AIK Programmieren 1 Übung: Statische und globale Variablen

Klaus Kusche

1.) Zufallszahlen

Die einfachste (und jahrelang übliche) Implementierung von **`rand()`** in Linux
hat wie folgt funktioniert:

- Die **`rand`**-Funktion enthält eine *interne Variable* vom Typ **`int`**,
  die ihren *Wert zwischen zwei Aufrufen behält*.
- Diese Variable wird *beim Programmstart auf 1 initialisiert*
  (oder mittels **`srand`** gesetzt) und bei jedem Aufruf von **`rand`** neu berechnet:

  *Neuer Wert = alter Wert * 1103515245 + 12345* (der *Überlauf* wird *ignoriert*)

  (Microsoft hat dasselbe Prinzip, aber etwas andere Konstanten verwendet.)
- Dann berechnet man aus dieser Variable die Ergebnis-Zufallszahl, indem man
  ihren Wert mit Restrechnung (**`%`**) auf den gewünschen Zahlenbereich reduziert.

Anmerkung: Die erzeugten Zufallszahlen sind nicht besonders gut.
Heute kennt man *viel bessere* (aber auch viel aufwändigere) Verfahren für Zufallszahlen.

Aufgabe:

- Schreib eine *Funktion* **`myRand`** mit *zwei* **`int`**-*Parametern* **`from`** und **`to`**,
  die eine nach der oben beschriebenen Methode erzeugte ganze Zufallszahl **`z`**
  mit **`from`** kleinergleich **`z`** kleinergleich **`to`** als *Ergebnis* zurückliefert.
- Schreib dazu ein **`main`**, das mit dieser Funktion *30 Zahlen zwischen 1 und 6*
  "würfelt" und diese Zahlen sowie deren Mittelwert (als Kommazahl!) ausgibt.

Hinweise:

- Nimm **`unsigned int`** statt **`int`** für deine Variablen, Parameter und Returnwerte:
  Unsere Zufallszahlen sind immer positiv, und die interne Berechnung
  liefert sonst falsche (weniger gut zufällige) Zahlen.

Zusatzaufgabe:

Da die interne Statusvariable ja fix initialisiert ist, liefert die Funktion
bei jedem Programmlauf *dieselben* Zufallszahlen. Wir wollen unser **`myRand`** so umbauen,
dass es sich mittels **`time(NULL)`** *automatisch auf verschiedene Anfangswerte initialisiert*
(so, wie man das beim Standard-**`rand`** mittels **`srand`** macht).

Realisiere dazu folgende Idee:

- **`myRand`** bekommt eine *zweite "ständig lebende" interne Variable*, und zwar
  vom Typ **`bool`**, die anzeigt, ob die Funktion *schon jemals aufgerufen* wurde
  (auf **`false`** initialisieren!).
- Wenn die Variable **`false`** ist (also *beim ersten Aufruf*), wird **`time(NULL)`**
  aufgerufen, das Ergebnis als Startwert in der internen Zufalls-Variable gespeichert,
  und unsere **`bool`**-Variable auf **`true`** gesetzt.

2.) Binomialkoeffizient

Der Binomialkoeffizient "n über k" kommt in der Mathematik an vielen Stellen vor (siehe "Pascalsches Dreieck"). Er hat eine direkte Definition als n! / (k! * (n-k)!) (deren Berechnung aber schnell zu *Überläufen* führt) und eine *rekursive Definition*, die sogar ohne Multiplikation auskommt:

- Ist k > n, so ist das Ergebnis 0.
- Ist k == 0 oder k == n, so ist das Ergebnis 1.
- Sonst ist das Ergebnis "(n-1) über (k-1)" + "(n-1) über k".

Aufgabe:

- Schreib eine *Funktion mit zwei Parametern* **n** und **k**,
  die den Binomialkoeffizient nach oben beschriebener Methode *rekursiv* berechnet und als Returnwert zurückliefert.
- Schreib dazu ein **`main`**, das zwei Zahlen von der Befehlszeile einliest und deren Binomialkoeffizient berechnet und ausgibt.
- Ergänze das um eine *globale Variable*,
  in der mitgezählt wird, *wie oft die Funktion aufgerufen wird*
  (erhöhe die Variable in der Funktion bei jedem Aufruf um 1),
  und gib die Gesamt-Anzahl der Aufrufe danach im Hauptprogramm aus.

Hinweise:

- Nimm auch hier **`unsigned int`** statt **`int`** für deine Variablen, Parameter und Returnwerte: Die obenstehende Definition gilt nur für **n** und **k** größergleich 0, und mit **`unsigned int`** sparst du dir die Prüfungen auf kleiner 0.
- Der Wertebereich von **`unsigned int`** kann für Binomialkoeffizienten ab **n** gleich 35 zu klein werden.

Zusatzaufgabe:

Bei höheren Werten (z.B. "30 über 15") wächst die Anzahl der Aufrufe extrem, weil sehr viele Aufrufe *mehrfach* gemacht werden. Dem wollen wir mit folgender Idee abhelfen:

- Wir legen in der Funktion ein *zweidimensionales statisches* **`unsigned int`**-*Array* an. Der erste Index ist unser **n**, der zweite Index ist **k**.
  Eine Größe von fix 35 * 35 reicht (prüfe in deiner Funktion, ob **n** kleiner 35 ist!), denn für höhere Werte von **n** und **k** würde "**n** über **k**" ohnehin überlaufen.
- *Nach* dem Prüfen der Rand- und Sonderfälle, aber *vor den zwei rekursiven Aufrufen prüfen* wir das Element **`[n][k]`** des Arrays:
  - Ist es 0, so wurde der Binomialkoeffizient für dieses **n** und **k** noch *nie berechnet*. Wir berechnen ihn wie bisher mit 2 rekursiven Aufrufen, *speichern* das Ergebnis im Array, und geben es als Returnwert zurück.
  - Ist es nicht 0, wurde dieser Wert schon einmal berechnet:
    Wir geben den *Wert aus dem Array* sofort als Ergebnis zurück, *ohne* nochmals rekursive Aufrufe zu machen.
- *Liefert das verbesserte Programm dasselbe Ergebnis?*
  Um wieviel verbessert sich dadurch die Anzahl der Aufrufe?