C Spickzettel:
Was haben wir bisher von C verwendet?

Klaus Kusche, 2010 / 2011

Unser Programmgerüst

Alle Programme werden bei uns in den nächsten Wochen Folgendes gemeinsam haben:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])
{
  /* hier kommen zuerst die Deklarationen ... */

  /* ... und dann die Programm-Befehle */

  return 0;
}
```

Die "**#include**" sorgen dafür, dass das Programm die vordefinierten Funktionen kennt:

- **stdio.h** enthält alle Funktionen für die Ein- und Ausgabe, u.a. das **printf**
- **stdlib.h** ist für **atoi**, **abs**, **exit** und dergleichen zuständig
- Hin und wieder werden wir noch mehr brauchen,
  z.B. **math.h** für die mathematischen Funktionen.

Zeilen mit **#** am Anfang haben keinen **;** am Ende!

Das **main** heißt "jetzt kommt der Code für die Funktion **main**".
Das ist die Funktion, die gerechnet wird, wenn wir unser Programm starten.
Das **int** vor **main** heißt, unser **main** liefert einen **int**-Wert zurück, wenn es endet
(nämlich den "Erfolgscode" für das Betriebssystem).
Die Dinge in den **( )** werden uns vom Betriebssystem geliefert,
wenn es unser **main** startet:

- Im **argc** (einer ganzen Zahl) steht die *Anzahl* der Worte auf der Befehlszeile
  (immer mindestens 1, nämlich der Programmname selbst).
- Im **argv** (mehrere Texte) stehen die *einzelnen Worte* als Text:
  argv[0] ist der Programmname,
  und **argv[1]** bis **argv[argc - 1]** sind die Worte,
  die der Aufrufer dahinter zusätzlich eingetippt hat.

Das **return 0;** heißt "beende das Programm und melde 'alles ok' zurück".
Wir können das Programm auch mittendrin mit weiteren **return** beenden
(z.B. wenn wir nach Ausgabe einer Fehlermeldung aufhören wollen),
aber dann sollten wir **return 1;** verwenden
("beende das Programm und melde 'Fehler aufgetreten' zurück").

Noch schöner wäre **exit(0);** (oder gar **exit(EXIT_SUCCESS);**) statt **return 0;**

und **exit(1);** (oder gar **exit(EXIT_FAILURE);**) statt **return 1;** .

Kommentare

```
/* Das ist ein Kommentar */
// Das auch...
```

Enthält Anmerkungen für den Menschen, wird vom Computer ignoriert.

Variablen-Deklarationen

Allgemein: *typ variablenname* **;**
Beispiel: **int a;**

In Computer-Deutsch: "Wir deklarieren eine Variable namens **a** vom Typ **int**."
Technisch:
"Reserviere einen Speicherplatz, in dem du einen **int**-Wert (Ganzzahl) speichern kannst. Immer wenn ich ab jetzt **a** schreibe, meine ich den Wert in diesem Speicherplatz."

Bisher kennen wir die Typen **int** (für ganze Zahlen) und **double** (für Kommazahlen), später kommen noch weitere dazu (z.B. **char** für Buchstaben).

Auch möglich:
Mehrere Variablen vom selben Typ: **int a, i, n;**
Angabe eines Initialisierungswertes: **int zaehler = 0, nenner = 1;**
(sonst hat eine Variable bis zur ersten Zuweisung einen zufälligen Wert, d.h. Blödsinn!)
Demnächst: Deklaration von Arrays: **int a[10];**
("**a** ist ein Array von 10 **int**-Werten **a[0]** bis **a[9]**")

Rechnen

Plus, minus, mal, dividiert: **+ − * /**
Divisionsrest: **%** (das Ergebnis von **a % b** ist der Rest der ganzzahligen Division **a / b**)
Vorzeichen umdrehen: **−** (z.B. **−a**)

Es gelten die üblichen Vorrang-Regeln: Vorzeichen-"−" vor *** / %** vor **+ −**
Man kann auch **( )** setzen.

In Rechnungen kann man Variablen (**a**, **zaehler**, ...) und Konstanten (**1**, **0.5**, ...) sowie einzelne Elemente von Arrays (**a[i]**, **argv[1]**) verwenden.

Wenn beide Seiten von *** / + −** vom Typ **int** sind, ist das Ergebnis auch **int**
(bei **/**: Abgeschnitten, nicht gerundet!)
Bei zwei **double** kommt ein **double** heraus.
Bei einem **double** und einem **int** wird zuerst der **int** in einen **double** verwandelt und dann mit zwei **double** gerechnet.

Funktionen

In Rechnungen können auch Funktionen verwendet werden.
Bisher kennen wir die vordefinierten Funktionen

- **`atoi(`***text*`)`**: Wenn *text* eine Zahl enthält, liefert **`atoi`** den Wert dieser Zahl. ("Ascii to int")
  Dasselbe für **`double`**: **`atof`** ("Ascii to float")
- **`abs(`***zahl*`)`**: Liefert *zahl* ohne Vorzeichen ("Absolutbetrag").
  Dasselbe für **`double`**: **`fabs`**

Wir werden noch mehr vordefinierte Funktionen kennenlernen
und später auch selbst welche schreiben.

Auch **`printf`** und **`exit`** sind technisch gesehen Funktionen,
nur interessiert uns der Wert nicht, den sie ausrechnen und zurückliefern,
sondern nur ihre "Wirkung": Ausgabe bzw. Programmende.

Speichern

Allgemein: *variable* **`=`** *rechnung* **`;`**

"Rechne *rechnung* (rechte Seite) aus und speichere den Wert,
der dabei herauskommt, in *variable* (linke Seite)."
(in EDV-Deutsch: "Zuweisung: Weise der Variable *variable* den Wert *rechnung* zu.")

Das heißt:

- Zuerst wird die rechte Seite ausgerechnet.
- Die linke Seite wird *nicht* ausgerechnet, sondern gibt die Variable an,
  d.h. den Speicherplatz, in dem wir uns das Ergebnis der Rechnung merken wollen.

Beispiel: **`m = (a + b) / 2;`** ("in **`m`** wird der Mittelwert von **`a`** und **`b`** gespeichert")
Bald haben wir auch: **`a[i] = n % 10;`**
("speichere die letzte Ziffer von **`n`** im **`i`**-ten Element von **`a`**")
Man kann einen Wert auch in mehreren Variablen speichern: **`x = y = z = 1;`**

Abkürzungen:

- **`++i;`** oder **`i++;`** ist (fast) dasselbe wie **`i = i + 1;`** ("zähle zu **`i`** eins dazu")
  Ebenso **`--i;`** oder **`i--;`** ("zähle von **`i`** eins weg")
- **`i += 2;`** steht für **`i = i + 2;`** ("zähle zu **`i`** zwei dazu")
  `n -= i;` steht für **`n = n - i;`** ("ziehe **`i`** von **`n`** ab")
  `n *= 2;` steht für **`n = n * 2;`** ("multipliziere **`n`** mit 2")
  `n /= 10;` steht für **`n = n / 10;`** ("dividiere **`n`** durch 10")

Vergleiche, Bedingungen

Allgemein: *linke Seite* **`<`** *rechte Seite* (oder **`<=`** **`>`** **`>=`** **`==`** **`!=`**)
Das heißt:

- Rechne die *linke Seite* aus,

- rechne die *rechte Seite* aus,
- vergleiche die beiden Ergebnisse wie angegeben
- und liefere als Resultat "wahr" (stimmt) oder "falsch" (stimmt nicht).

Beispiele: `n > 0` `i < argc` `n % 10 == 0`

Vergleiche stehen in den `( )` von `if` und `while`
(und in der Mitte von `( ; ; )` beim `for`)

Irgendwann werden wir auch zusammengesetzte Bedingungen brauchen:
`(`*vergleich1*`) && (`*vergleich2*`)`
"und": liefert "wahr", wenn beide Vergleiche "wahr" sind
`(`*vergleich1*`) || (`*vergleich2*`)`
"oder": liefert "wahr", wenn zumindest einer der beiden Vergleiche "wahr" ist
`!(`*bedingung*`)`
"nicht": liefert "wahr", wenn die *bedingung* "falsch" ist

Wenn … dann …: Der `if`*-Befehl*

Allgemein:
```
if (vergleich) {
  dann-Befehle;
} else {
  sonst-Befehle;
}
```

Das heißt:
- Rechne den *vergleich* aus.
- Wenn er "wahr" ergibt, mach die *dann-Befehle* (genau einmal).
- Wenn er "falsch" ergibt, lass die *dann-Befehle* aus,
  und mach stattdessen die *sonst-Befehle* (auch genau einmal).
- Dann mach normal nach der letzten `}` weiter.

Wenn im sonst-Fall nichts zu tun ist, kann man das `else` komplett weglassen:
Dann wird nichts gemacht, wenn der *vergleich* "falsch" ergibt.

Achtung: Kein `;` hinter den `( )` !!!

Beispiele:
```
if (x < 0) {
  x = -x;
}

if (a < b) {
  min = a;
} else {
  min = b;
}
```

Und bei einer ganzen Kette von Prüfungen / Fällen nacheinander:

```
if (x < 0) {
  printf("Negativ!\n");
} else if (x > 0) {
  printf("Positiv!\n");
} else {
  printf("Null!\n");
}
```

In die **( )** gehört das "wenn" (was geprüft werden soll),
und in die **{ }** gehört das "dann" (was getan werden soll, wenn die Prüfung zutrifft),
und in die **{ }** nach dem `else` gehört das "sonst" (was getan werden soll,
wenn die Prüfung nicht zutrifft).

Solange … wiederhole …: Die **while**-*Schleife (und die* **do-while**-*Schleife)*

Allgemein:

```
while (vergleich) {
  befehle;
}
```

Kurz: Solange *vergleich* "wahr" ergibt, wiederhole die *befehle* immer wieder.

Lang:

- Rechne den *vergleich* aus.
- Wenn er "wahr" ergibt, mach die *befehle* einmal.
- Rechne wieder den *vergleich* aus.
- Wenn er wieder "wahr" ergibt, mach die *befehle* noch einmal.
- Rechne wieder den *vergleich* aus, mach wieder die *befehle*, usw.
- Wenn der *vergleich* dann endlich einmal "falsch" ergibt,
  lass die *befehle* aus, und mach normal nach der **}** weiter.

Wenn der *vergleich* schon von Anfang an "falsch" ergibt,
werden die *befehle* gar nicht gemacht.

Achtung: Kein **;** hinter den **( )** !!!

Beispiele:

```
while (n > 0) {
  s += n % 10;
  n /= 10;
}
```

In die **( )** gehört das "solange" (was vor jeder Wiederholung geprüft werden soll),
und in die **{ }** gehört das "wiederhole"
(was immer wieder getan werden soll, solange die Prüfung zutrifft).

Wenn in der Lösungsidee "wiederhole … bis" statt "wiederhole … solange" vorkommt,
so ist das auch eine **while**-Schleife, aber man muss die Prüfung umdrehen!
In den **( )** vom **while** steht dann als *vergleich* das genaue *Gegenteil* vom "bis …".
Beispiel: "wiederhole … bis **a** und **b** gleich sind" wird zu **while (a != b) { ... }**

d.h. “wiederhole … solange **a** und **b** verschieden sind”

Es gibt auch eine **while**-Schleife, nämlich die **do–while**-Schleife, die den *vergleich* erst *nach* dem ersten Ausführen der *befehle* prüft. Die braucht man, wenn das, was geprüft werden soll, erst in der Schleife selbst ausgerechnet oder eingegeben wird.
Diese Schleife schaut allgemein wie folgt aus:

```
do {
   befehle;
} while (vergleich);
```

Kurz: Wiederhole die *befehle* immer wieder, solange *vergleich* “wahr” ergibt.

Lang:

- Mach die *befehle* einmal.
- Rechne den *vergleich* aus.
- Wenn er “wahr” ergibt, mach die *befehle* noch einmal.
- Rechne wieder den *vergleich* aus.
- Wenn er wieder “wahr” ergibt, mach die *befehle* noch einmal, vergleiche wieder, usw.
- Wenn der *vergleich* dann endlich einmal “falsch” ergibt, mach normal nach dem **while (***vergleich***);** weiter.

Achtung: Die *befehle* werden mindestens ein Mal gemacht,
selbst wenn der *vergleich* schon vor dem ersten Mal “falsch” wäre!

Achtung: Diese Schleife hat ausnahmsweise einen **;** hinter den **( )** der Schleife!!

Beispiel:

```
do {
  cout << "Was sagt der Würfel? ";
  cin >> wurf;
  ++cnt;
} while (wurf != 6);
```

Auch hier gilt: Der *vergleich* muss “wahr” sein, solange wiederholt werden soll, und “falsch”, wenn die Schleife aufhören soll. Wenn in der Lösungsidee “wiederhole … bis …” steht (also “hör auf, wenn das ’bis’ gilt”), muss (*vergleich*) genau das Gegenteil von “bis …” prüfen!

Wiederhole … x mal, zähle von – bis, usw.: Die for*-Schleife*

Allgemein:

```
for (anfangszuweisung; vergleich; weiterzählen) {
  befehle;
}
```

Bedeutet:

- Mach ein einziges Mal die *anfangszuweisung*.
- Rechne den *vergleich* aus.

- Wenn er “wahr” ergibt, mach die *befehle* einmal und danach das *weiterzählen*.
- Rechne wieder den *vergleich* aus.
- Wenn er wieder “wahr” ergibt,
  mach die *befehle* und danach das *weiterzählen* noch einmal.
- Rechne wieder den *vergleich* aus, mach wieder die *befehle*, zähle weiter, usw.
- Wenn der *vergleich* dann endlich einmal “falsch” ergibt,
  lass die *befehle* aus, und mach normal nach der **}** weiter.

Wenn der *vergleich* schon von Anfang an “falsch” ergibt,
werden die *befehle* und das *weiterzählen* gar nicht gemacht
(aber die *anfangszuweisung* vorher schon!).
Das passiert beispielsweise, wenn ich von 1 bis **n** zählen will, und **n** ist 0.

Normalerweise hat eine **for**-Schleife eine Zählvariable,
die oft **i** (oder **j**, …) genannt wird:

- Die *anfangszuweisung* setzt die Zählvariable auf den *Anfangswert* (ersten Wert),
  für den die *befehle* das erste Mal gemacht werden sollen,
  also z.B. **i = 0** oder **i = 1** (oder **i = n - 1** wenn man hinunterzählt).
- Das *weiterzählen* sorgt dafür, dass die Zählvariable jedesmal,
  nachdem die *befehle* gemacht worden sind,
  den *nächsten Wert* bekommt (für die nächste Wiederholung der *befehle*).
  Das wird fast immer **++i** sein (zähle zu **i** eins dazu),
  oder **--i** (wenn die Schleife runterzählt statt raufzählt).
  Es kann aber z.B. auch **i += 2** sein (zähle in Zweierschritten)
  oder **i *= 10** (dann zählt die Schleife 1, 10, 100, 1000, …).
- Der *vergleich* wird fast immer prüfen,
  ob das **i** beim Zählen schon den *Endwert* erreicht hat, und zwar so,
  - dass die Prüfung “wahr” ergibt, solange **i** noch nicht über dem Endwert ist
    (also für **alle** Werte, für die *befehle* wiederholt werden soll),
  - und das erste Mal “falsch” wird, wenn **i** den Endwert überschritten hat.

  Also z.B. **i <= n**, wenn ich von 1 bis einschließlich **n** zählen will,
  oder **i < n**, wenn ich von 1 bis einschließlich **n - 1** (ausschließlich **n**) zählen will,
  oder **i > 0**, wenn ich bis einschließlich 1 runterzählen will.
  Es ist aber auch zulässig, eine Prüfung zu haben, die nichts mit **i** zu tun hat.

Achtung: Kein **;** hinter den **( )** !!!

Beispiele:

```
for (i = 1; i <= n; ++i) {
  printf("%d ", i);
}
```

In die **( )** gehört die Zählvorschrift (von wo weg, wie weit, in welchen Schritten zählen) und in die **{ }** gehört, was für jeden Wert des Zählers einmal wiederholt werden soll.

Ausgabe: printf

printf("*Lückentext***",** *wert1***,** *wert2***, …);**

printf gibt den Lückentext aus und ersetzt dabei alle **%d** usw. der Reihe nach durch die angegebenen Werte.

Bisher kennen wir im Lückentext:
%d … **int**-Wert (“**d**ezimal”) (**%i** (“**i**nt”) ist genau dasselbe)
%s … Text (z.B. **argv[i]**) (“**s**tring”)
%f … **double**-Wert, in Komma-Darstellung (“**f**loat”)
%e … **double**-Wert, in wissenschaftlicher Darstellung (mit **e***nn*, d.h. “mal 10 hoch *nn*”) (“**e**xponent”)
%g … **double**-Wert, mit **%f** oder **%e**, je nachdem, was für den Wert besser passt
(es gibt noch mehr, z.B. **%c** … einzelner Buchstabe (“**c**har”)
oder **%x** … **int**-Wert als Hex**x**ahl)

Man kann noch Breite usw. angeben, z.B.:
%5d … **int**-Wert, 5 Stellen breit, rechtsbündig
%05d … dasselbe, aber vorne mit 0 aufgefüllt statt mit Leerzeichen
%-30s … Text, 30 Stellen breit, linksbündig statt rechtsbündig
%.15f … **double**, mit 15 Nachkommastellen (wenn man nichts angibt, sind es 6)

Weiters: **\n** im Lückentext bewirkt eine neue Zeile in der Ausgabe,
mit **\"** im Lückentext kann man ein **"** ausgegeben, und **%%** gibt ein **%** aus.

Ausgabe: cout

Achtung: cout ist C++, nicht C !

cout << *text_oder_wert* **<<** *text_oder_wert* **<<** … **<<** *text_oder_wert* **;**

text_oder_wert können Textkonstanten (z.B. **"Ergebnis: "**) sein oder Variablen (z.B. **sum** oder **argv[i]**) oder ganze Rechnungen oder **endl** (normalerweise am Schluss) für eine neue Zeile (wie das **\n** im **printf**). Sie werden alle der Reihe nach (ohne zusätzlichen Zwischenraum oder sonstige Trennzeichen) ausgegeben.

Die folgenden beiden Zeilen tun das gleiche:

```
printf("Der ggT von %d und %d ist %d.\n", a, b, result);
cout << "Der ggT von " << a << " und " << b << " ist " << result << "." << endl;
```

Bei Verwendung von **cout**:
#include <iostream> statt **#include <stdio.h>**
und gleich unter dem Include
using namespace std;
sonst kennt er **cout** trotz **iostream** nicht!

Man kann die Ausgabe in einem Programm wahlweise mit **printf** oder mit **cout** erledigen, aber "Entweder – Oder", man kann in einem Programm nicht beides mischen!

- **printf** (und einige andere Funktionen, die wir noch nicht kennengelernt haben) ist der Standard für die Ausgabe in der Programmiersprache C.
  Man kann es in C und in C++ verwenden.
- **cout** für die Ausgabe kam erst mit dem C++-Standard als Ersatz für **printf** neu dazu. Man kann es an sich nur in C++ verwenden, nicht in C, aber bei uns ist Dev-C++ so eingestellt, dass es unsere C-Programme sowieso wie C++-Programme behandelt, also können wir auch **cout** verwenden.

Ebenso kann man für die Eingabe entweder die C-Funktionen **scanf** usw. verwenden (werden wir wahrscheinlich nicht lernen) oder das **cin** von C++.

Für die, die es ganz genau wissen wollen:

- Wie ist das mit **iostream** und **iostream.h** ?
  Das ist ***nicht*** dasselbe!
  iostream.h ist alt (bevor C++ genormt wurde) und sollte nicht mehr verwendet werden, **iostream** ohne **.h** ist der neue C++-Standard
  (das gilt für alle C++-Includes: Die aktuell gültige Variante hat kein **.h** !).
  Aktuelle C++-Compiler akzeptieren nur mehr die neue Form.
  Sowohl **iostream.h** als auch **iostream** enthalten ein **cout**
  (und es funktioniert auch bei beiden im wesentlichen gleich),
  aber **iostream.h** kennt noch keine Namensräume (siehe nächster Punkt).
  Bei **iostream** braucht man daher das **using namespace std;** ,
  bei **iostream.h** darf man es nicht verwenden.
  Ähnliches gilt für die Verwendung von C-Includes in C++-Programmen nach neuem Standard: Zu jedem alten C-Include (z.B. **stdlib.h**) gibt es ein neues mit **c** vorn dran und ohne **.h** (also z.B. **cstdlib**), das dieselben Funktionen enthält, aber im Namesraum **std** statt ohne Namensraum.
- Was macht das **using namespace std;** ?
  Da C++-Programme immer größer und die vordefinierten Funktionen auch immer mehr wurden (viele tausend!) , wurde es immer schwieriger, kurze und eindeutige Namen für Funktionen, Konstanten usw. zu finden, die noch nicht vergeben waren.
  Der neue C++-Standard hat die Namen daher auf mehrere "Namensräume" aufgeteilt, wobei alle vordefinierten Namen im Namensraum "**std**" liegen
  (und jedes große Stück Software seinen eigenen Namensraum oder sogar mehrere bekommt).
  Damit der Compiler einen Namen findet, muss man ihm sagen, in welchem Namensraum er suchen muss. Das kann man mit **using namespace std;** einmal für das ganze Programm tun, oder man läßt das weg und
  schreibt dafür bei jeder Verwendung des Namens den Namensraum mit **::** davor
  (also überall **std::cout** statt nur **cout** und **std::endl** statt **endl**).

Arrays

Arrays (deutsch: "Feld") enthalten eine in der Deklaration festgelegte Anzahl von Werten desselben Typs. Früher musste die Anzahl ein fixer Wert (Zahl) sein, seit dem C-Standard C99 kann die Anzahl auch berechnet werden.

Beispiele für Array-Deklarationen:
`int count[10];` // ein Array "**`count`**" mit 10 **`int`**-Werten
`double zahl[argc];` // ein Array "**`zahl`**" mit **`argc`** vielen **`double`**'s
Auch das **`argv`** ist ein Array ("**`const char *argv[]`**" = Array von Zeigern auf konstante Buchstaben, Anzahl der Elemente vorab nicht bekannt).

Die Elemente sind durchnummeriert, die Nummer eines Elementes nennt man *"Index"*.
Der Zugriff auf einzelne Elemente erfolgt mit **`[`***index***`]`** .
Man kann Array-Elemente so wie normale Variablen lesen und in Rechnungen verwenden, und man kann so wie in normale Variablen etwas hineinspeichern.

Beispiele:
```
sum = sum + a[i];
a[0] = 1;
++(a[i]);
```

Achtung:
Der Index eines Arrays `a` mit `n` Elementen geht von 0 bis `n - 1` :
Das erste Element ist `a[0]`, das letzte `a[n-1]`,
und es gibt *kein* `n`-tes Element `a[n]` !!!

Achtung:
C prüft *nicht*, ob der Index gültig ist oder außerhalb des Arrays liegt
==> "zufälliges" Programm-Verhalten oder Absturz, wenn man "danebengreift"!

Man kann Arrays ***nicht***
- als Ganzes zuweisen,
- als Ganzes vergleichen,
- als Ganzes in Rechnungen verwenden,
- oder als Ganzes ein- oder ausgeben.

Alle diese Dinge muss man Element für Element machen, alle Elemente auf einmal geht nicht. Zu diesem Zweck verwendet man **`for`**-Schleifen: Der Befehl in der Schleife macht das Gewünschte für das Element **`i`**, und die Zählvariable **`i`** läuft über alle Indexwerte (beginnend bei 0):
```
for (i = 0; i < argc; ++i) {
  zahl[i] = 0;
}
```

Mehrdimensionale Arrays sind auch möglich:

```
double matrix[100][100];
for (zeile = 0; zeile < 100; ++zeile) {
  for (spalte = 0; spalte < 100; ++spalte) {
    if (zeile == spalte) {
      matrix[zeile][spalte] = 1;  /* 1 in der Haupt-Diagonale */
    } else {
      matrix[zeile][spalte] = 0;  /* und 0 überall sonst */
    }
  }
}
```

Solange man nichts zugewiesen hat, enthalten die Elemente eines Arrays (so wie normale Variablen) zufällige, nicht vorhersagbare Werte.
An diesem Beispiel sieht man auch gleich die Verwendung und Wirkung ineinander geschachtelter Schleifen:

- Für jeden einzelnen Durchlauf der äußeren Schleife wird die innere Schleife einmal komplett ausgeführt (d.h. die innere Schleife macht jedesmal alle Durchläufe).
  In diesem Beispiel:
  - Die äußere Schleife macht eine Zeile nach der anderen.
  - Für jede einzelne Zeile geht die innere Schleife einmal über alle Spalten in dieser Zeile, eine Spalte nach der anderen.
- Ineinander geschachtelte **`for`**-Schleifen müssen **verschiedene** Zählvariablen haben! Sie zählen ja auch verschiedene Dinge: Hier zählt die eine die Zeilen, die andere die Spalten. In unserem Balkendiagramm-Beispiel zählt die äußere Schleife die Zahlen durch, und die innere Schleife zählt für jede Zahl die Anzahl der #.