Programmieren 1 Übung: Einfache Schleifen

Klaus Kusche

1.) Rechnen mit ganzen Zahlen: Potenzieren

C kennt zwar eine vordefinierte Potenz-Funktion **pow** für Kommazahlen
(die sehr langsam und kompliziert rechnet),
aber keine Potenzrechnung für ganze Zahlen, also schreiben wir eine:

a^0 ist 1, und für a^n wird *a* einfach *n-mal nacheinander zu 1 dazumultipliziert*.

Hinweise:

- Das Programm bekommt die beiden Zahlen beim Aufruf auf der Befehlszeile mitgegeben und soll mit **`printf`** ein schönes Ergebnis ausgeben.

  Beispiel:
  Aufruf "**`power 2 8`**"
  Ausgabe "**`2 hoch 8 = 256`**"

- Da **`argv`** ja Texte enthält und keine Zahlen, müssen wir **`argv[1]`** und **`argv[2]`** (jeweils einzeln nacheinander) in eine Zahl verwandeln. Dazu gibt es eine vordefinierte Funktion **`atoi(...)`** ("Ascii to integer", aus **`stdlib.h`**):
  In diese Funktion steckt man einen Text hinein, und zurück kommt der Wert der ganzen Zahl, die dieser Text darstellt (oder **0**, wenn der Text gar keine Zahl ist).
  Diesen Wert kannst du dann in einer **`int`**-Variable speichern.

- Du findest online ein Rahmenprogramm, in dem die Kommentare
  Schritt für Schritt schon drinstehen, aber der eigentliche Programmcode fehlt.

- Wenn wir ganz gründlich sind: Was müssten wir eigentlich noch tun, bevor wir mit der Schleife anfangen? (**`int`** und **`atoi`** kennen Zahlen mit Vorzeichen!)

- Ich empfehle, schrittweise zu arbeiten: Mach zuerst einmal die Ein- und Ausgabe ohne die Schleife und die Multiplikation (d.h. es wird immer "… **`= 1`**" angezeigt), damit du zuerst einmal siehst, ob dein **`atoi`** usw. richtig funktioniert.
  Dann füge dazwischen die Schleife ein.

Zusatzaufgabe:

- Kannst du das Programm so umbauen, dass die erste der beiden Zahlen und das Ergebnis auch eine *Kommazahl* sein können, d.h. vom Typ **`double`** sind?
  Du brauchst dazu **`atof`** statt **`atoi`** zum Einlesen und **`%g`** statt **`%d`** zur Ausgabe.

  Dann kannst du auch Potenzen mit *negativen ganzen Exponenten* berechnen:
  a^{-n} ist dasselbe wie $(1/a)^n$ oder $1/(a^n)$

2.) Die `while`-Schleife: Größter gemeinsamer Teiler nach Euklid

Der größte gemeinsame Teiler (ggT, englisch "greatest common divisor", gcd)
zweier Zahlen ist die größte Zahl, durch die sich beide Zahlen *glatt teilen* lassen.
Der ggT von 15 und 6 ist beispielsweise 3. Man braucht den ggT zum Kürzen von Brüchen.

Man berechnet den ggT immer von den *Absolutwerten* der beiden gegebenen Zahlen.
In C liefert dir die Funktion **abs** (aus **stdlib.h**) als Ergebnis den Absolutwert eines **int**.

In der Schule berechnet man den ggT durch Primfaktorenzerlegung beider Zahlen und Zusammenmultiplizieren der gemeinsamen Faktoren, aber seit den alten Griechen ist ein viel besseres Verfahren (zumindest für Computer) bekannt, um den ggT von *a* und *b* zu ermitteln:

- Wiederhole die folgenden beiden Schritte, solange dein *b* nicht 0 ist:
  - Ermittle den Rest *r* der Division *a* / *b*.
  - Verwende das bisherige *b* als dein neues *a* und *r* als dein neues *b*.
- Wenn dein *b* 0 ist, so ist *a* der gesuchte ggT.

Jetzt haben wir also keine Schleife, die eine fixe Anzahl von Durchläufen macht und dabei mitzählt, sondern eine, die etwas *immer wieder* macht, *solange eine Bedingung gilt* (ohne vorher zu wissen, wie oft, und ohne mitzuzählen).

In C sieht das wie folgt aus:

```
while (bedingung) {
  Befehle in der Schleife;
}
```

Außerdem brauchst du noch die *Prüfung auf "ungleich"* (**!=** statt **==**) und die *Restbildung*: **a % b** liefert (für **b** ungleich 0) den Rest der ganzzahligen Division **a / b**.

Zusatzaufgabe "Bruch kürzen":

- Speichere das *ursprüngliche* *a* und *b* in zwei zusätzlichen Variablen, damit du am Ende den gekürzten Bruch ausgeben kannst!
- Erkenne *a* / *0* (==› Fehlermeldung) und behandle *negative* Zähler und Nenner richtig (das – gehört in der Ausgabe immer in den Zähler, zwei – heben sich auf!).
- Und wenn die Ausgabe ganz schön sein soll: Unterscheide Bruch und ganze Zahl! (d.h. wenn der gekürzte Nenner 1 ist, wird *nur der Zähler* ausgegeben)

3.) Und noch eine `while`-Schleife: Quersumme

Die Quersumme einer Zahl ist die *Summe ihrer Ziffern*, die Quersumme von 123 ist beispielsweise 6. Bedeutung hat die Quersumme als Teilbarkeitsregel: Ist die Quersumme einer Zahl glatt durch 3 teilbar, so ist auch die Zahl selbst glatt durch 3 teilbar, und für 9 gilt dasselbe. Bei negativen Zahlen wird die Quersumme vom *Absolutbetrag* gerechnet.

Schreib ein Programm, das mit einer Zahl auf der Befehlszeile aufgerufen wird und deren Quersumme ausgibt.

Du sollst **argv[1]** in eine Zahl verwandeln (wie?) und dann mit **/** und **%** (Division und Restbildung; die Division rundet nicht, sondern schneidet ab!) die einzelnen Ziffern der Zahl ermitteln (nicht direkt auf die einzelnen Buchstaben in **argv[1]** zugreifen!).

Die wichtigste Überlegung bei der Lösung ist, wie die *Schleife* (vermutlich eine **while**-Schleife!) aussehen muss, damit sie *Ziffer für Ziffer* arbeitet und aufhört, wenn keine Ziffern mehr übrig sind.

Es gibt mehrere richtige Möglichkeiten, aber ich empfehle, in jedem Durchlauf der Schleife die *hinteste Ziffer* der Zahl zu extrahieren (mit welcher Restrechnung bekommt man die hinteste Ziffer einer Zahl?), aufzusummieren, und wegzudividieren (durch was dividiert man eine Zahl, damit die letzte Ziffer wegfällt?), bis die Zahl 0 ist.

Alle anderen Dinge, die du brauchst, hatten wir schon einmal…

Zusatzaufgabe:

- Berechne die *wiederholte* Quersumme, d.h. so lange immer wieder die Quersumme der Quersumme, bis eine *einstellige* Zahl überbleibt (auch mit der wiederholten Quersumme funktioniert der Teilbarkeitstest für 3 und 9).

  *Dazu wirst du zwei Schleifen **ineinander** brauchen!*

4.) Rechnen mit Gleitkommazahlen: Wurzelziehen

Jetzt verwenden wir Kommazahlen statt ganze Zahlen:

- Der Typ dafür heisst **`double`** statt **`int`**.
- Der **`printf`**-Platzhalter heisst **`%f`** statt **`%d`**
  (bzw. z.B. **`%.15f`**, wenn du 15 Kommastellen sehen willst).
  Für besonders große oder kleine Zahlen ist **`%.15g`** besser.
- Und das Umwandeln von Text auf Kommazahl funktioniert mit **`atof`** statt **`atoi`**.

Die Aufgabe ist folgende:

"Ermittle die Wurzel einer gegebenen Zahl nach folgendem schrittweisen Näherungsverfahren:

- *Beginne mit der Zahl selbst als erstem Näherungswert.*
- *Berechne in jedem Durchlauf den neuen Näherungswert als Mittelwert des alten Näherungswertes und des Divisionsergebnisses aus ursprünglicher Zahl durch alten Näherungswert.*
- *Wiederhole das, bis das Quadrat des Näherungswertes erst nach der zehnten Stelle von der ursprünglichen Zahl abweicht (anders ausgedrückt: Wiederhole das, solange der Absolutwert von (Quadrat des Näherungswertes - ursprüngliche Zahl) größer ist als (ursprüngliche Zahl * 10^{-10}))."*

Beim Prüfen auf das Milliardstel Abweichung wird die Bedingung einfacher, wenn du die Funktion **`fabs`** verwendest (wie **`abs`**, aber für **`double`**: Absolutbetrag, d.h. das Ergebnis ist der hineingesteckte Wert ohne Vorzeichen).
Für **`fabs`** gilt: Ein **`double`** geht rein, ein **`double`** kommt raus, kommt aus **`math.h`** .

Tipps / Zusatzaufgaben:

- Zum Vergleich kann dein **`printf`** auch das Ergebnis der *eingebauten Wurzelfunktion* ausgeben: Die Funktion heißt **`sqrt`** (ein **`double`** geht rein, und das Ergebnis ist auch ein **`double`**), und sie kommt aus **`math.h`** .
- Für Neugierige: Kannst du die *Anzahl* der Näherungsschritte ausgeben? Den *Näherungswert nach jedem Schritt*?
- Und die übliche Frage: *Wie deppensicher* ist dein Programm? (Was kann bei der *Eingabezahl* für die Wurzel *schiefgehen*?)