `%10s` `%-10s`
  – Bei `%f` ist **Anzahl Stellen insgesamt** und **Anzahl Nachkommastellen** angebbar `%6.2f`
  – Bei `%d` / `%o` / `%x` wird **abgeschnitten**, bei `%f` / `%g` **gerundet**.

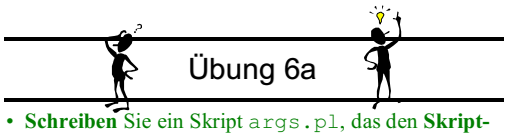
Formatierte Ausgabe

- `printf` wird normalerweise **nicht** mit einem **Array** benutzt, da die Anzahl der auszugebenden Elemente durch den **Formatstring** festgelegt ist.
- Allerdings ist auch der Formatstring **generierbar** (`@a` *steht im skalaren* ***und*** *im Listenkontext!*):

```
@a = (1, 2, 3, 4);
$fmt = "Erg:" . ("%8d " x @a) . "\n";
printf $fmt, @a;
```

oder

```
printf "Erg:" . ("%8d " x @a) . "\n", @a;
```

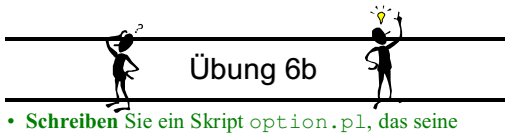


Übung 6a

- **Schreiben** Sie ein Skript `args.pl`, das den **Skript-Namen** (`$0`), die **Anzahl**, **letzten Index** und **alle Aufruf-Parameter** (`@ARGV`) ausgibt.
  - **Testen** Sie die Übergabe von Argumenten an das Skript.

Zusatzaufgabe (*unter UNIX*)

- **Erstellen** Sie einen **Soft-Link** `soft.pl` und einen **Hard-Link** `hard.pl` auf `args.pl` und rufen Sie das Perl-Skript über diese beiden Namen auf.
  - Was ändert sich an der Ausgabe?

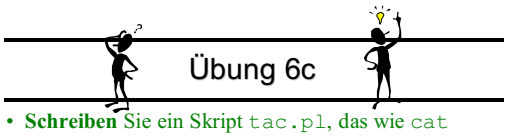


Übung 6b

- **Schreiben** Sie ein Skript `option.pl`, das seine Argumente in **Optionen** + **Argumente trennt**:
  - **Optionen** beginnen mit "`-`" (*in* `@OPTS` *speichern*).
  - **Argumente** beginnen **nicht** mit "`-`" (*in* `@ARGS` *speichern*).
  - Als Abschluß folgende **Ausgabe** erzeugen:

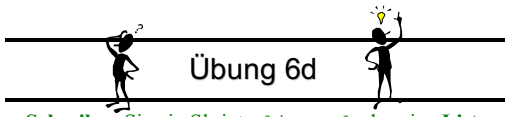
```
print "Optionen:  @OPTS";
print "Argumente: @ARGS"; (nicht @ARGV!)
```

- **Schreiben** Sie ein Skript `printf.pl`, das **die Funktion** `printf` mit den diversen **Konversionen** `%X` ausprobiert (`perldoc -f printf`):



Übung 6c

- **Schreiben** Sie ein Skript `tac.pl`, das wie `cat` funktioniert, die **Reihenfolge der Zeilen jedoch herumdreht** (*letzte Zeile der letzten Datei zuerst bis 1. Zeile der 1. Datei zuletzt*).
  - Kehren Sie auch noch die **Zeichenreihenfolge** um (`\n` *!*).
- **Schreiben** Sie ein Skript `uniq.pl`, das wie `uniq` funktioniert, d.h. **direkt hintereinander stehende gleiche Zeilen** durch eine ersetzt.
  - Schalter `-d` ergänzen (**double**, *nur doppelte ausgeben*).
  - Schalter `-u` ergänzen (**unique**, *nur einfache ausgeben*).

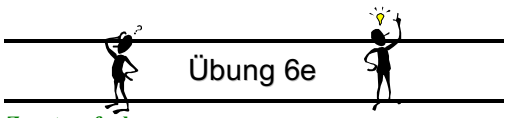


Übung 6d

- **Schreiben** Sie ein Skript `align.pl`, das eine **Liste von Zahlen** `zahlen.dat` einliest und pro Zeile 5 Zahlen nebeneinander in der Breite 8 Zeichen ausgibt.
  - **Geben** Sie darüber folgendes "**Lineal**" aus:

```
00000000111111111122222222233333...
12345678901234567890123456789012...
```

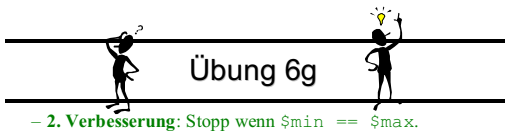
  - **Ergänzen** Sie die Möglichkeit, auf der **Kommandozeile** die Anzahl der nebeneinander stehenden Zahlen anzugeben.
  - **Verlängern** Sie das Lineal entsprechend der Angabe auf der Kommandozeile.



Übung 6e

Zusatzaufgabe

- **Schreiben** Sie ein Skript `guess2.pl`, das versucht, eine von Ihnen vorgegebene Zahl `1-100` zu **erraten**.
  - Merkt sich **noch möglichen Bereich** (`$min..$max`).
  - Rät **mittleren Wert** (`int(($min+$max)/2)`).
  - Je nach Ihrer Eingabe "`r/k/g`" ist der geratene Wert
    - `r` = **richtig** → Stoppt und gibt Anzahl Ratevesuche aus.
    - `k` = **zu klein** → Ändert **linke Grenze** `$min` auf Ratewert+`1`.
    - `g` = **zu groß** → Ändert **rechte Grenze** `$max` auf Ratewert-`1`.
  - **1. Verbesserung**: Erste Zahl **zufällig** raten (`rand()`).

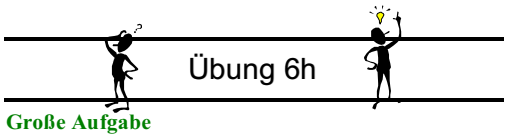


Übung 6g

  - **2. Verbesserung**: Stopp wenn `$min == $max`.
  - **3. Verbesserung**: Betrügerische/falsche Eingabe entdecken.

Zusatzaufgabe

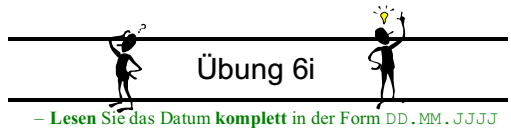
- **Schreiben** Sie ein Skript `chkguess.pl`, das `guess2.pl` für alle 100 Fälle **testet** und ausgibt, **wieviel Rateversuche durchschnittlich nötig** waren.



Übung 6h

Große Aufgabe

- Die Tage eines Jahres sind von 1 bis 365 (*bzw. 366 in einem Schaltjahr*) durchnumeriert. **Schreiben** Sie ein Perl-Programm `date2day.pl`, das einen Tag 1..31 und einen Monat 1..12 einliest und zu diesem Datumswert den **Jahrestag** ausrechnet.
  - **Berücksichtigen** Sie zunächst Schaltjahre **nicht**.
  - **Lesen** Sie zunächst nur Tag + Monat **einzeln** ein.
  - **Prüfen** Sie, ob Tag + Monat im **gültigen Bereich** sind.
  - **Lesen** Sie solange ein, **bis** Tag + Monat **gültig** sind.

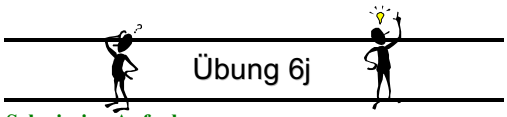


Übung 6i

- **Lesen** Sie das Datum **komplett** in der Form `DD.MM.JJJJ` ein (*auch Trennzeichen "/" erlauben*).
- **Berücksichtigen** Sie **Schaltjahre** beim Überprüfen des Datums und bei der Berechnung, ihre Definition lautet:

```
Schaltjahr(JAHR) = JAHR teilbar durch 4 und
       JAHR NICHT teilbar durch 100 oder
       JAHR teilbar durch 400
```

- Erlauben Sie die **Übergabe** des Datums beim Aufruf des Programms **auf der Kommandozeile**.
- Bei Fehlern ist eine **Usage-Meldung** auszugeben.
- **Option** `-d` soll einige **Debugging-Ausgaben** aktivieren.



Übung 6j

Schwierige Aufgabe

- **Schreiben** Sie ein Perl-Programm `day2date.pl`, das zu einem Jahrestag 1..365/366 und einen Jahr das Datum `DD.MM.JJJJ` ermittelt.
  - **Berücksichtigen** Sie **Schaltjahre**.
  - **Testen** Sie den Algorithmus, indem Sie Ergebnisse von `date2day.pl` verwenden.