Universität Bielefeld - Fakultät für Physik

Skript zur Vorlesung

Computerphysik

Lukas Bogunovic & Edwin Laermann

Diese Ausfertigung ist vorläufig und erhebt nicht den Anspruch, vollständig und fehlerfrei zu sein.

Dieses Skript basiert auf Lukas Bogunovics Mitschrift der Vorlesung *Computerphysik* von Edwin Laermann, gehalten im Sommersemester 2006. Die vorliegende Version wurde von den Autoren überarbeitet, unter Mithilfe von Olaf Kaczmarek und den Teilnehmern der gleichen Lehrveranstaltung im Sommersemester 2007.
Dank gebührt ferner Herrn Prof. Jürgen Engels, der diese Vorlesung erstmalig im Sommersemester 2004 an der Universität Bielefeld angeboten hat.

Dozent: Prof. Dr. Edwin Laermann
edwin@physik.uni-bielefeld.de

Satz in LaTeX: Lukas Bogunovic
bogunovic@physik.uni-bielefeld.de

Version vom 13. Mai 2009.

Fakultät für Physik
Universität Bielefeld
Universitätsstraße 25
D-33615 Bielefeld
Germany

http://www.physik.uni-bielefeld.de/

Inhaltsverzeichnis

Appendix

A. Kleine Einführung in die Programmiersprache C

Zum Abschluss möchten wir uns einer (sehr) kleinen Einführung in die Programmiersprache C widmen, in der Sie in den Übungen Ihre Aufgaben lösen werden. Ihre Entstehungsgeschichte reicht bis in die 1970er Jahre zurück, als Ken Thompson und Dennis Ritchie sie für das neu aufkommende Betriebssystem 'UNIX' an den Bell Laboratories entwickelten. Alle grundlegenden Programme und sogar der UNIX-Kernel selber sind in C verfasst worden. Sie werden hier Ihre Programme unter 'Linux', einem UNIX-Derivat, entwickeln.
Vorweg zwei Literaturhinweise:

- Brian W. Kernighan & Dennis Ritchie: „*The C Programming Language*“, das Standardwerk
- RRZN: „*Die Programmiersprache C*“ im HRZ beim Dispatcher, eine Referenz

A.1. Der Weg zum ausführbaren Programm

In diesem kurzen Abschnitt diskutieren wir die drei grundlegenden Schritte, die nötig sind, um ein C-Programm zu schreiben und auszuführen.

1. **Editieren**: Schreiben Sie die Quelltextdatei (auch bezeichnet als Quellprogramm oder Quellcode) wie einen normalen Text mit einem Texteditor, z.B. Emacs. Dabei notieren Sie Schritt für Schritt die C-Anweisungen untereinander, in der Reihenfolge, wie sie ausgeführt werden sollen. Sie öffnen eine Datei 'test.c' in Emacs mit dem Linux-Kommando

   ```
   emacs test.c
   ```

   Nachdem Sie den Quellcode verfasst haben, können Sie ihn mit einem Klick auf das Diskettensymbol abspeichern.

2. **Kompilieren/Linken** mit gcc: Dieser Schritt ist nötig, um das von Ihnen verfasste Programm in für den Computer verständlichen Maschinencode zu übersetzen. Der Prozessor kann die Anweisungen einer Hochsprache wie C nicht direkt verstehen.
   Sie starten den Compiler mit dem Linux-Kommando

   ```
   gcc -Wall -o test test.c -lm
   ```

   Das Argument `-o` erzeugt den Programmnamen 'test' , `-Wall` startet den so genannten 'verbose mode' (zeigt sämtliche aufgetretenen Probleme und nicht nur die wichtigsten) und `-lm` linkt Mathebibliotheken, damit das Programm auch für Computer komplexe Funktionen wie z.B. Sinus oder die Wurzeloperation ausführen kann. Spätestens hier macht Sie der Compiler auch auf eventuelle Programmierfehler aufmerksam, indem er die Zeilennummer angibt und einen Kommentar mit einem Hinweis auf mögliche Fehlerquellen. Beachten Sie, dass der Programmname 'test' und der Dateiname des Quellcodes 'test.c' nicht unbedingt identisch sein müssen, sondern frei wählbar sind.

3. **Ausführen:** Nun ist das Programm bereit, um ausgeführt zu werden. Sie starten es, indem Sie einfach den Programmnamen eingeben und mit der Enter-Taste bestätigen:

   ```
   test
   ```

A.2. Hallo Welt!

Traditionell befasst sich jeder, der eine neue Programmiersprache erlernt, zunächst mit einem sehr einfachen Programm, welches nur den Sinn hat, den Text „Hallo Welt!“ auf dem Bildschirm auszugeben. Hierdurch erhält man einen ersten Eindruck von der verwendeten Syntax. Das so genannte Hallo-Welt-Programm sieht unter C folgendermaßen aus:

```
001 #include <stdio.h>
002
003 int main(void)
004 {
005     printf("Hallo Welt!\n");
006     return 0;
007 }
```

In Zeile 001 binden wir zunächst die Headerdatei `stdio.h` (Standard Input/Output) ein, damit das Programm über die Möglichkeit verfügt, Daten auszugeben und einzulesen. Darauf beginnt die Funktion 'main'. Eine Funktion ist eine funktionelle Untereinheit eines Programms. Mehr dazu gleich.
In Zeile 005 steht nun die eigentliche Ausgabeanweisung. Dabei sorgt `\n` dafür, dass ein Zeilenwechsel stattfindet. Das darauf folgende Kommando sendet an das Betriebssystem zurück, dass das Programm ordnungsgemäß beendet wurde. Mit einem von 0 verschiedenen Wert im Argument von 'return' würde man üblicherweise das System auf einen Fehler aufmerksam machen.

A.3. Wichtige Elemente eines C-Programms

- **Funktionen**
  Eine Funktion ist eine funktionelle Untereinheit eines C-Programms. Es empfiehlt sich, bestimmte Funktionseinheiten in Funktionen zu gruppieren. Jedes Programm benötigt die Hauptfunktion 'main'. Ansonsten sind die Namen der Funktionen beliebig, solange sie keine reservierten Zeichen oder Operatoren enthalten. Die Syntax der Definition einer Funktion lautet wie folgt:

```
Funktionstyp Funktionsname (Parameterdeklaration)
  {
     Anweisungen
  }
```

  Verdeutlichen wir dies an einem

  Beispiel:

```
float zylinder_oberflaeche(float h, float r)
  {
    float o;
    o=2.0*3.141*r*(h+r);
    return(o);
  }
```

  Diese Funktion trägt den Namen '`zylinder_oberflaeche`' und liefert einen Wert des Typs `float` (nämlich die Zylinderoberfläche) zurück. Dafür benötigt sie die Werte `h` und `r` (ebenfalls `float`, nämlich Höhe und Radius), welche ihr beim Start übergeben werden.
  An einer beliebigen Stelle unseres Programmes können wir nun durch Eingabe eines einzigen Befehls die Funktion aufrufen und damit die Oberfläche eines Zylinders berechnen:

```
a=zylinder_oberflaeche(h,r);
```

- **Variablen**
  Dieses Objekt repräsentiert eine Speicherstelle, deren Inhalt während der Lebensdauer der Variablen jederzeit verändert werden kann. Die Art des Inhaltes einer Variablen wird durch den ihr zugeordneten Datentyp festgelegt.
  Variablen müssen vor der ersten Verwendung deklariert werden, d.h. ihr Name und ihr Datentyp müssen dem Programm bekannt gegeben werden:

  ```
  int n=0;
  ```

  Damit wurde die Variable mit dem Bezeichner `n` dem Programm bekannt gemacht. Dabei können Bezeichner auch aus einem Wort bestehen, solange keine reservierten Zeichen enthalten sind. Die Variable `n` wird hier im Folgenden Daten vom Typ `int` enthalten. Es empfiehlt sich oft, die Variable gleich bei der Definition auch mit einem Startwert zu belegen.
  Im Programm kann der Variablen jederzeit ein neuer Wert zugeordnet werden:

  `n=23;` oder z.B. auch `n++;`

- **Konstanten**
  Möchte man einen Wert im Folgenden nicht mehr verändern, kann man ihn auch als Konstante definieren. Bei der Definition muss der Wert und der Datentyp gleich festgelegt werden:

  ```
  const int b=23;
  ```

  Konstanten sind grundsätzlich schreibgeschützt.

- **Datentypen**
  C kennt eine Vielzahl an Datentypen. Sie legen fest, welche Art von Daten eine Variable aufnehmen kann. Beispiele für Datentypen sind: `float` und `char`.

- **Anweisungen**
  Anweisungen sind einfache Befehle, die der Rechner ausführen soll, beispielsweise `i=1;`.
  Hinter jeder Anweisung steht ein Semikolon, um anzuzeigen, dass die Anweisung beendet ist.

- **Header-Dateien / Bibliotheken**
  Header-Dateien bzw. Bibliotheken enthalten bereits vordefinierte Funktionen. Durch Einbinden einer solchen Datei, wie z.B. `stdio.h` oder `math.h`, können die dort definierten Funktionen dem eigenen Programm hinzugefügt werden (z.B. `printf()` oder `sin(x)`). Man bindet eine Header-Datei mit dem Befehl `#include <Datei>` bzw. `#include "Datei"` ein, wobei bei Verwendung von Anführungszeichen die Suche nach der Datei üblicherweise in dem Verzeichnis beginnt, in dem sich auch die Quelldatei befindet.

A.4. Datentypen

A.4.1. Variablen- und Konstantentypen

- `int`: (Integer)
  Hierbei handelt es sich um ganzzahlige, positive oder negative Zahlen (Länge: 4 Byte bzw. 8 Byte bei `long int`). Außerdem benutzt man Integer, um logische Werte (true und false) in Form von 1 oder 0 zu handhaben.

- `float` (single precision):
  Eine so genannte Fließkommazahl (floating point number) mit einer Länge von 4 Byte, mit der normalerweise Zahlen gespeichert werden, die keine Integer sind (z.B. $-4 \cdot 10^{-5}$ oder 3.141.) Es können aber auch Integer gespeichert werden, wie z.B. $30000 = 3 \cdot 10^4 = 3.0e04$.

- `double` (precision):
  Eine Fließkommazahl mit einer doppelten Länge von 8 Byte.
- `char`: (Character)
  Dieser Datentyp enthält normale Zeichen (etwa `'a'` oder `'.'`), aber auch Sonderzeichen wie z.B. Zeilenumbrüche `'\n'`. Für den Computer ist ein `char` tatsächlich ein Integer mit einer Länge von 1 Byte. Man kann mit `char` Variablen also prinzipiell auch rechnen, sollte dabei aber den eingeschränkten Wertebereich beachten.
- `void`:
  Hierbei handelt es sich um einen besonderen Typ, welcher keine Werte verwalten kann und welcher keine Operationen zulässt. Man kann ihn mit der leeren Menge vergleichen. Wir benötigen ihn aber dennoch, z.B. als Rückgabedatentyp bestimmter Funktionen, die keinen Wert zurück liefern, sondern nur der Form halber mit einer entsprechenden Anweisung ausgestattet sein müssen.

Es ist sinnvoll, den Datentyp einer Variable nicht pauschal zu wählen (z.B. immer `double`, auch wenn man nur einen logischen Wert (1 oder 0) verarbeiten möchte), sondern ihn stets den aktuellen Anforderungen anzupassen. Eine `int` Variable benötigt z.B. nur halb so viel Speicher wie eine `double` Variable, selbst wenn darin nur eine Integer-Zahl gespeichert ist.
Es gibt noch eine Vielzahl weiterer Typen, z.B. lassen sich für `int` nochmals Unterteilungen vornehmen. Hier sei auf die angegebene Literatur verwiesen.
Es besteht auch die Möglichkeit, Typen ineinander umzuwandeln (type casting).

A.4.2. Typumwandlung (Casting)

Mit Hilfe von so genanntem 'Casting' lassen sich Datentypen ineinander konvertieren. Man schreibt allgemein: `x=(TYPNAME) i;` um einen Wert aus der Variable `i` in den angegebenen Datentyp zu konvertieren und in `x` abzulegen.

Beispiel:

```
double x;
int i=5;
x=(double) i;
```

Wir definieren zunächst die Variable `x` als `double` und die Variable `i` als `int`. Letzterer weisen wir gleich den Wert 5 zu. Wir casten nun `i` nach `x`, also `int` nach `double`. Dabei ist zu beachten, dass z.B. ein Casting in die andere Richtung, also von `double` nach `int` Probleme machen kann, weil der Datentyp `int` nicht alle möglichen Inhalte von `double` aufnehmen kann.

A.4.3. Felder

Bislang haben wir nur skalare Variablen bzw. Datentypen diskutiert. Es sind jedoch auch Vektoren und sogar Matrizen denkbar.

Vektoren
Wir können in C Variablen als n-dimensionale Vektoren verwenden. Sie werden folgendermaßen definiert:

```
int vek[n];
```

Die Elemente des Vektors können unabhängig voneinander verändert und benutzt werden. Das *erste* Element steuert man mit

```
vek[0]
```

an und die übrigen analog (das letzte Element ist also `vek[n-1]`). Das Zuweisen eines Wertes erfolgt wie bei einer normalen Variable, also z.B.

```
vek[3] = 1000;
```

Zeichenketten
Die Verwaltung von Zeichenketten erfolgt im Prinzip analog. Wir definieren eine n-stellige Zeichenkette mit

```
char string[n] = "Hallo!";
```

und weisen ihr dabei gleich die Buchstabenfolge 'Hallo!' zu. Zu beachten ist, dass eine solche Zuweisung in C nur bei der Deklaration der Zeichenkette möglich ist. Soll die Zeichenkette im weiteren Verlauf des Programms verändert werden, so muss sie elementweise (wie bei Vektoren) bearbeitet oder mit Hilfe einer der vordefinierten Funktionen aus `string.h` manipuliert werden.

Matrizen
Wir können auch n×m Matrizen definieren, indem wir dem bereits erläuterten Vektor zwei Nummern für die Komponenten zuweisen. Beispiel:

```
int mat[n][m];
```

Die Zuweisung von Werten ist auch hier für jede Komponente unabhängig möglich und erfolgt analog. Beispiel:

```
mat[7][15] = 10;
```

A.4.4. Zeiger (Pointer)

Für jeden Typ T (z.B. einen der obigen) kann man einen *Zeigertyp* „Zeiger auf T“ erzeugen. Der Wert eines Zeigers bzw. Pointers ist die (Speicher-)Adresse eines Objektes (in unserem Fall einer unserer Variablen). Ein Zeiger *„zeigt“* also auf das entsprechende Objekt.

- Der Adressoperator `&` liefert die Adresse eines Objektes.
- Der Dereferenzierungsoperator `*`, angewendet auf einen Zeiger, liefert das Objekt, welches an der Adresse abgelegt ist.

Beispiel:

```
int *p;
int i;
p=&i;
*p=5;
```

Zeiger sind von besonderer Wichtigkeit, wenn es darum geht, Variablen an Funktionen zu übergeben, damit diese sie weiterverarbeiten können.

A.4.5. Initialisierung & Konstanten

Variablen können bei der Deklaration initialisiert werden, z.B.:

```
double x=1.5;
```

Es besteht zudem die Möglichkeit, Konstanten zu definieren:

```
const double c=4.5;
```

Diese dürfen später im Programm nicht mehr Ziel einer Zuweisung sein (schreibgeschützt).

A.5. Operatoren

A.5.1. Rechenoperatoren

Ohne Weiteres stehen u.A. die folgenden Rechenoperatoren zur Verfügung. Weitere Rechenfunktionen, wie z.B. das Berechnen der Quadratwurzel, stehen nur nach der Einbindung der entsprechenden Header-Datei zur Verfügung (im Fall der Wurzel `sqrt()` ist dies z.B. `math.h`).
Die Ausdrücke in Klammern können Sie direkt in Ihrem Programm verwenden:

- Addition (`+`)
- Subtraktion (`-`)
- Multiplikation (`*`)
- Division (`/`)
  Hier sollten Sie beachten, dass die Integer-Division trunkiert, d.h. das Ergebnis ist wieder ein Integer, der durch das Verwerfen möglicher Nachkommastellen erzeugt wird.
- Modulooperation (`%`)
  `a%b` liefert den Rest der Division von `a` durch `b`. Dazu müssen `a` und `b` jedoch vom Typ `int` sein.
- `x++` $\Leftrightarrow$ `x=x+1`
- `x--` $\Leftrightarrow$ `x=x-1`
- `x+=1.5` $\Leftrightarrow$ `x=x+1.5` (analog: `-=`, `*=` und `/=`)

Für Operatoren gilt auch in C eine gewisse Prioritätsfolge. So hat z.B. die Multiplikation Priorität gegenüber der Addition und Klammern haben eine nochmals höhere Priorität.

A.5.2. Vergleichsoperatoren (relationale Ausdrücke)

Unter C stehen u.A. die folgenden relationalen Ausdrücke sofort zur Verfügung:

- `x<y` 'x ist kleiner als y'
- `x>y` 'x ist größer als y'
- `x<=y` 'x ist kleiner oder gleich y', analog: 'größer oder gleich' (`>=`)
- `x==y` 'x ist gleich y' [1]

Beispiel:

```
i=1;
j=2;
k=i>j;
```

k hat hier am Ende den Wert 0.
Es stehen weiterhin logische Operatoren zur Verfügung, wie z.B.:

- `&&` und
- `||` oder
- `!` nicht

[1] Das einfache Gleichheitszeichen wird bereits für die Zuweisung eines Wertes für eine Variable benutzt und darf hiermit nicht verwechselt werden.

A.6. Blöcke

Ein Block besteht aus Anweisungen, die durch geschweifte Klammern eingeschlossen sind. Darin können stehen:

```
{
   Deklarationen
   Anweisungen
}
```

Im Programm werden Blöcke wie eine einzige Anweisung behandelt. Dies kann z.B. nötig sein, wenn in einer `if`-Abfrage mehrere Aktionen bei Erfüllung der Bedingung ausgeführt werden sollen. Mehr dazu später.

A.7. Anweisungen

A.7.1. Schleifen (Loops)

Schleifen sind Objekte, welche Anwendung finden, wenn eine Operation mehrmals hintereinander ausgeführt werden soll. Dabei ist es möglich, in jedem Durchlauf bestimmte Parameter automatisch zu ändern. Anwendung finden Schleifen z.B. beim Berechnen einer Summe wie

$$a = \sum_{i=1}^{10} i^2 \ .$$

Dies kann mit einer `while`-Schleife folgendermaßen formuliert werden:

```
int a=0;
int i=1;
while(i<11)
   {
      a=a+i*i;
      i++;
   }
```

Solange der Wert von `i` kleiner als 11 ist, wird die Schleife immer wieder ausgeführt, bis `i` schließlich zu groß ist. Fehler in der Programmierung können hier zu ungewollten „Endlos-Schleifen“ führen, in diesem Fall z.B. wenn die Anweisung `i++;` vergessen wird.
Es geht auch umgekehrt:

```
int a=0;
int i=1;
do
   {
      a=a+i*i;
      i++;
   }
while(i<11)
```

Die `for`-Schleife ist die eleganteste und am meisten verwendete Schleifenstruktur. Die Summation hätte hier die folgende Gestalt:

```
int a=0;
int i;
for(i=1;i<11,i++)
   {
      a=a+i*i;
   }
```

A.7.2. Fallunterscheidungen

Die `if`-Abfrage
Mit Hilfe einer solchen Auswahlanweisung kann der Rechner abhängig von gegebenen Bedingungen eine Entscheidung treffen, wie weiter zu verfahren ist. Beispiel:

```
if(i<j)
   {
      Anweisungen
   }
```

Der Block wird ausgeführt, wenn `i` kleiner als `j` ist. Ansonsten wird er ignoriert. Dann gibt es noch:

```
if(i<j)
   m=i;
else
   m=j;
```

Hier haben wir die geschweiften Klammern weggelassen, da wir nur *eine* Anweisung ausführen möchten. Falls `i<j` gilt, wird die Anweisung `m=i;` ausgeführt und ansonsten `m=j;`. Es wird also das Minimum von `i` und `j` in `m` gespeichert.

Die `switch`-Anweisung
Falls mehrere mögliche Fälle untersucht werden sollen und je nach Fall unterschiedliche Aktionen durchgeführt werden sollen, eignet sich auch die `switch`-Anweisung:

```
switch(i)
   {
      case 0: m=5;
              break;
      case 5: m=0;
              break;
      default: m=-1;
               break;
   }
```

Die Abfrage betrachtet die `int` Variable `i`. Die zu betrachtenden Fälle (konstante Integerwerte) werden mit `case` gelistet und dahinter wird die jeweilige Aktion definiert, welche ausgeführt werden soll, wenn der Fall eingetreten ist. Falls keiner der Fälle eintritt, wird die Anweisung nach `default` angewendet. Der erste `break;`-Befehl, der ausgeführt wird, beendet die `switch`-Anweisung.

A.8. Parameterübergabe

Übergabe von Skalaren
Damit eine Funktion ausgeführt werden kann, benötigt sie oft Werte, die ihr beim Start übergeben werden müssen (vgl. die Funktion oben, welche die Zylinderoberfläche berechnet). Es gibt in C zwei Möglichkeiten dies zu tun:

- *call by value*: Dies bedeutet, dass die Werte der Parameter als 'Kopien' übergeben werden. Die Funktion kann daher diese Werte nur lesen, aber nicht ändern.
- *call by reference*: Es werden die Adressen der Parameter als Zeiger an die Funktion übergeben. Die Funktion kann damit auf die entsprechenden Speicherbereiche direkt zugreifen und die Variablen so ändern.

Ein **Beispiel:**

```
void chaab(int *x1, int *x2)
   {
      int temp;
      temp=*x1;
      *x1=*x2;
      *x2=temp;
   }
```

Wenn wir jetzt die Funktion aufrufen

```
int a=1, b=2;
chaab(&a, &b);
```

hat danach `a` den Wert 2 und `b` den Wert 1.

Übergabe von Feldern
Für Felder lässt sich nur das *call by reference* Verfahren benutzen. Dabei ist zu beachten, dass der Name eines Feldes zugleich der Zeiger auf das erste Element des Feldes ist. Die Deklaration einer Funktion, die als Argument ein Feld erhält, kann auf zwei äquivalente Art und Weisen erfolgen:

`int quad(int a[]);` oder `int quad(int *a);`

Die erste Variante wird oft vorgezogen, um deutlich zu machen, dass `a` tatsächlich ein Feld darstellt (vgl. das vorige Beispiel). Im Hauptprogramm kann man die Funktion dann wie folgt benutzen:

```
int a[10];
int b;
b=quad(a);
b=quad(&a[0]);
```

Beide Aufrufe der Funktion sind hier gleichwertig, denn bei der Übergabe von Feldern wird nur die Adresse (Zeiger) vom ersten Element übergeben.
Bei mehrdimensionalen Feldern läuft die Parameterübergabe ähnlich, z.B.:

```
int quad2(int a[][10])
```

Jedoch muss die Länge des Feldes, welches die Funktion als Parameter erhalten soll, bezüglich jeder weiteren Dimension (in diesem Fall der zweiten) *bei der Definition der Funktion fest vorgegeben* werden. Dies hängt damit zusammen, dass mehrdimensionale Felder intern tatsächlich als eindimensionale Felder gespeichert werden.

A.9. Dateneingabe und -ausgabe

Datenausgabe auf dem Bildschirm
Wir haben das `printf`-Kommando bereits im Rahmen des 'Hallo Welt'-Programmes kennen gelernt:

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

Der auszugebende Text wird in doppelte Anführungszeichen gesetzt und so als eine String-Konstante gekennzeichnet. Formal ist ein String ein `char`-Feld, in dem das Ende der Zeichenkette durch den so genannten 'null character' `'\0'` markiert ist. Die wichtigsten Steuerzeichen sind `'\n'` und `'\t'` (einfache Anführungszeichen werden für `char`-Konstanten verwendet). Mit `'\n'` erhält man einen Zeilenumbruch und mit `'\t'` einen Tabulatorvorschub.

Mit `printf` lassen sich aber auch Werte von Variablen ausgeben. Dies geschieht über folgende Syntax:

```
printf("Text %DATENTYP1 Text %DATENTYP2", a, b);
```

An die Stelle, an der der Wert der Variablen `a` bzw. `b` erscheinen soll, wird ein Platzhalter gesetzt, welcher den auszugebenden Datentyp charakterisiert. Anschließend werden hinter dem abschließenden Anführungszeichen alle Variablen in der richtigen Reihenfolge aufgezählt. Die folgende Liste gibt einen Überblick über die häufigsten Datentypen bei der Ausgabe mit `printf` (und beim Einlesen mit `scanf`, s. unten):

Kürzel	Datentyp	Beschreibung
d,i	`int`	Integer-Zahl
c	`char`	einzelnes Zeichen
s	`char`	String (s. oben)
f	`float`	reelle Zahl
lf	`double`	dito
e	`float`	reelle Zahl, jedoch in wissenschaftlicher Darstellung
le	`double`	dito

Dateneingabe über die Tastatur
Mit Hilfe der Anweisung `scanf` können Eingaben über die Tastatur eingelesen werden. Die Funktion ist zu `printf` recht analog. Allgemein:

```
scanf("%DATENTYP", &a);
```

Hier wird eine Tastatureingabe in der Variablen `a` abgelegt. Bei `scanf` muss jedoch ein Zeiger auf die Zielvariable übergeben werden und nicht die Variable selbst.

Benutzung von Dateien
Mit C kann auch in Dateien geschrieben bzw. aus Dateien gelesen werden. Der Zugriff darauf erfolgt über Dateizeiger (Filepointer). Ein Dateizeiger ist vom Typ `FILE *`. Er wird deklariert über:

```
FILE *datei_zeiger;
```

Ist der Zeiger nun dem System bekannt, kann ihm eine Datei zugewiesen werden. Dies erfolgt über die Funktion `fopen`:

```
datei_zeiger=fopen("dateiname", "Mode");
```

Mit der Zeichenfolge `Mode` wird die Zugriffsart festgelegt. Wir werden hier fast nur `"r"` für 'read' nutzen. Dies dient dazu, eine existierende Datei zum Lesen zu öffnen. Für das Programm ist sie dabei immer schreibgeschützt. Hingegen verwendet man `"w"` (für 'write'), um die Datei zum Schreiben zu öffnen. Dabei wird ihr Inhalt gelöscht, wenn sie bereits vorhanden ist! `"a"` steht für 'append' und dient dazu, die Ausgabe an eine bestehende Datei anzuhängen.

Die Ein- und Ausgabebefehle bei Verwendung von Dateien sind ähnlich, z.B.:

```
printf("%d", i);      wird zu      fprintf(datei_zeiger, "%d", i);

scanf("%d", &i);      wird zu      fscanf(datei_zeiger, "%d", &i);
```

Wenn der Lese- bzw. Schreibvorgang abgeschlossen ist, sollten geöffnete Dateien wieder geschlossen werden. Die Anweisung hierfür lautet:

```
fclose(datei_zeiger);
```

Beispiel:
Die Datei `"test.dat"` habe folgendes Aussehen

$$\begin{array}{cc} 1.0 & 2.0 \\ 2.0 & 3.0 \\ 4.0 & 5.5 \\ \vdots & \vdots \end{array}$$

und wir möchten eine Funktion entwickeln, welche die Daten in zwei Felder einliest und die Anzahl der eingelesenen Datenpaare zurückgibt. Wir nennen diese Funktion 'readin':

```
001 int readin(char *fname, double x[], double y[])
002    {
003       FILE *fin;
004       int i=0, ndata;
005       fin=fopen(fname, "r");
006       do
007          {
008             if(fscanf(fin, "%le %le\n", &x[i], &y[i]) == 2)
009             i++;
010          }
011       while(!feof(fin));
012       // feof entscheidet, ob der Dateizeiger am Ende der Datei steht
013       ndata=i;
014       fclose(fin);
015       return(ndata);
016    }
```

Bei C (bzw. eigentlich C++) zeigt `"//"` den Beginn eines Kommentars an, der bis zum Ende der jeweiligen Zeile reicht (andere Möglichkeit: `/* Kommentar */`). Kommentare werden vom Compiler ignoriert. Die Funktion `fscanf` hat als Rückgabewert die Anzahl der erfolgreich eingelesenen Objekte. Man kann nun die Funktion `readin` im Hauptprogramm wie folgt nutzen:

```
n=readin("test.dat", x, y);
```

Für die Verwendung aller hier vorgestellten Funktionen zur Ein- und Ausgabe muss die Header-Datei `stdio.h` eingebunden sein.

B. Literaturhinweise