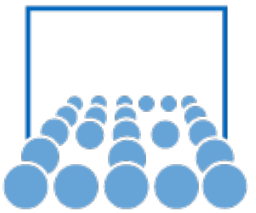
Kompaktkurs
Einführung in die Programmierung

1. Einführung

Stefan Zimmer

1.3.2010



Organisatorisches

- WWW-Seite der Vorlesung auf http://www5.in.tum.de/ (→ „Teaching“ → „Summer 10“)
- Es gibt pro Tag ein Übungsblatt mit Programmieraufgaben (die sind der wesentliche Bestandteil der Veranstaltung: auf's Selbermachen kommt es an!)
- Besprechung in Tutorübung (keine Abgabe/Korrektur)
- Schein durch Bestehen der Abschlussklausur (Freitag, 19.3., 15:15-16:30 Uhr, Hörsaal MW 2001)

Warum Programmieren lernen / C lernen?

- Rechner als Werkzeug, Mathematik zu verstehen und anzuwenden
- Oft: große Datenmengen erfordern effiziente Verarbeitung (z.B. Strömungsimulation für Wettervorhersage, Eigenwertberechnungen in Suchmaschinen, …)
- Notwendig:
  - Arbeitsweise des Rechners verstehen
  - Dieses Verständnis beim Aufschreiben des Rechenverfahrens ausnutzen können, damit sie auf der vorhandenen Hardware effizient ausgeführt werden kann
- Daher Programmieren in C – einer Sprache, in der man die Prinzipien des Rechners noch gut erkennen kann (im Gegensatz z.B. zu Matlab oder Python)

Ein einfaches Modell vom Rechner (1)

- Um Programme für unsern Rechner schreiben zu können, sollten wir eine grobe Vorstellung davon haben, wie das Ding arbeitet
- Folgendes simple Modell reicht für unsere Zwecke völlig aus. Es besteht aus
  - Dem Speicher, dargestellt durch ein Regal mit Fächern, von denen jedes einen Zettel mit einer Zahl, einem Buchstaben o. Ä. enthält.
    Die Fächer sind nummeriert.
  - Ein Rechenknecht mit Papier und Bleistift, der eine Dienstanweisung vor sich hat und nach der Daten aus dem Speicher holt, damit Operationen durchführt (Rechnungen, Vergleiche etc.) und das Ergebnis wieder im Speicher ablegt.

Ein einfaches Modell vom Rechner (2)

- Um die folgenden Beispiele lesbarer zu gestalten, werden wir statt Nummern für die Fächer Namen verwenden.
- Vorerst können wir uns vorstellen, dass es ein „Telefonbuch“ gibt, mit dem man zu einem Namen die Nummer nachschlagen kann.
- In ein paar Tagen werden wir sehen, dass hier in Wirklichkeit eine etwas kompliziertere Konstruktion notwendig ist.
- Der Rechenknecht ist außerordentlich stur: wenn z.B. in dem Fach, aus dem er etwas holen soll, nichts drin ist und er keine Anweisungen für diesen Fall hat, bleibt er einfach ewig davor stehen...

Unser Rechner rechnet

- Angenommen, wir hätten vier Fächer *a*, *b*, *c* und *d* mit Werten versehen und wollten Fach *e* mit dem Wert von $(a+b)\times(c+d)$ füllen.
- Das könnte so gehen:

  Hole den Inhalt von *a*
  Hole den Inhalt von *b*
  Addiere beide Werte
  Speichere das Ergebnis in einem Zwischenspeicher
  Hole den Inhalt von *c*
  Hole den Inhalt von *d*
  Addiere beide Werte
  Hole das erste Zwischenergebnis
  Multipliziere beide Zwischenergebnisse
  Speichere das Ergebnis in Fach *e*

Algorithmen

- Ein Algorithmus ist eine präzise, endliche Verarbeitungsvorschrift, die auf endlich vielen wohldefinierten Grundoperationen von jeweils endlicher Ausführungszeit aufbaut.
- Eine Funktionseinheit, die diese Arbeitsschritte ausführt, nennen wir eine Maschine
- Unsere Aufgabe ist nun, einen Algorithmus zu finden, den unsere Maschine (der Rechenknecht oder der Computer) ausführen kann
- Hilfreich dazu ist ein zweistufiges Vorgehen: erst den Algorithmus informell und unabhängig von einer speziellen Maschine zu entwickeln und ihn dann in eine maschinengerechte Form zu bringen

Beispiel Teilersumme

- Gesucht: Summe aller echten Teiler einer natürlichen Zahl $n \in \mathbb{N}$:

$$s(n) = \sum_{1 \leqslant k < n : k \mid n} k$$

- Erster Schritt: Beispiel rechnen!
  Z.B. $s(12) = 1+2+3+4+6 = 16$
- Nun genauer hinschauen: was haben wir im Detail gemacht?
- Für alle infrage kommenden Zahlen k prüfen, ob $k \mid 12$ (Rest bei der ganzzahligen Division $12 / k$ ansehen):
  Teiler sind 1,2,3,4 und 6, nicht aber 5,7,8,9,10,11
- Die Zahlen 7...11 haben wir vermutlich nicht wirklich angeschaut – aber vorerst ist uns die Effizienz egal

Beispiel Teilersumme (2)

- Nun die Summe berechnen – und zwar eine Addition nach der anderen: 1+2 = 3, 3+3=6, 6+4=10, 10+6=16
- Der Algorithmus wird schöner, wenn wir den „Sonderfall“ 1+2 = 3 in den Normalfall „addiere den gefundenen Teiler zur bisherigen Summe dazu“ integrieren – dazu fangen wir einfach mit einer Summe 0 an
- Es gibt keinen Grund, die Teiler zu speichern
- Damit sieht unser Algorithmus nun so aus:

  0 ⇨ *summe*

  Für alle k=1 ... n-1:

  Falls $k \mid n$:

  summe + k ⇨ *summe*

  Ergebnis: *summe* enthält nun den Wert $s(n)$

Beispiel Teilersumme (3)

- Ist Zählen („Für alle k=1 ... n-1“) eine Grundoperation?
- Für unseren Rechner eigentlich nicht, der will es lieber so haben:

  0 ⇨ *summe*

  1 ⇨ *k*

  ❶ Falls $k \mid n$:

  summe + *k* ⇨ *summe*

  k+*1* ⇨ *k*

  Falls k<n:

  Weiter bei ❶

  Ergebnis: *summe* enthält nun den Wert $s(n)$

Beispiel Teilersumme (4)

- Auch „Falls *k*|*n:*“ müssen wir unserem Rechenknecht noch näher erklären:

  Hole den Inhalt von Fach k
  Hole den Inhalt von Fach n
  Berechne den Rest der ganzzahligen Division n/k
  Wenn der Rest ungleich 0 ist, überspringe die
  Anweisung(en) „*summe* + *k* ⇨ *summe*“

- Und so weiter – aus der primitiven Summenformel würde, wenn man das zu Ende führte, schon ein langes (und unübersichtliches) Programm

- Zum Glück brauchen wir so nicht (mehr) zu programmieren – wir haben einen *Übersetzer*, der aus etwas in der Art des ersten Algorithmus etwas macht, das der Rechner ausführen kann

Programmiersprache

- Programme in menschenlesbarer Form aufschreiben!
- Ziel: sich statt um technische Details möglichst um wesentliche Dinge (Ablauf der Berechnungen, Korrektheit, Effizienz...) kümmern
- z.B. neben Grundoperationen auch Möglichkeiten, die Ablaufstruktur des Programms übersichtlich aufzuschreiben („Für alle k=1 ... n-1“)
- Übersetzer (*Compiler*) übersetzt in ein vom Rechner ausführbares Programm
- Ziel erreicht? Heutige Übersetzer erlauben eine ziemlich hohe Abstraktion von dem tatsächlichen Rechner - eine Grundvorstellung, wie der arbeitet, ist aber immer noch notwendig

Problem → Algorithmus → Programm

- Eigentliches Problem bei der Erstellung unserer Programme (die nicht umfangreich sind – große Projekte haben wieder ganz eigene Schwierigkeiten) ist der Schritt vom Problem zum Algorithmus
- Die meisten Folien dieses Kurses behandeln hingegen die Umsetzung des Algorithmus in der Programmiersprache C
- Das ist wichtig für das Bearbeiten der Programmieraufgaben: erst verstehen, was wie berechnet werden soll („Wie würde ich es mit Papier und Bleistift machen?“), dann erst um die Formulierung in C kümmern!

Was der Übersetzer so übersetzt (1)

- Zum Formulieren unser Algorithmen brauchen wir eine grobe Vorstellung, was wir in der Programmiersprache aufschreiben dürfen (und was dann vom Übersetzer in eine lange Abfolge elementarer Operationen übersetzt wird)
- Formeln mit den Grundrechenarten, Vergleichen etc. werden erlaubt sein: $(a+b)\times(c+d)$ ⇨ e oder $x \geq y$ (letzteres eine Formel mit Zahlen (?) als Variablen und einem Wahrheitswert als Ergebnis)
- Das ist doch viel besser als:

  Hole den Inhalt von *a,* hole den Inhalt von *b,* addiere beide Werte, speichere das Ergebnis in einem Zwischenspeicher, hole den Inhalt von *c,...*

Was der Übersetzer so übersetzt (2)

- Zur Steuerung des Programmablaufs gibt es Fallunterscheidungen und Schleifen
- Bei der *Fallunterscheidung* können Teile des Programms in Abhängigkeit einer Bedingung ausgeführt werden oder nicht: Falls $k|n$: *summe* + *k* ⇨ *summe*
- Auch möglich: anhand der Bedingung wird einer von zwei Programmteilen ausgeführt:

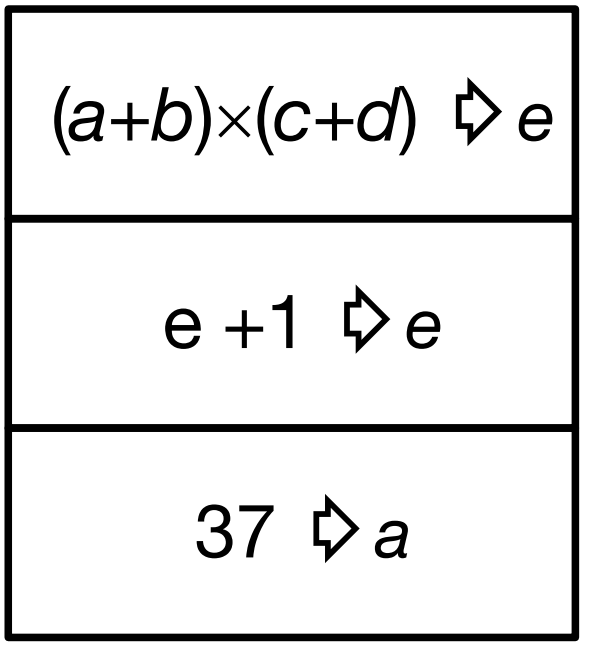
  Falls $x \geq 0$: *x* ⇨ *ergebnis*

  Sonst: *-x* ⇨ *ergebnis*

- *Schleifen* erlauben es, Programmteile zu wiederholen
  - eine vorher festgelegte Anzahl von Durchläufen („Für alle $k = 1 \ldots n - 1$“) oder
  - solange eine bestimmte Bedingung erfüllt ist („Solange *summe* < 100“)

Struktogramme

- Als graphische Repräsentation der Struktur des Programms werden wir *Struktogramme* malen.
- Zunächst malt man nur Kästen, einen für jede Aktion, und stapelt die aufeinander, um die Reihenfolge der Abarbeitung anzudeuten.



- Die Schleifen und Fallunterscheidungen sind (jeweils als Ganzes) auch Anweisungen, bekommen also auch Kästen, aber mit mehr Innenleben.
- Sie werden untergeordnete Anweisungen enthalten und Elemente, die deren Ausführung steuern.

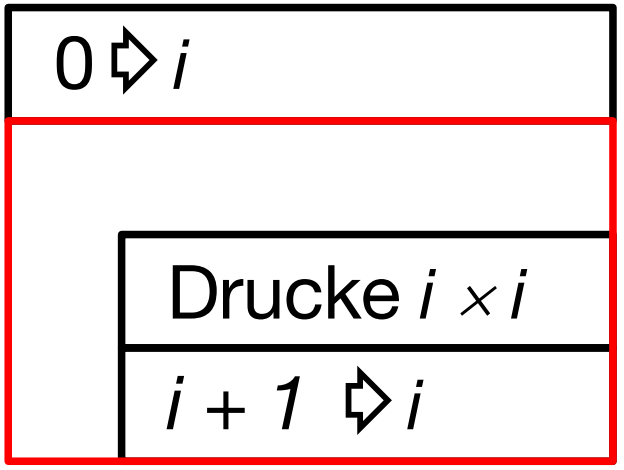
Wozu Struktogramme?

- Man kann auch während des Entwurfs ruhig eine komplizierte Aktion in Worten reinschreiben und später präzisieren.
- Ziel ist die strukturierte Zerlegung der komplizierten Aktion in viele einfache, was helfen soll, den Überblick zu behalten und ein gutes (korrektes, effizientes, leicht zu wartendes...) Programm abzuliefern.

- Auch bei unseren kleinen Übungsprogrammen gibt es schon einen Vorteil: beim Malen bleiben einige technische Probleme noch verborgen, was uns hilft, über die Logik des Programms nachzudenken.

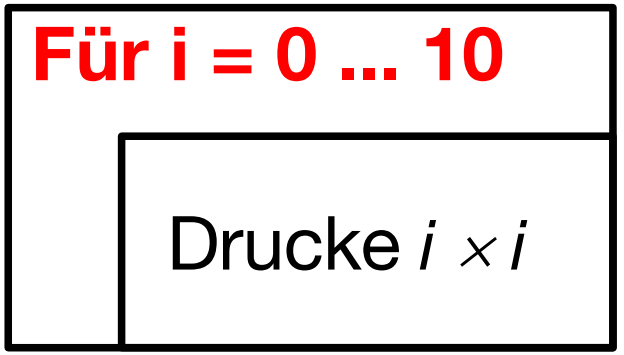
(Endlos-) Schleifen

- Eine Schleife wird durch einen Kasten (im Beispiel: der rote) repräsentiert, der
  - die Anweisungen enthält, die wiederholt werden sollen und
  - links und oben etwas Platz lässt
- Z.B. für ein Programm, das die Quadratahlen 0,1,4,9,16,... ausdruckt:



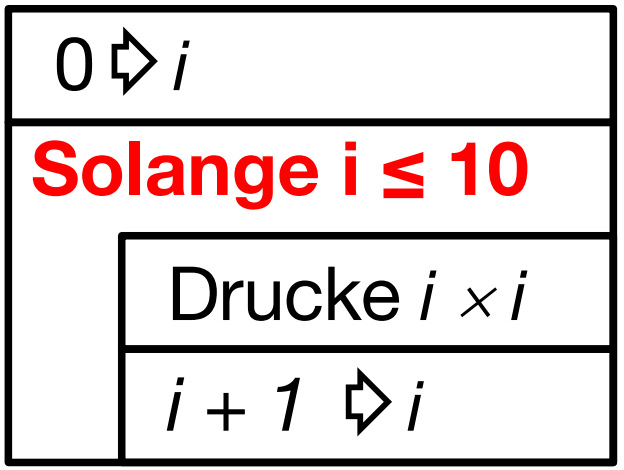
Zählschleifen

- Der nächste Fall ist, dass ich zu Beginn der Schleife weiß, wie oft ich sie ausführen möchte.
- Dann schreibe ich in den Bereich oberhalb der zu wiederholenden Anweisungen hinein:
  - wo mit Zählen begonnen werden soll,
  - wie weit gezählt werden soll
  - und (normalerweise) einen Namen für einen *Schleifenzähler* (damit wir in der Schleife sehen können, wo wir gerade sind)



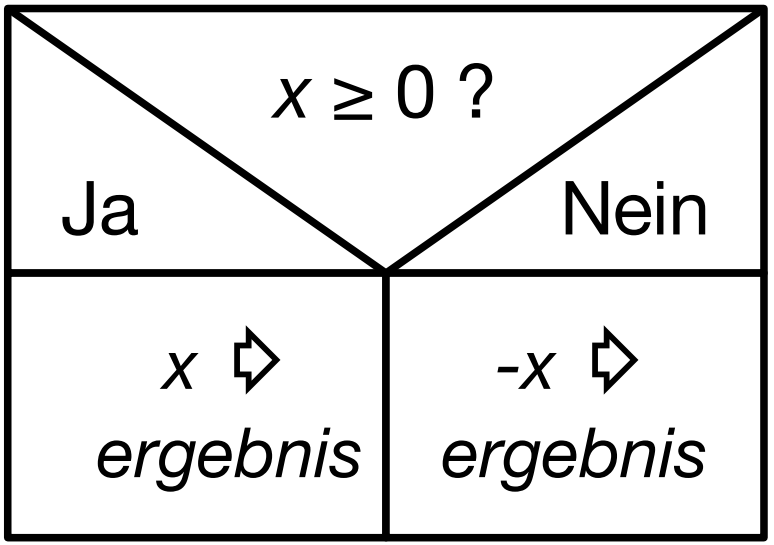
Mehr Schleifen

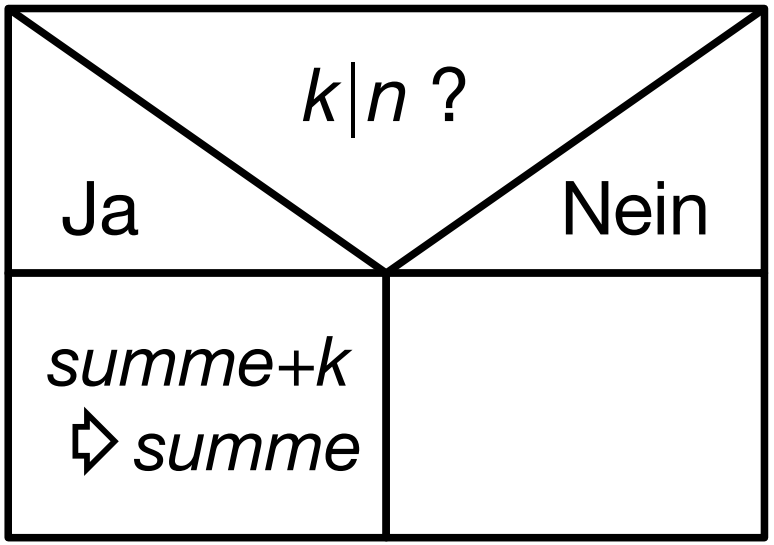
- Etwas allgemeiner ist der Fall, dass die Schleife wiederholt werden soll, solange ein bestimmtes Kriterium erfüllt ist
- Eine Zählschleife lässt sich leicht in diese Form bringen, im Beispiel von der vorigen Folie etwa so:



Fallunterscheidung

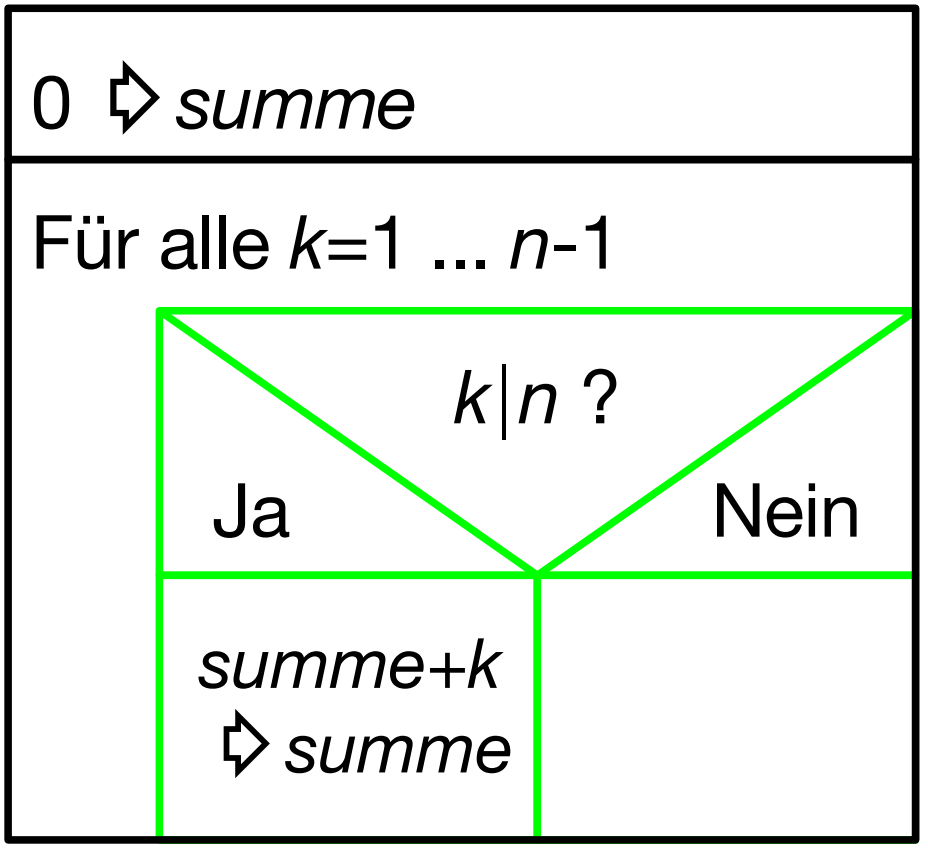
- Für die Fallunterscheidung gibt es nebeneinander zwei (ggf. Stapel von) Kästen und drüber ein Kasten, der das Kriterium angibt
- Falls nur in bei erfülltem Kriterium was zu tun ist, bleibt der rechte Kasten leer





Struktogramme schachteln

- Ein komplizierterer Kasten kann seinerseits Bestandteil eines Struktogramms sein, etwa eine Fallunterscheidung innerhalb einer Schleife
- Damit könnte das Struktogramm zur Teilersumme so aussehen:



Struktogramme malen!

- Auch wenn es etwas mühselig ist, Struktogramme zu malen (Die Kästen haben am Anfang nie die richtige Größe...): trotzdem tun!
- Wenn man seinen Algorithmus in ein Struktogramm gepackt hat, ist sie Umsetzung in ein Programm leicht
- Und die Chancen sind gut, dass es auch ein „schönes“ Programm gibt – Konstruktionen wie „Weiter bei ❶“ gibt es im Struktogramm mit gutem Grund nicht: sie führen zu notorisch unübersichtlichen und somit in aller Regel falschen Programmen!

Gliederung in Funktionen

- Wichtiges Prinzip beim Programmentwurf: Aufgabe in kleine Teilaufgaben zerteilen, Teilaufgaben (weitgehend) ohne Betrachtung deren „Innenlebens“ zusammensetzen – auch das unterstützt der Übersetzer
- Beispiel: für Test, ob ein $n \in \mathbb{N}$ vollkommene Zahl (gleich der Summe ihrer Teiler) ist, können wir die Berechnung der Teilersumme von vorhin wiederverwenden:

  Falls *Teilersumme*(*n*)=*n*:
  Ergebnis = ja
  sonst:
  Ergebnis = nein
- Hier noch wenig hilfreich, bei größeren Programmen aber wesentlich, um die Übersicht zu behalten!

Die Programmiersprache C

- Damit haben wir die wesentlichen Elemente zusammen, deren Realisierung in einer realen Programmiersprache – und die sich daraus ergebenden Konsequenzen – der Inhalt des Kurses sein werden
- Mann kann jetzt (oder später im Laufe des Kurses) noch mal zurückblättern: es sind nicht viele Konzepte, das ist doch beruhigend
- Die Programmiersprache, die wir lernen werden, heißt C – neben einigem C-spezifischen Stoff finden sich die meisten Dinge aber auch in den meisten anderen Programmiersprachen wieder, so dass die Wahl der Sprache, in der man das Programmieren lernt, gar nicht allzu wichtig ist

C vs. C++

- Bevor die Literaturhinweise kommen, ist eine technische Merkwürdigkeit zu klären: wir werden in Wirklichkeit nicht die Sprache C selber benutzen, sondern deren Weiterentwicklung C++
- Grund dafür ist hauptsächlich, dass man früher oder später C++ lernen sollte, daher ist dieser Kurs so angelegt, dass man dass anschließend gut im Selbststudium tun kann
- Bis dahin halten wir uns an folgender Regel fest: fast jedes C-Programm ist auch ein C++-Programm (das auch dasselbe tut...), auf Ausnahmen werde ich jeweils hinweisen. Von C++ interessiert uns vorerst nur der „C-Teil“, deshalb sage ich auch weiterhin C
- Also: wir benutzen C++, um C zu lernen. Einfach nichts dabei denken ☺

Literatur (1)

- Aus der merkwürdigen vorigen Folie folgt für den Erwerb eines Buches folgendes: es kann ein Buch über C sein oder eines über C++, wobei es in letzterem Fall die grundlegenden Konzepte aus C (im Gegensatz zu: objektorientierte Programmierung) ausgiebig behandeln sollte.
- Mein Lieblings-C-Buch ist nach wie vor das von den Erfindern von C:
  B. Kernighan, D. Ritchie: *The C Programming Language* / *Programmieren in C*
  Wichtig: zweite Auflage „ANSI-C“ (die erste Auflage beschreibt die mittlerweile verdient aus der Mode gekommene Urform von C)

Literatur (2)

- Ein dicker Wälzer, der sich auch für unsere Zwecke gut eignet:
  U. Kaiser, C. Kecher: *C/C++. Von den Grundlagen zur professionellen Programmierung*
- Nur zum Nachschlagen (da kein Lehrbuch), aber billig: das Heft *Die Programmiersprache C*, erhältlich beim LRZ:
  http://www.lrz-muenchen.de/services/schriften/verkauf/
- Online verfügbares Material gibt's auch jede Menge, z.B. hier:
  http://www.haw-hamburg.de/rzbt/dankert/c_tutor.html/

Material

- Für Software brauchen wir hier kein Geld auszugeben, notwendig sind eigentlich nur zwei Dinge:
  - Ein Texteditor (unter Windows z.B. Notepad oder Wordpad; nicht Word und Konsorten, die zum Erstellen von formatiertem Text gedacht sind, das benötigte Dateiformat ist „Nur Text“)
  - Ein Übersetzer, der aus unserem Text ein ausführbares Programm erzeugt (Details dazu später)
- Übersetzer, die für unsere Zwecke völlig ausreichen, gibt es kostenlos (in der Regel müssen sich nur Windows-Benutzer drum kümmern, bei den meisten Linux-Installationen und in der SUN-Halle gibt's schon einen).
  Drei von vielen Möglichkeiten:
  - Dev-C++: http://www.bloodshed.net/dev/,
  - Cygwin: http://www.cygwin.com/
  - MinGW: http://www.mingw.org/

Ausprobieren!

- Außer über den Weg vom Problem zum Struktogramm nachzudenken, soll man heute lernen, ein fertiges C-Programm (`prog0.cpp` von der WWW-Seite der Vorlesung) zu bearbeiten, übersetzen und auszuführen
- Unbedingt heute erledigen, auch wenn's wenig spannend aussieht/ist! Ab morgen gibt’s genug andere Dinge, um die wir uns kümmern müssen...
- Also
  - Programm runterladen
  - Mit dem Texteditor unserer Wahl öffnen (erstmal noch nichts ändern)
  - Übersetzen, laufen lassen; die Werte ein wenig ändern, speichern, übersetzen, laufen lassen,...

Das Beispielprogramm

Das Programm sieht ungefähr so aus (im Moment brauchen wir uns um die einzelnen Bestandteile noch keine Gedanken zu machen):

```
#include <iostream>
int main() {
  int a=37, b=41, c=-4, d=6, e;


  e = (a + b)*(c + d);


  std::cout << "a: " << a << ", b: " << b
            << ", c: " << c << ", d: " << d
            << ", e: " << e;

}
```