# Modellierung und Programmierung

Volker John

Wintersemester 2007/08

# Inhaltsverzeichnis

**I Einführung in die Programmierung** 4

**1 Das Betriebssystem Linux** 5

**2 Algorithmen** 9

**3 Einführung in MATLAB** 12
3.1 Allgemeines . . . . . 12
3.2 Bemerkungen zu Vektoren und Matrizen . . . . . 13
3.3 Matrizen . . . . . 14
3.4 Wichtige Funktionen in MATLAB . . . . . 15
3.5 Ablaufkontrolle . . . . . 16
3.6 Graphik . . . . . 17

**II Mathematische Modellierung** 18

**4 Mathematische Modelle** 19

**5 Grundprinzipien der Mathematischen Modellierung** 21
5.1 Modellierungszyklus . . . . . 21
5.2 Dimensionslose Variablen und Skalierung . . . . . 22
5.3 Sensitivitätsanalyse . . . . . 25
5.4 Modellvereinfachungen . . . . . 26

**6 Modellierung von Wachstumsprozessen** 28
6.1 Ein einfaches Modell . . . . . 28
6.1.1 Ein Kontinuumsmodell . . . . . 28
6.1.2 Ein diskretes Modell . . . . . 29
6.1.3 Numerische Verfahren zur Lösung des Kontinuumsmodells . . 31
6.2 Ein realistischeres Modell . . . . . 35
6.2.1 Ein Kontinuumsmodell . . . . . 35
6.2.2 Ein diskretes Modell . . . . . 35

**7 Wärmeleitung** 38
7.1 Thermodynamik . . . . . 38
7.2 Transport . . . . . 39
7.3 Materialgesetze . . . . . 40
7.4 Die Wärmeleitungsgleichung . . . . . 40

III Die Programmiersprache C 44

8 Einführung 45
8.1 Das erste C–Programm . . . 45
8.2 Interne Details beim Compilieren (∗) . . . 47

9 Variablen, Datentypen und Operationen 48
9.1 Deklaration, Initialisierung, Definition . . . 48
9.2 Elementare Datentypen . . . 48
9.3 Felder und Strings . . . 50
9.3.1 Felder . . . 50
9.3.2 Mehrdimensionale Felder . . . 51
9.3.3 Zeichenketten (Strings) . . . 51
9.4 Ausdrücke, Operatoren und mathematische Funktionen . . . 52
9.4.1 Arithmetische Operatoren . . . 53
9.4.2 Vergleichsoperatoren . . . 54
9.4.3 Logische Operatoren . . . 54
9.4.4 Bitorientierte Operatoren (∗) . . . 55
9.4.5 Inkrement- und Dekrementoperatoren . . . 56
9.4.6 Adressoperator . . . 57
9.4.7 Prioritäten von Operatoren . . . 57
9.5 Operationen mit vordefinierten Funktionen . . . 59
9.5.1 Mathematische Funktionen . . . 59
9.5.2 Funktionen für Zeichenketten (Strings) . . . 60
9.6 Zusammengesetzte Anweisungen . . . 61
9.7 Nützliche Konstanten . . . 61
9.8 Typkonversion (cast) . . . 62
9.9 Standardein– und –ausgabe . . . 63
9.9.1 Ausgabe . . . 63
9.9.2 Eingabe . . . 64

10 Programmflusskontrolle 67
10.1 Bedingte Ausführung . . . 67
10.1.1 Die if()–Anweisung . . . 67
10.1.2 Die switch()–Anweisung . . . 68
10.2 Schleifen . . . 69
10.2.1 Der Zählzyklus (for–Schleife) . . . 70
10.2.2 Abweisender Zyklus (while–Schleife) . . . 73
10.2.3 Nichtabweisender Zyklus (do–while–Schleife) . . . 73
10.3 Anweisungen zur unbedingten Steuerungsübergabe . . . 74

11 Zeiger (Pointer) 76
11.1 Adressen . . . 76
11.2 Pointervariablen . . . 77
11.3 Adressoperator und Zugriffsoperator . . . 78
11.4 Zusammenhang zwischen Zeigern und Feldern . . . 79
11.5 Dynamische Felder mittels Zeiger . . . 80

12 Funktionen 83
12.1 Deklaration, Definition und Rückgabewerte . . . 83
12.2 Lokale und globale Variablen . . . 84
12.3 Call by value . . . 86
12.4 Call by reference . . . 88
12.5 Rekursive Programmierung . . . 88

12.6 Kommandozeilen–Parameter . . . . . 89
12.7 Wie werden Deklarationen gelesen? . . . . . 91
12.8 Zeiger auf Funktionen . . . . . 91

**13 Strukturierte Datentypen (∗) 94**
13.1 Strukturen . . . . . 94
13.1.1 Deklaration von Strukturen . . . . . 94
13.1.2 Definition von Strukturvariablen . . . . . 95
13.1.3 Felder von Strukturen . . . . . 95
13.1.4 Zugriff auf Strukturen . . . . . 95
13.1.5 Zugriff auf Strukturen mit Zeigern . . . . . 96
13.1.6 Geschachtelte Strukturen . . . . . 97
13.1.7 Listen . . . . . 98
13.2 Unions . . . . . 99
13.3 Aufzählungstyp . . . . . 100
13.4 Allgemeine Typendefinition . . . . . 100

**A Computerarithmetik (∗) 102**
A.1 Zahlendarstellung im Rechner und Computerarithmetik . . . . . 102
A.2 IEEE Gleitkommazahlen . . . . . 104
A.3 Computerarithmetik . . . . . 105

# Teil I

# Einführung in die Programmierung

# Kapitel 1

# Das Betriebssystem Linux

### Allgemeines

Die praktischen Programmierübungen im Computer–Pool werden an PCs stattfinden, auf denen das Betriebssystem Linux installiert ist. Linux ist neben WINDOWS das am meisten verbreitete Betriebssystem. Die Vorteile von Linux sind

- höhere Sicherheit durch strenge Unterteilung der Zugriffsrechte,
- Preis,
- offener Code.

Die Nachteile sind, dass

- manche Dinge nicht so komfortabel wie unter WINDOWS sind,
- WINDOWS–Software auf Linux nicht läuft und man sich umgewöhnen muss, wenn man die entsprechende Linux–Software nutzt.

Bezüglich des ersten Nachteils wurde allerdings im Laufe der vergangenen Jahre viel getan, dass oft kein Unterschied zu WINDOWS mehr vorhanden ist.

Der Name Linux tauchte erstmals 1991 nach der Veröffentlichung des ersten Linux–Kernels durch Linus Torvalds auf. Dieses Betriebssystem wird vor allem auf Servern eingesetzt, so zum Beispiel laufen die Server von Google und Wikipedia unter Linux. Der Einsatz auf Desktops ist vor allem im universitären Bereich zu finden, und auch da vor allem bei Mathematikern und Informatikern. Für den Desktop gibt es viele unterschiedliche Linux–Distributionen:

- SuSe, ist im deutschsprachigen Raum am meisten verbreitet,
- RedHat, ist im amerikanischen Raum am meisten verbreitet,
- Debian, läuft im Computer–Pool.

Informationen zu Unterschieden, Vor– und Nachteilen der einzelnen Distributionen findet man im Internet. Von einem Hörer der vergangenen Vorlesung ist das Buch [Bar04] zur Einarbeitung in Linux empfohlen wurden.

### Zugriffsrechte

Jeder Nutzer (user) ist in Linux einer Gruppe (group) zugeordnet. Die Zugriffsrechte jedes Files und Verzeichnisses in in Linux sind lesen (read, r), schreiben (write, w) und ausführen (execute, x). Diese Zugriffsrechte sind getrennt gesetzt für den user, die group und alle übrigen. Zum Beispiel, besagt

```
-rw-r----- 1 john users 29444 2007-10-13 12:22 tmp.txt
```

dass das File `tmp.txt` vom Nutzer john gelesen und geschrieben werden kann (Zeichen 2–4), von der Gruppe users nur gelesen werden kann (Zeichen 5–7) und alle übrigen dürfen mit diesem File nichts machen (Zeichen 8–10). Bei einem Verzeichnis

sieht diese Information zum Beispiel wie folgt aus

```
drwxr-xr-x 3 john users 312 2007-10-12 18:26 MOD_PROG
```

Das `d` am Anfang besagt, dass es sich um ein Verzeichnis (directory) handelt, der Nutzer john darf im Verzeichnis lesen, schreiben und in das Verzeichnis wechseln, die Gruppe users und alle übrigen dürfen nur lesen und in das Verzeichnis wechseln, dort aber nichts ändern (schreiben). Das Setzen und Ändern der Zugriffsrechte geschieht mit dem Befehl `chmod`. Wenn man zum Beispiel das File `tmp.txt` für alle les– und schreibbar machen will, so kann man das mit

```
chmod a+rw tmp.txt
```

und erhält danach die Information

```
-rw-rw-rw- 1 john users 29444 2007-10-13 12:22 tmp.txt
```

Man kann die Zugriffsrechte nur von Dateien und Verzeichnissen ändern, die einem gehören.
Das heißt, die Zugriffsrechte regeln wer was machen darf. Ist der Rechner ordentlich administriert, hat ein Virus oder ein Wurm keine Möglichkeit wichtige Dinge zu verändern, da er dazu nicht die Rechte hat. Die Rechte müssen so gesetzt sein, dass das nur der Administrator, der in Linux *root* heißt, machen darf und um als Administrator zu arbeiten, muss man ein entsprechendes Password eingeben. Aus Sicherheitsgründen sollte ein Password immer so gewählt sein, dass der Nutzernamen nicht ein Teil des Passwords ist und das Password auch Ziffern enthält.

## Werkzeuge

Man hat in Linux die Möglichkeit mit graphischen Benutzteroberflächen (wie in WINDOWS) oder auch auf der Kommandozeilenebene (Shell) zu arbeiten. Die Shell ist ein Kommandointerpreter, der von der Kommandozeile die Anweisungen einliest, diese auf Korrektheit überprüft und ausführt. Wenn man sich von außen auf einem Computer mit Betriebssystem Linux einloggt, kann man oft keine graphischen Oberflächen öffnen, zum Beispiel um in das Netz der Mathematik von zu Hause zu gelangen, muss man folgenden Weg gehen:

PC zu Hause → `contact.math.uni-sb.de` → PC im Büro

Dann ist man gezwungen, mit der Shell zu arbeiten. Ein Ziel dieser Veranstaltung besteht darin, dass die wichtigsten Shell–Kommandos vermittelt und praktiziert werden. Eine Liste wichtiger Kommandos wird am Ende des Kapitels gegeben.

Wichtige Werkzeuge, die wir brauchen sind:

- Konquerer (erfüllt etwa die Aufgaben wie der Explorer in Windows)
- Editoren: Kate, Kile, emacs, Xemacs (sind selbsterklärend), `vi` ist ein Kommandoline–Editor, der dann nützlich ist, wenn man keine graphische Oberflächen öffnen kann (wenn man beispielsweise von außerhalb auf einem Linux–Rechner eingeloggt ist)
- `matlab`, Software zum Programmieren von Verfahren, dieses Programm gehört nicht zur Linux–Distribution und muss gekauft werden.
- `gcc`, der Gnu–C–Compiler,
- `latex`, Programmiersprache zur Textverarbeitung.

Standardsoftware wie `Firefox, acroread` läuft natürlich auch unter Linux. Es gibt natürlich noch viel mehr nützliche Dinge, siehe Literatur, Internet, Handbücher, Übungen.

## Dateinamen, Baumstruktur

Dateinamen:

- Linux unterscheidet Groß– und Kleinschreibung,
- Der Schrägstrich / darf nicht verwendet werden, da er zur Trennung von Verzeichnisnamen dient,
- Sonderzeichen (Leerzeichen, Umlaute, &, ...) sollten vermieden werden, da einige eine spezielle Bedeutung besitzen.

Linux besitzt eine hierarchische, baumstrukturierte Verzeichnisstruktur, siehe Abbildung 1.1. Ausgangspunkt ist die Wurzel `/` (root). Die Position eines beliebigen (Unter)–Verzeichnisses ist durch den Pfad gegeben. Der Pfad gibt an, wie man von der Wurzel zu der gewünschten Datei beziehungsweise zum gewünschten Verzeichnis gelangt, zum Beispiel

```
pwd
```

(pwd – print name of current/working directory) gibt

```
/home/john
```

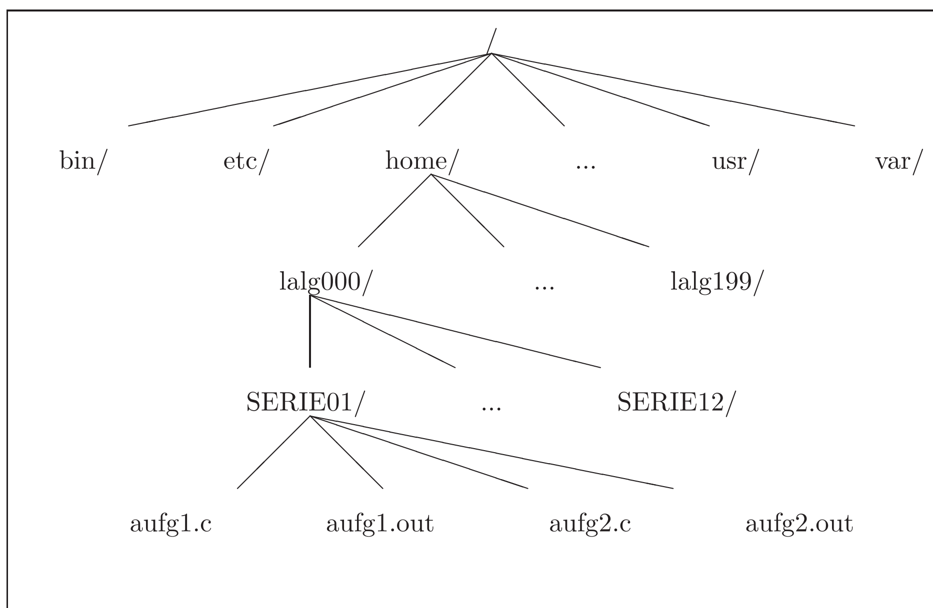


Abbildung 1.1: Beispiel für einen Verzeichnisbaum in Linux.

Pfade, die mit / beginnen, heißen absolute Pfade. Daneben gibt es noch die relativen Pfade, die relativ zur gegenwärtigen Position im Verzeichnisbaum sind. Wichtige relative Pfadbezeichner sind

- `.` : aktuelle Position im Verzeichnisbaum (Arbeitsverzeichnis),
- `..` : das übergeordnete Verzeichnis.

Es ist möglich, mehrere Dateien oder Verzeichnisse gleichzeitig anzusprechen, mit Hilfe sogenannter Wildcards

- `*` : ersetzt beliebig viele Zeichen (auch keines)
- `?` : ersetzt genau ein Zeichen
- `[zeichen1 - zeichen2]` : Auswahlsequence; alle Zeichen zwischen `zeichen1` und `zeichen2` werden ersetzt.

So kann zum Beispiel

```
ls tmp*
```

die Ausgabe

```
tmp tmp.txt
```

geben.

## Liste von Kommandos

Nähere Informationen zu den Kommandos erhält man mit dem `man` Befehl

```
man Befehlsname
```

Sucht man Befehle, die mit einem Schlüsselbegriff zusammenhängen, so verwende man

```
man -k Schlüsselbegriff
```

| Befehl | Bemerkungen |
|---|---|
| `cd` | wechsle Verzeichnis; zurück zum vorherigen Verzeichnis : `cd -`<br>ins Homeverzeichnis: `cd` |
| `chmod` | verändere Zugriffsrechte; z.B. Schreibrecht für alle Nutzer<br>`chmod a+w` *filename* |
| `cp` | kopiere Datei |
| `df` | gibt den belegten und freien Speicherplatz (Harddisc) zurück |
| `env` | zeigt die Umgebungsvariablen an |
| `find` | findet Dateien, z.B. um all Dateien mit Namen `core` zu finden<br>`find / -name core -print` |
| `grep` | sucht nach einem Muster in einer Datei, z.B. um alle `printf`–Befehle in `test.c` zu finden<br>`grep 'printf' test.c \| more` |
| `gzip` | komprimiert Dateien |
| `gunzip` | dekomprimiert Dateien *filename.gz* |
| `kill` | beendet Prozesse |
| `ll` | zeigt Inhalt von Verzeichnissen |
| `ls` | zeigt Inhalt von Verzeichnissen, wichtige Optionen`-ali` |
| `man` | Handbuch, z.B.<br>`man man` |
| `mkdir` | Anlegen von Verzeichnissen |
| `more` | Ansehen von Dateien |
| `mv` | verschieben von Dateien |
| `passwd` | Veränderung des Passwords |
| `ps` | zeigt laufende Prozesse an, z.B. vom Nutzer *abc*<br>`ps -u abc` |
| `pwd` | zeigt Pfadnamen zum gegenwärtigen Verzeichnis |
| `rm` | löscht Dateien, **nutze besser immer** `rm -i` |
| `rmdir` | löscht Verzeichnisse |
| `tail` | zeigt die letzten Zeilen einer Datei |
| `tar` | Erstellung von Archiven |
| `typeset` | setzen von Umgebungsvariablen, z.B.<br>`typeset -x PATH=$PATH:$HOME/bin` |
| `who` | zeigt wer im System eingeloggt ist |
| `which` | lokalisiert ausführbares Programm |

# Kapitel 2

# Algorithmen

Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder eines bestimmten Typs von Problemen. Eine exakte Definition dieses Begriffes ist nicht trivial und erfolgt in Informatikvorlesungen. Wir werden Algorithmen zur Lösung mathematischer Aufgabenstellungen verwenden.

**Beispiel 2.1** *Quadratische Gleichung.* Zur Lösung der quadratischen Gleichung

$$0 = x^2 + px + q, \quad p, q \in \mathbb{R},$$

im Bereich der reellen Zahlen kann man folgenden Algorithmus verwenden:

*Algorithmus 2.2 Lösung der quadratischen Gleichung.*

- berechne
$$a := -\frac{p}{2}$$
- berechne
$$D := a^2 - q$$
- falls $D \geq 0$, dann
$$x_1 := a + \sqrt{D}, \quad x_2 := a - \sqrt{D}$$
sonst Ausgabe *keine reelle Lösung*

Man kann aber auch einen anderen Algorithmus verwenden, der sich im letzten Schritt unterscheidet und die Vieta[1]sche Wurzelformel nutzt:

*Algorithmus 2.3 Lösung der quadratischen Gleichung.*

- berechne
$$a := -\frac{p}{2}$$
- berechne
$$D := a^2 - q$$
- falls $D < 0$, dann Ausgabe *keine reelle Lösung*
sonst
  - falls $a < 0$, setze
$$x_1 := a - \sqrt{D}, \quad x_2 := q/x_1$$
sonst falls $a > 0$, setze
$$x_1 := a + \sqrt{D}, \quad x_2 := q/x_1$$
sonst
$$x_1 := \sqrt{-q}, \quad x_2 := -x_1.$$

[1] Francois Vieta (Viète) 1540 – 1603

Beide Algorithmen liefern mathematisch dasselbe Ergebnis. In den Übungen wird man sehen, dass das auf einem Computer im allgemeinen nicht mehr der Fall ist.

Diese Algorithmen enthalten bereits ein wichtiges Merkmal, nämlich Verzweigungen. Man muss an hand von berechneten Werten sich zwischen zwei oder mehr Möglichkeiten entscheiden, welcher Teil des Algorithmus abzuarbeiten ist. □

**Beispiel 2.4** *Summe von Zahlen.* Man soll die Summe der natürlichen Zahlen von 1000 bis 10000 bilden. Ein möglicher Algorithmus ist, diese nacheinander aufzuaddieren:

*Algorithmus 2.5*

- setze $summe := 1000$
- setze für $i = 1001$ bis $i = 10000$

$$summe := summe + i$$

Hierbei ist *summe* als Variable zu verstehen, die einen bestimmten Speicherplatz im Computer einnimmt. Der neue Wert wird somit auf demselben Platz gespeichert, wie der bisherige Wert. Man muss zur Berechnung der Summe 9000 Additionen durchführen.

Auch dieser Algorithmus hat ein charakteristisches Merkmal, nämlich eine Schleife. Es gibt unterschiedliche Arten von Schleifen, siehe später.

Ein anderer Algorithmus nutzt die bekannte Summationsformel von Gauß[2]

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Damit erhält man

$$\sum_{i=1000}^{10000} i = \sum_{i=1}^{10000} i - \sum_{i=1}^{999} i = \frac{10000 \cdot 10001}{2} - \frac{999 \cdot 1000}{2} = 5000 \cdot 10001 - 500 \cdot 999.$$

Damit kann man das Ergebnis mit zwei Multiplikationen und einer Subtraktion berechnen.

Man sieht, es kann billige (effiziente) und teure Algorithmen geben, die zum gleichen Ergebnis führen. Ziel ist es natürlich, den jeweils effizientesten Algorithmus zu verwenden. □

Zur Beschreibung von Algorithmen nutzt man die folgenden Grundstrukturen:
- Anweisung,
- bedingte Verzweigung (Alternative),
- Wiederholung (Schleife, Zyklus).

Die Zusammenfassung mehrerer Anweisungen wird auch Sequenz genannt.

**Beispiel 2.6** Am Anfang von Algorithmus 2.2 kommt die folgende Sequenz

```
a = -p/2;
D = a*a-q;
```

(Hier wird MATLAB–Notation genutzt.) □

Bei der Alternative unterscheidet man die einfache und die vollständige Alternative.

**Beispiel 2.7** Die einfache Alternative haben wir bereits in den Algorithmen 2.2 und 2.3 gesehen:

[2]Johann Carl Friedrich Gauss (1777 – 1855)

```
if D<0
   disp('Keine reelle Loesung')
else
   Anweisungen zur Berechnung der Lösung
end
```

Im Algorithmus 2.3 findet man auch eine vollständige Alternative:

```
if a<0
   x_1 = a-sqrt(D);
   x_2 = q/x_1;
elseif a>0
   x_1 = a+sqrt(D);
   x_2 = q/x_1;
else
   x_1 = sqrt(-q);
   x_2 = -x_1;
end
```

Möglichkeiten zur einfachen Behandlung mehrerer Alternativen sind in MATLAB und C ebenfalls vorhanden: `switch`–Anweisung, siehe später. □

In einer Schleife oder einem Zyklus ist eine Sequenz wiederholt abzuarbeiten. Die Anzahl der Durchläufe muss dabei vorab nicht unbedingt bekannt sein, sie bestimmt sich oft im Laufe der Abarbeitung. Man unterscheidet abweisende (kopfgesteuerte) Zyklen und nichtabweisende (fußgesteuerte) Zyklen:

- abweisender Zyklus: Bedingung für die Abarbeitung wird vor Beginn des Zyklus getestet, es kann passieren, dass diese beim ersten Mal schon nicht erfüllt ist und der Zyklus wird nie abgearbeitet,
- nichtabweisender Zyklus: Bedingung für die Abarbeitung wird am Ende des Zyklus getestet, damit wird der Zyklus mindestens einmal abgearbeitet.

Beispiele dafür wird es in den Übungen geben. Ist die Anzahl der Durchläufe bekannt, dann wird der Zyklus durch einen Zähler gesteuert.

**Beispiel 2.8** Die Steuerung durch einen Zähler hatten wir bereits in Algorithmus 2.5:

```
summe = 1000;
for i=1001:10000
   summe = summe + i;
end
```

□

Jeder Zyklus erfordert Vorbereitungen, das heißt, vor Eintritt in den Zyklus sind entsprechende Variablen zu belegen. Im obigen Beispiel ist `summe = 1000` zu setzen. Im Zykluskörper selbst muss so auf die Abbruchbedingung eingewirkt werden, dass der Abbruch nach endlich vielen Durchläufen garantiert ist. Hat man dabei einen Fehler gemacht, dann kann das Programm den Zyklus nicht verlassen und es hilft nur, das Programm abzubrechen und den Fehler zu suchen.

# Kapitel 3

# Einführung in MATLAB

## 3.1 Allgemeines

MATLAB ist eine kommerzielle mathematische Software zur Lösung mathematischer Probleme und zur graphischen Darstellung der Ergebnisse. Die Verfahren in MATLAB beruhen auf Matrizen (MATrix LABoratory).

MATLAB ist leider nicht ganz billig. Im Computer–Pool kann man das Programm mit

```
/usr/local/matlab/bin/matlab
```

starten.

Zur Einarbeitung in MATLAB gibt es viele Bücher, siehe `www.amazon.de`. Das Buch [DS04] ist so eine Art Klassiker, der einen kurzen und knappen Überblick gibt (man muss wissen, wonach man suchen soll). Eine Uraltversion von [DS04] ist im Internet (siehe Homepage, auf der die Übungen stehen) verfügbar. Weitere frei verfügbare Beschreibungen findet man auf der gleichen Homepage und im Internet. Diese Dokumentationen beruhen zwar auf älteren Versionen von MATLAB, sind aber für diese Vorlesung vollkommen ausreichend. Es gibt eine umfangreiche und gute Hilfe innerhalb von MATLAB, Aufruf mit `help`.

Man programmiert in MATLAB mit einer plattformunabhängigen Programmiersprache, die auf der jeweiligen Maschine interpretiert wird. Durch den einfachen, mathematisch orientierten Syntax der MATLAB–Skriptsprache und durch umfangreiche vorhandene Funktionsbibliotheken ist die Erstellung von Programmen wesentlich einfacher möglich als beispielsweise unter C. Man braucht sich vor allem nicht um die Organisation des Speichers kümmern. Allerdings sind MATLAB–Programme im allgemeinen bedeutend langsamer als C–Programme.

Man kann sein Programm direkt in das MATLAB–Befehlfenster eintippen. Sinnvoller ist es jedoch, es in eine separate Datei zu tun und diese vom MATLAB–Befehlfenster aus aufzurufen. *Vorlesung: an Summe der ersten 100 Zahlen demonstrieren.* Mit dem Befehl `edit` wird ein Editor geöffnet, in dem man die Datei erstellen kann. MATLAB–Befehlsdateien besitzen die Endung `.m`, (M–Files). Mit dem Befehl `what` kann man sich die im gegenwärtigen Verzeichnis vorhandenen M–Files ansehen. Sie werden ausgeführt, indem sie im MATLAB–Befehlfenster einfach aufgerufen werden (die benötigten Parameter müssen natürlich übergeben werden). Weitere wichtige allgemeine MATLAB–Befehle sind `ls, cd, pwd`. Sie haben die gleiche Bedeutung wie in LINUX. Des weiteren sind die Befehle

`clear;` löscht alle Variablen
`clf;` löscht alle Bildfenster
`who;` zeigt alle Variablen an

wichtig, damit bei einem wiederholten Starten von Programmen nicht alte Belegungen die Ergebnisse verfälschen. Die Ausgabe von Text erfolgt mit `disp`. Die Formatierung mit `format`.

Die Nutzung von MATLAB ist an vielen Hochschulen Standard im Rahmen von Vorlesungen, die sich mit numerischen Verfahren beschäftigen. Hier werden nur die wichtigsten Befehle vorgestellt. Ansonsten gilt, was für jede Programmiersprache gilt: *Learning by doing.*

## 3.2 Bemerkungen zu Vektoren und Matrizen

Vektoren sind aus der Schule bekannt. Man unterscheidet

$$\mathbf{a} = (a_1, \ldots, a_n),$$

einen $n$–dimensionalen Zeilenvektor und

$$\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix},$$

einen $n$–dimensionalen Spaltenvektor. Die Anzahl der Komponenten eines Vektors nennt man Dimension (hier $n$, in der Schule im allgemeinen $n \in \{2, 3\}$).

Wandelt man einen Zeilenvektor in einen Spaltenvektor mit den gleichen Einträgen um (oder Spalten– in Zeilenvektor), so nennt man diese Operation transponieren. Der transponierte Vektor des obigen Zeilenvektors ist

$$\mathbf{a}^T = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}.$$

Das Skalarprodukt zweier Spaltenvektoren ist aus der Schule für $n = 3$ bekannt. Für zwei $n$–dimensionale Spalenvektoren ist es

$$\mathbf{a} \cdot \mathbf{b} = (\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b} = \sum_{i=1}^{n} a_i b_i.$$

Die Norm oder Länge eines Vektors ist

$$\|\mathbf{a}\| = (\mathbf{a} \cdot \mathbf{a})^{1/2} = \left( \sum_{i=1}^{n} a_i^2 \right)^{1/2}.$$

Matrizen und ihre tiefere Bedeutung sowie ihre Eigenschaften werden am Ende von Lineare Algebra I behandelt. Hier ist es nur wichtig, dass es zwei–dimensionale Felder sind, mit $m$ Zeilen und $n$ Zeilen:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}.$$

In diesem Sinne sind $n$–dimensionale Zeilenvektoren $1 \times n$ Matrizen und $m$–dimensionale Spaltenvektoren $m \times 1$ Matrizen.

## 3.3 Matrizen

MATLAB rechnet mit Matrizen. Dabei ist zum Beispiel eine Zahl eine $1 \times 1$–Matrix und ein Spaltenvektor eine $n \times 1$–Matrix. Operationen mit Matrizen sind nur dann wohldefiniert, wenn ihre Dimensionen entsprechende Bedingungen erfüllen, zum Beispiel müssen bei der Addition die Dimensionen gleich sein. Die Dimensionskontrolle wird in MATLAB streng durchgeführt. So ist es zum Beispiel nicht möglich, einen Zeilen– und einen Spaltenvektor der gleichen Länge zu addieren. Die Indizierung von Matrizen beginnt in MATLAB mit 1.

MATLAB rechnet auch mit komplexen Zahlen. Wichtige Konstanten sind:

- `pi`
- `i`, `j`, imaginäre Einheit.

Die Eingabe beziehungsweise Erzeugung von Matrizen kann auf verschieden Art und Weisen erfolgen:

- Eingabe der Matrixeinträge,
- Laden aus externen Dateien,
- Erzeugen mit vordefinierten Funktionen,
- Erzeugen mit eigenen Funktionen.

**Beispiel 3.1** Die Matrix

$$A = \begin{pmatrix} 2 & 4 & 7 \\ -5 & 4 & 2 \end{pmatrix}$$

kann wie folgt eingegeben werden

```
A = [2 4 7; -5, 4, 2]
```

Man kann auch jedes Element einzeln angeben:

```
A(1,1) = 2
A(1,2) = 4
A(1,3) = 7
A(2,1) = -5
A(2,2) = 4
A(2,3) = 2
```

Die Einheitsmatrix der Dimension $n \times n$ erhält man mit

```
B = eye(n)
```

wobei `n` vorher mit einer positiven ganzen Zahl belegt sein muss. Analog erhält man eine Matrix mit Nullen durch

```
C = zeros(n)
```

Eine $(m \times n)$–Matrix mit zufälligen Einträgen erhält man mit

```
D = rand(m,n)
```

Will man die Ausgabe der Matrizen auf dem Bildschirm unterdrücken, so beendet man die Befehle mit einem Semikolon. □

Wichtige Operationen mit Matrizen:

- Die transponierte Matrix $A^T$ einer gegebenen Matrix `A` erhält man mit

  ```
  B = A'
  ```

  Man kann die transponierte Matrix auch auf dem Speicherplatz der ursprünglichen Matrix speichern

  ```
  A = A'
  ```

- Die Dimension von `A` erhält man mit

  ```
  [m,n] = size(A)
  ```

  Hierbei ist `m` die Anzahl der Zeilen und `n` die Anzahl der Spalten.
- Die Teilmatrix mit den Zeilenindizes $i1, \ldots, i2$ und den Spaltenindizes $j1, \ldots, j2$ einer Matrix `A` erhält man mit

```
B = A(i1:i2,j1:j2)
```

Wird der Doppelpunktoperator ohne vorderes Argument gebraucht, so wird mit dem ersten Index begonnen; ohne hinteres Argument, wird mit dem letzten Index aufgehört. So erhält man die erste Zeile von `A` durch

```
B = A(1,:)
```

- Addition zweier Matrizen `A` und `B`, Subtraktion sowie Multiplikation werden mit den üblichen Symbolen bezeichnet

  ```
  C = A+B      C = A-B      C = A*B
  ```

- Multiplikation, Division und Potenzierung einer Matrix `A` mit einem Skalar $a$ ($(1 \times 1)$–Matrix) werden mit den üblichen Symbolen bezeichnet

  ```
  B = a*A      B = A/a      B=A^a
  ```

- Elementweise Multiplikation und Division werden wie folgt durchgeführt

  ```
  C = A.*B      C = A./B
  ```

  Beispiel:

$$\mathtt{A} = (1, 5), \quad \mathtt{B} = \begin{pmatrix} 3 \\ -2 \end{pmatrix}, \qquad \mathtt{A} * \mathtt{B} = -7, \quad \mathtt{A.} * \mathtt{B}' = (3, -10)$$

Die folgende Liste enthält wichtige Befehle, die man für Matrizen in MATLAB zur Verfügung hat. Die Einfachheit dieser Befehle sollte nicht darüber hinwegtäuschen, dass innerhalb von MATLAB zum Teil komplizierte Verfahren zur Berechnung der Ergebnisse ablaufen (siehe Vorlesung Praktische Mathematik). Vergleichbare Befehle stehen zum Beispiel in C nicht zur Verfügung. Daraus erklärt sich die Einfachheit, mit der man Programme in MATLAB erstellen kann. Für die angegebenen Befehle gibt es teilweise alternative Aufrufe, siehe MATLAB–Hilfe.

- der Rang einer $n \times n$–Matrix `A`:

  ```
  r = rank(A)
  ```

- die Inverse einer regulären $n \times n$–Matrix `A`:

  ```
  B = inv(A)
  ```

- die Determinante einer $n \times n$–Matrix `A`:

  ```
  d = det(A)
  ```

- die Spektralnorm einer $m \times n$–Matrix `A`:

  ```
  d = norm(A)
  ```

  Andere Normen können ebenfalls berechnet werden, siehe MATLAB–Hilfe. Ist `A` ein Vektor, dann ist die Spektralnorm die Euklidische Vektornorm.

- die Eigenwerte und Eigenvektoren einer $n \times n$–Matrix `A`:

  ```
  [u,v] = eig(A);
  ```

  Dabei enthält `v` eine Diagonalmatrix mit den Eigenwerten und die Matrix `u` enthält die zugehörigen Eigenvektoren.

- die Lösung eines linearen Gleichungssystems `Ax` = `b` mit einer regulären $n \times n$–Matrix `A` erhält man mit

  ```
  x = A\b
  ```

## 3.4 Wichtige Funktionen in MATLAB

| Befehl | Bedeutung |
|---|---|
| `atan` | Arcus–Tangens |
| `cos` | Kosinus |
| `exp` | Exponentialfunktion |
| `log` | Logarithmus naturalis ln ! |

| | |
|---|---|
| log10 | Logarithmus zur Basis 10 |
| sin | Sinus |
| sqrt | Wurzel |
| tan | Tangens |

Für Einzelheiten, zum Beispiel was bei matrixwertigen Argumenten passiert, siehe MATLAB–Hilfe. Weitere Funktionen findet man ebenso in der MATLAB–Hilfe.

## 3.5 Ablaufkontrolle

Die Ablaufkontrolle beinhaltet Alternativen und Zyklen.

Die Alternative wird wie folgt programmiert:

```
if expr
   sequenz
elseif expr
   sequenz
else
   sequenz
end
```

Dabei gibt *expr* einen Wahrheitswert (true oder false) zurück. Folgende Relationen und wichtige logische Operatoren stellt MATLAB zum Vergleich von Ausdrücken zur Verfügung:

| Befehl | Bedeutung |
|---|---|
| < | kleiner |
| <= | kleiner oder gleich |
| > | größer |
| >= | größer oder gleich |
| == | gleich |
| ~= | ungleich |
| && | logisches und |
| || | logisches oder |
| ~ | logisches nicht |
| xor(·,·) | exklusives oder (entweder oder) |

Soll zum Beispiel kontrolliert werden, ob eine Matrix ein Zeilen– oder Spaltenvektor ist, so kann man das mit

```
[m,n] = size(a)
if (m==1) || (n==1)
```

tun.

Innerhalb der `if`–Anweisung sind mehr als eine `elseif`–Abfrage möglich. Eine Alternative für Mehrfachverzweigungen bildet die `switch`–Anweisung:

```
switch switch expr
   case expr,
      sequenz
   case expr,
      sequenz
   case expr,
```

```
        sequenz
    otherwise,
        sequenz
end
```

Hier wird der Ausdruck nach dem `switch`–Befehl ausgewertet und dann der entsprechende Teil der Anweisung abgearbeitet, bei welcher der Ausdruck hinter dem `case`–Befehl mit der Auswertung des `switch`–Befehls übereinstimmt.

Mit einer `for`–Schleife wird eine vorgegebene Anzahl von Zyklen durchlaufen, etwa

```
for i=1:100
    sequenz
end
```

Eine `while`–Schleife wiederholt eine Sequenz so oft, bis ein logisches Abbruchkriterium erfüllt ist, etwa

```
i=1;
while i<100
    i= input('i ');
end
```

Diese Schleife wird erst abgebrochen, wenn eine Zahl $i \geq 100$ eingegeben wird.

Sowohl die `for`– als auch die `while`–Schleife sind abweisend. Ein vorzeitiges Verlassen eine Schleife ist mit dem `break`–Befehl möglich.

## 3.6 Graphik

Eine Stärke von MATLAB ist die Graphik, mit der man sich schnell und unkompliziert die berechneten Ergebnisse ansehen kann. Wie generell in MATLAB, werden bei der Graphik Daten gezeichnet, die in Vektoren und Matrizen gespeichert sind.

Zweidimensionale Graphiken erhält man mit dem `plot`–Befehl:

```
for i=1:101
    x(i) = (i-1)/100;
    y(i) = x(i)^3-0.5;
end
plot(x,y)
```

Damit wird die Funktion $x^3 - 0.5$ im Intervall $[0, 1]$ gezeichnet. Übergibt man nur ein Argument an `plot`, dann sind die Werte auf der $x$–Achse die Indizes der Vektoreinträge. Für verfügbare Linienarten und –farben sei auf `help plot` sowie auf die Literatur verwiesen. Von der Hilfe zu `plot` wird man dann auch zu weiteren Befehlen geführt, mit denen man eine Graphik verschönern kann, wie `legend, xlabel, axis`. Man kann die Graphiken auch interaktiv editieren.

Die graphische Darstellung von Flächen über der Ebene geht mit dem `mesh`–Befehl:

```
mesh(x,y,Z)
```

Dabei sind `x,y` die Vektoren mit den x– und y–Koordinaten und in der Matrix `Z` stehen die zugehörigen Funktionswerte: $Z(i, j)$ ist der Funktionswert im Punkt $(x(i), y(j))$.

Will man in eine vorhandene Graphik weitere Bildelemente einfügen, dann nutzt man den Befehl `hold`:

```
hold on
```

Damit werden die vorhandenen Bildelemente nicht gelöscht.

# Teil II

# Mathematische Modellierung

# Kapitel 4

# Mathematische Modelle

Die[1] Herleitung, Analysis und numerische Simulation von mathematischen Modellen realer Prozesse ist die Grundaufgabe der Angewandten Mathematik. Selbst wenn man sich nur für Teilaspekte interessiert, ist es meist wichtig, die Bedeutung und Struktur der zu Grunde liegenden mathematischen Modelle zu verstehen. In dieser Vorlesung kann nur eine Einführung in die Mathematische Modellierung gegeben werden.

Als ein mathematisches Modell kann man grundsätzlich jede berechenbare (im deterministischem oder stochastischem Sinn) Menge mathematischer Vorschriften, Gleichungen und Ungleichungen bezeichnen, die einen Aspekt eines realen Vorgangs beschreiben sollen. Dabei sollte man sich von vornherein bewusst sein, dass es sich bei einem Modell immer um eine Vereinfachung handelt und der reale Vorgang praktisch nie in seiner vollen Komplexität beschrieben wird. Die erste Unterscheidung erfolgt in

- qualitative Modelle, das heißt Modelle, die prinzipiell die Struktur eines Prozesses beschreiben sollen und gewisse qualitative Voraussagen (etwa über langfristige Geschwindigkeit von Wachstumsprozessen) ermöglichen sollen, die aber keine expliziten Werte für die Variablen des Systems liefern,
- quantitative Modell, das heißt Modelle, die für quantitative Voraussagen der Werte von gewissen Variablen genutzt werden sollen.

Qualitative Modell verwendet man oft in den Wirtschaftswissenschaften, zum Beispiel um die Dynamik der Preisbildung zu verstehen, und auch in manchen Naturwissenschaften wie der Ökologie. Dort kann ein qualitatives Modell genügen um zu verstehen, ob sich ein ökologisches Gleichgewicht ausbildet oder ob es zu einer möglichen Katastrophe kommt. Im Allgemeinen bevorzugt man in Naturwissenschaft und Technik jedoch quantitative Modelle. In der Vorlesung werden auch nur solche Modelle behandelt werden.

Bevor man ein mathematisches Modell entwickelt oder spezielle Modelle auf einen bestehenden Prozess anwendet, sollte man sich Klarheit über die Skalen (Orts– und Zeitskalen) verschaffen auf denen man den Prozess betrachtet, sowie auf jene Skalen, die einen Einfluss auf den Prozess besitzen. So werden etwa für die Beschreibung einer Straßenbeleuchtung quantenmechanische Effekte kaum von Bedeutung sein. Auf der anderen Seite wird die Dynamik turbulenter Strömungen stark von den kleinen Wirbeln beeinflusst. Die Reduktion auf die sogenannten relevanten Skalen ist wichtig, um das Modell in einer sinnvollen Größe zu halten, die dann auch numerische Simulationen in aktzeptabler Zeit erlaubt. Ebenso ist es wichtig, nur jene Effekte zu modellieren, die den Prozess auch tatsächlich beeinflussen, um das Modell und die Rechenzeit klein zu halten. Zum Beispiel könnte man bei der Model-

[1] nach [Bur07]

lierung einer Strömung auch die Wärmeleitung darin modellieren. Da kleine Temperaturschwankungen jedoch vernachlässigbare Auswirkungen besitzen, wird man oft darauf verzichten. Nur bei Prozessen mit starken Temperaturschwankungen, zum Beispiel in Gasturbinenbrennkammern, ist die Kopplung von Strömungs– und Wärmeleitungsmodellen unerlässlich.

Eine weitere Unterscheidung von mathematischen Modellen besteht in der Natur der Unbekannten:

- diskrete Modelle bestehen aus einer endlichen Anzahl von Partikeln (Atomen, Molekülen, ...), deren Eigenschaften (Position, Geschwindigkeit, Spin, ...) durch das Modell beschrieben werden,
- Kontinuumsmodelle beschreiben die Dichten der Variablen, normalerweise als Funktionen von Ort und Zeit.

# Kapitel 5

# Grundprinzipien der Mathematischen Modellierung

In[1] diesem Abschnitt werden die Grundprinzipien der Mathematischen Modellierung vorgestellt. Ausführlichere Darstellungen findet man in der Literatur, zum Beispiel in [Seg72, CS74].

## 5.1 Modellierungszyklus

Der Zyklus der Mathematischen Modellierung läuft im allgemeinen wie folgt ab:

1. Verständnis des realen Problems.
2. Wahl der Skalen und der entsprechenden mathematischen Beschreibung.
3. Entwicklung eines mathematischen Modells.
4. Sensitivitätsanalyse und eventuelle Vereinfachung des Modells.
5. Numerische Simulation des Modells.
6. Interpretation der Lösung.
7. Vergleich der Lösung mit den realen Daten.
8. Falls nötig, Verfeinerung des Modells oder Änderung der Parameter.

Oft müssen die Ergebnisse noch entsprechend aufbereitet und präsentiert werden (→ (Pro–)Seminare im Studium).

Mathematische Modellierung ist, vor allem in der Technik, keine Einbahnstraße. Die Modellierung verfolgt meist das klare Ziel durch besseres Verständnis in den Prozess eingreifen zu können. Dies kann durch die Anpassung von Parametern (Kontrolle) oder sogar durch die Auslegung eines neuen Prozesses (Prozess–Design) erfolgen. Deshalb werden sich in der Praxis die obigen Schritte stark gegenseitig (und nicht nur in aufsteigender Richtung) beeinflussen. So können zum Beispiel die numerische Simulation und Interpretation der Lösung zum besseren Verständnis des Verhaltens des ursprünglichen Prozesses beitragen. Sie können aber auch dazu führen, dass man die ursprüngliche Wahl der Skalen und des Modells korrigieren muss. Der gesamte Modellierungsprozess ist in Abbildung 5.1 dargestellt.

Der Vergleich mit realen Daten ermöglicht es oft, Fehlerquellen zu finden und zu eliminieren. Diese können von Modellierungsfehlern, über Fehler bei der numerischen Berechnung (Diskretisierungsfehler, Verfahrensfehler, Rundungsfehler, siehe

[1] nach [Bur07]

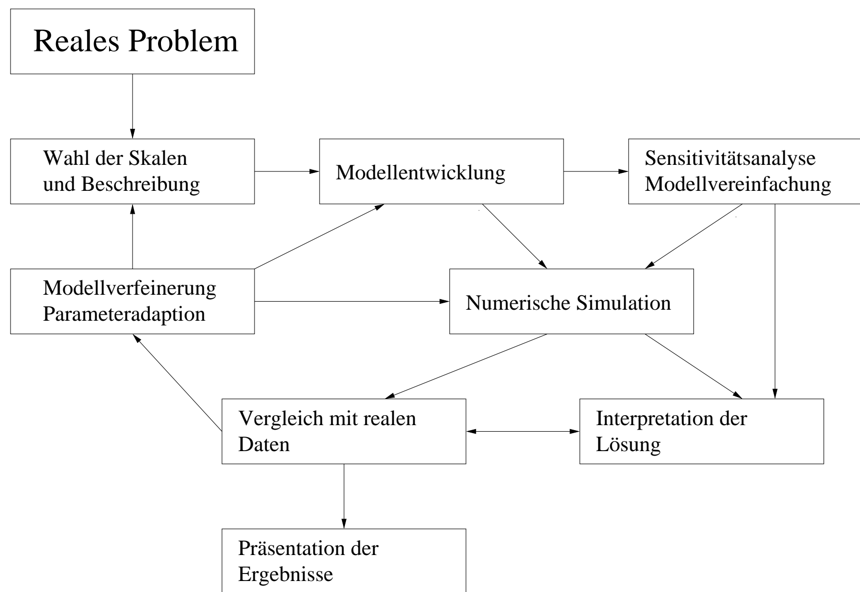


Abbildung 5.1: Schematische Darstellung des Modellierungszyklus.

Vorlesung Praktische Mathematik) bis hin zu Programmierfehlern bei der Implementierung reichen. Man darf bei diesem Vergleich aber auch nicht außer acht lassen, dass reale Daten oft mit nicht unerheblichen Messfehlern behaftet sind. Um Fehler effizient aufspüren zu können, ist es wichtig, geeignete (einfache) Testfälle zu betrachten.

Für die weitere Darstellung der einzelnen Schritte werden wir ein gemischtes Modell verwenden, das heißt eine Abbildung der Gestalt

$$y = M(x(p), p), \tag{5.1}$$

wobei $M : X \times P \to Y$ eine Abbildung zwischen Mengen in geeigneten Funktionenräumen ist. Hierbei werden $x \in X$ als Variablen, $y \in Y$ als Output und $p \in P$ als Parameter bezeichnet. Das Modell wird zunächst als abstrakte Abbildungsvorschrift betrachtet. In der Praxis wird die Auswertung des Operators $M$ aber die Lösung von Gleichungssystemen, Optimierungsproblemen oder stochastische Simulationen erfordern, aus denen man die Variablen $x(p)$ bestimmt.

## 5.2 Dimensionslose Variablen und Skalierung

Der erste Schritt bei der Betrachtung eines realen Modells ist die Überführung in eine dimensionslose Form und eine geeignete Skalierung. Die Variablen und Parameter in einem technischen Modell haben im allgemeinen eine physikalische Dimension und es kann nur im Vergleich mit anderen auftretenden Grössen entschieden werden, ob ein Wert groß oder klein ist. Eine Länge von einem Millimeter ist zum Beispiel für die Simulation der Wärmeleitung in einem Wohnraum relativ klein, für die Simulation eines modernen Halbleitertransistors aber riesig. Um absolute Größen zu erhalten, ist es wichtig, alle auftretenden Größen richtig zu skalieren.

Sei $x_i \in \mathbb{R}$ eine Komponente der Variablen. Dann kann man die Skalierung als eine Variablentransformation der Form

$$\tilde{x}_i = f_i(x_i)$$

mit einer geeigneten bijektiven Funktion $f_i : \mathbb{R} \to \mathbb{R}$ betrachten. In der Praxis sollte am besten $\tilde{x}_i \approx 1$ oder $|x_i| \leq 1$ gelten. Um dies zu erreichen, muss man typische Werte von $x_i$ abschätzen, was im allgemeinen eine grundlegende Einsicht in die Physik des Problems erfordert.

Die neue Variable $\tilde{x}_i$ heißt dimensionslos, falls $f_i(x_i)$ keine physikalische Dimension besitzt. Die einfachste und am häufigsten verwendete Art der Skalierung ist die Nutzung einer affinen Funktion, das heißt

$$\tilde{x}_i = a_i x_i + b_i$$

mit Konstanten $a_i, b_i \in \mathbb{R}$. Dabei besitzt $b_i$ keine physikalische Dimension und $a_i^{-1}$ hat dieselbe Dimension wie $x_i$. Man wählt dann $a_i^{-1}$ als typischen Wert oder Maximalwert von $|x_i|$.

In der gleichen Weise wie die Variable $x_i$ kann man auch den Output $y_j$ und folglich die Abbildung $M$ skalieren und in eine dimensionslose Form transformieren. Für die Parameter $p_k$ bleibt dann weniger Freiheit. Bei richtiger Skalierung erhält man automatisch dimensionslose Parameter $\tilde{p}_k$.

**Beispiel 5.1 Wurf.** Wir betrachten den Flug eines (sehr kleinen) Balls, der von einer Ebene mit der Normalen $(0,0,1)$ mit der Geschwindigkeit $\mathbf{V} = (V_1, V_2, V_3)$ abgeschossen wird. Als Output soll seine maximal erreichte Höhe und die Entfernung bis zum Auftreffen auf der Ebene berechnet werden.

Dazu werden die Zeit $t \in \mathbb{R}$ und die zeitabhängigen Variablen $(x_1, x_2, x_3)$ eingeführt, um die Ortskoordinaten des Balls zu bestimmen. Der Radius des Balls wird ignoriert und er wird als Massepunkt betrachtet. Wir wählen die Zeitskala und die Anfangswerte so, dass

$$\mathbf{x}(0) = (x_1(0), x_2(0), x_3(0)) = (0,0,0) \tag{5.2}$$

gilt. Seine Anfangsgeschwindigkeit ist

$$\frac{d\mathbf{x}}{dt}(0) = \left( \frac{dx_1}{dt}(0), \frac{dx_2}{dt}(0), \frac{dx_3}{dt}(0) \right) = \mathbf{V} = (V_1, V_2, V_3). \tag{5.3}$$

Als nächstes nutzen wir das Grundgesetz der Dynamik (Newton[2]sche Bewegungsgleichungen): Kraft ist gleich Masse mal Beschleunigung. In unserem Beispiel wirkt nur die Schwerkraft und wir erhalten

$$m \frac{d^2\mathbf{x}}{dt^2} = mg \frac{R^2}{(x_3(t) + R)^2} (0,0,-1), \tag{5.4}$$

wobei $m$ die Masse des Balls ist, $g$ die Erdbeschleunigung und $R$ der Erdradius. Der Vektor $(0,0,-1)$ zeigt an, dass die Kraft nach unten gerichtet ist, wobei vernachlässigt wurde, dass die Erde eine Kugel ist. Der dritte Faktor modelliert, dass die Erdanziehungskraft mit wachsendem Abstand zum Erdmittelpunkt kleiner wird.

Um den Output zu berechnen, benötigen wir noch die Variablen $T_1$, $T_2$ und die Gleichungen

$$\frac{dx_3}{dt}(T_1) = 0, \quad x_3(T_2) = 0$$

um die Outputs zu bestimmen. Die erste Gleichung beschreibt die Stelle, an der sich die Flugbahn des Balls umkehrt und die zweite Gleichung den Auftreffpunkt des Balls. Der Output ist also gegeben durch

$$y_1 = x_3(T_1), \quad y_2 = \sqrt{x_1^2(T_2) + x_2^2(T_2)}.$$

[2] Issac Newton (1643 – 1727)

Zusammenfassend besitzt das Modell die Variablen $t, T_1, T_2$ und $\mathbf{x}(t)$, die Parameter $m, g, R$ und $\mathbf{V}$ sowie den Output $y_1, y_2$.

Wir beginnen die Skalierung mit der Zeitvariablen und führen eine typische Zeitskala $\tau$ ein. Als transformierte, dimensionslose Variablen erhält man

$$\left(\tilde{t}, \tilde{T}_1, \tilde{T}_2\right) = \tau^{-1}\left(t, T_1, T_2\right).$$

In gleicher Weise wird die Ortsvariable mittels einer typischen Länge $\lambda_i$ skaliert

$$\tilde{x}_i(\tilde{t}) = \tilde{x}_i(\tau^{-1}t) = \lambda_i^{-1} x_i(\tau^{-1}t).$$

Für die Ableitung der skalierten Ortsariablen nach der skalierten Zeit erhält man mit Kettenregel

$$\frac{d\tilde{x}_i}{d\tilde{t}} = \frac{d\left(\lambda_i^{-1}x\right)}{dt}\frac{dt}{d\tilde{t}} = \frac{\tau}{\lambda_i}\frac{dx_i}{dt}.$$

Setzt man dies in die Formel (5.3) für die Anfangsgeschwindigkeit ein, so erhält man

$$\frac{d\tilde{x}_i}{d\tilde{t}}(0) = \frac{\tau}{\lambda_i}\frac{dx_i}{dt}(0) = \frac{\tau}{\lambda_i}V_i.$$

Das heißt, aus der Skalierung der Orts– und Zeitvariablen erhält man automatisch die dimensionslosen Anfangsgeschwindigkeiten $\tau V_i/\lambda_i$. Bei unserem Beispiel sind die gegebenen Werte die Geschwindigkeiten und wir kennen aus der Aufgabenstellung keine typischen Längen. Wir wählen die Skalierung

$$\lambda_i = \tau V_i, \tag{5.5}$$

falls $V_i \neq 0, i = 1, 2, 3$. Das setzen wir im weiteren immer voraus. Die Spezialfälle, dass es Anfangsgeschwindigkeiten $V_i = 0$ gibt, werden nicht betrachtet. Es ist einleuchtend die typische Länge proportional zu $V_i$ zu nehmen, denn wenn die Geschwindigkeit in eine Richtung doppelt so gross wie in eine andere ist, wird der Ball auch ungefähr die doppelte Länge in die erste Richtung zurücklegen. Die dimensionslosen Anfangsbedingungen sind nun einfach

$$\frac{d\tilde{x}_i}{d\tilde{t}}(0) = 1. \tag{5.6}$$

Durch Anwendung der Kettenregel auf die zweiten Ableitungen der Bewegungsgleichung erhält man

$$\frac{d^2\tilde{x}_i}{d\tilde{t}^2} = \frac{\tau}{\lambda_i}\frac{d}{d\tilde{t}}\left(\frac{dx_i}{dt}\right) = \frac{\tau^2}{\lambda_i}\frac{d^2x_i}{dt^2}.$$

Einsetzen in (5.4) gibt die dimensionslosen Bewegungsgleichungen (komponentenweise)

$$\begin{aligned}
\frac{d^2\tilde{x}_1}{d\tilde{t}^2}(\tilde{t}) &= 0, \\
\frac{d^2\tilde{x}_2}{d\tilde{t}^2}(\tilde{t}) &= 0, \\
m\frac{d^2\tilde{x}_3}{d\tilde{t}^2}(\tilde{t}) &= -\frac{mg\tau^2}{\lambda_3}\frac{R^2}{(x_3(t)+R)^2} = -\frac{mg\tau^2}{\lambda_3}\frac{R^2}{(\lambda_3\tilde{x}_3(\tilde{t})+R)^2}.
\end{aligned}$$

Man kann die Masse kürzen und die dritte Gleichung in die Form

$$\frac{d^2\tilde{x}_3}{d\tilde{t}^2}(\tilde{t}) = -\frac{\alpha}{\left(\beta\tilde{x}_3(\tilde{t})+1\right)^2}, \quad \alpha = \frac{g\tau^2}{\lambda_3},\ \beta = \frac{\lambda_3}{R} \tag{5.7}$$

umschreiben, wobei die Parameter $\alpha, \beta$ dimensionslos sind.

Nun besteht noch die Freiheit, die typische Zeiteinheit $\tau$ zu wählen. Das kann so realisiert werden, dass $\alpha = 1$ wird, also

$$\tau = \sqrt{\frac{\lambda_3}{g}} \overset{(5.5)}{=} \sqrt{\frac{\tau V_3}{g}} \quad \Longrightarrow \quad \tau = \frac{V_3}{g}.$$

Für die typischen Längenskalen gilt damit

$$\lambda_i = \frac{V_3 V_i}{g}, \quad i = 1, 2, 3. \tag{5.8}$$

Man beachte, dass man aus der Skalierung automatisch Informationen über typische Orts– und Zeitskalen in Abhängigkeit der gegebenen Parameter (hier Geschwindigkeit und Erdbeschleunigung) erhält. Andererseits ist diese Wahl nicht eindeutig, man hätte die Skalierung auch so wählen können, dass $\beta = 1$ gilt.

Für den Output nimmt man die natürlichen Skalierungen

$$\tilde{y}_1 = \lambda_3^{-1} y_1, \quad \tilde{y}_2 = \min\{\lambda_1^{-1}, \lambda_2^{-1}\} y_2.$$

Nehmen wir an, dass $\lambda_2 \leq \lambda_1$ gilt, dann sind

$$\tilde{y}_1 = \tilde{x}_3(\tilde{T}_1), \quad \tilde{y}_2 = \sqrt{\tilde{x}_1^2(\tilde{T}_2) + \gamma \tilde{x}_2^2(\tilde{T}_2)}, \ \gamma = \frac{\lambda_2^2}{\lambda_1^2} \leq 1.$$

Im resultierenden dimensionslosen System treten nur noch die dimensionslosen Parameter

$$\beta = \frac{\lambda_3}{R} \overset{(5.8)}{=} \frac{V_3^2}{gR}, \quad \gamma = \frac{\lambda_2^2}{\lambda_1^2} \overset{(5.5)}{=} \frac{V_2^2}{V_1^2}$$

auf. Die Anzahl der Parameter hat sich damit von ursprünglich sechs auf zwei reduziert. Solch ein Verhalten ist typisch, es gibt fast immer redundante Parameter (hier die Masse $m$) beziehungsweise weitere, die man durch Skalierung eliminieren kann. Die am Ende auftretenden Parameter sind fast immer relative Größen zwischen den ursprünglichen Parametern. Man nennt sie effektive Parameter. □

## 5.3 Sensitivitätsanalyse

Ein weiterer wichtiger Aspekt der Modellierung ist die Sensitivitätsanalyse. Man betrachtet dabei die Sensitivität des Systems bezüglich der Parameter $p$. Im speziellen ist man daran interessiert, wie sich der Output des Modells bei kleinen Variationen der Parameter ändern wird.

Wir ein generisches Modell mit Parametern $p$ betrachtet, so kann der Output als Funktion der Parameter aufgefasst werden, das heißt $\mathbf{y} = \mathbf{y}(\mathbf{p})$. Bei einer kleinen Variation $\Delta p$ der Parameter kann man die Änderung des Outputs durch eine Taylor[3]–Approximation erster Ordnung beschreiben, das heißt

$$\mathbf{y}(\mathbf{p} + \Delta\mathbf{p}) \approx \mathbf{y}(\mathbf{p}) + \frac{\partial \mathbf{y}}{\partial \mathbf{p}} \Delta\mathbf{p}.$$

(Skizze, die diese Formel erklärt.) Für die relative Änderung des Outputs hat man dann die Abschätzung

$$\frac{\|\Delta\mathbf{y}\|}{\|\Delta\mathbf{p}\|} = \frac{\|\mathbf{y}(\mathbf{p} + \Delta\mathbf{p}) - \mathbf{y}(\mathbf{p})\|}{\|\Delta\mathbf{p}\|} \approx \left\| \frac{\partial \mathbf{y}}{\partial \mathbf{p}} \right\|.$$

[3]Brook Taylor (1685 – 1731)

Damit kann die relative Änderung erster Ordnung durch die Größe der Ableitung nach den Parametern abgeschätzt werden. Man nennt die Größe der Ableitung des Outputs nach den Parametern auch Sensitivität.

Die Sensitivitätsanalyse von Beispiel 5.1 benötigt leider mathematische Hilfsmittel, die im ersten Semester noch nicht zur Verfügung stehen. Man kann aber zum Beispiel zeigen:

- Der Output $y_1$ ist sehr sensitiv zur Anfangsgeschwindigkeit $V_3$. Das ist einleuchtend, denn der Ball wird umso höher fliegen, desto schneller er in vertikale Richtung abgeschossen wird.
- Der Output $y_1$ hängt nicht von der Anfangsgeschwindigkeit $V_1$ ab.

## 5.4 Modellvereinfachungen

Sehr häufig enthalten Modelle Terme, die das Ergebnis nicht stark beeinflussen, die aber die (numerische) Lösung des Modells erschweren. In solchen Fällen ist es wünschenswert, die Modelle durch Weglassen dieser Terme zu vereinfachen.

Im speziellen vereinfacht man Modelle durch Eliminieren kleiner Terme und Parameter. Um entscheiden zu können, welche Terme klein sind, muss man das Problem geeignet skalieren. Dann sieht man, welche Terme mit kleinen Parametern multipliziert werden und weggelassen werden können.

**Beispiel 5.2 Modellvereinfachung im Beispiel 5.1.** In der skalierten Version treten nur die Parameter $\beta$ und $\gamma$ auf. Im allgemeinen wird man vermuten, dass die Höhe in welcher der Ball sich bewegt, klein ist im Vergleich zum Erdradius. Diese Höhe wird durch die charakteristische Länge $\lambda_3$ charakterisiert, also

$$\lambda_3 \ll R \stackrel{(5.8)}{\iff} \frac{V_3^2}{g} \ll R \iff \frac{V_3^2}{Rg} \ll 1.$$

Damit gilt $\beta \ll 1$. Da nur bereits skalierte Terme mit $\beta$ multipliziert werden, also Terme der Größenordnung 1, kann man folgern, dass damit auch $\beta\,|\tilde{x}_i| \ll 1$ und $\beta\tilde{x}_i + 1 \approx 1$ gelten. Somit vereinfacht sich die Bewegungsgleichung (5.7) zu

$$\frac{d^2\tilde{x}_3}{d\tilde{t}^2}(\tilde{t}) = -1.$$

Aus dieser Gleichung erhält man durch zweimaliges Integrieren und die Nutzung der Anfangsbedingungen (5.2) (diese muss noch entdimensionert werden) und (5.6) die Lösung

$$\tilde{x}_3 = \tilde{t} - \frac{\tilde{t}^2}{2},$$

eine Parabel. Für die maximal erreichte Höhe ergibt sich

$$T_1 = \frac{V_3}{g}, \quad y_1 = \frac{V_3^2}{2g}.$$

□

**Bemerkung 5.3 Modellfehler.** Bei der Betrachtung des Kanonenschusses wurden einige physikalische Aspekte nicht betrachtet beziehungsweise vereinfacht:

- Der richtige Ball ist dreidimensional und keine Punktmasse.
- Der richtige Ball besitzt eine Eigenbewegung, zum Beispiel Rotation, die vernachlässigt wurde.

- Wenn man keine Punktmasse betrachtet, sondern einen richtigen Körper, tritt Reibung durch den Luftwiderstand auf. Diese muss modelliert werden (Stokes[4]'sches oder Newton'sches Reibungsgesetz).
- Die Fallbeschleunigung ist nur näherungsweise bekannt.

Wie groß die durch diese Dinge verursachten Modellfehler sind, hängt vom konkreten Problem ab. □

[4] George Gabriel Stokes (1819 – 1903)

# Kapitel 6

# Modellierung von Wachstumsprozessen

Dieses[1] Kapitel befasst sich mit Modellen, die das Wachstum von Lebewesen beschreiben.

## 6.1 Ein einfaches Modell

### 6.1.1 Ein Kontinuumsmodell

Wir beginnen mit einem einfachen Modell zur Populationsdynamik. Seien $t \geq 0$ die Zeit und $t \mapsto x(t)$ eine Funktion, die die Anzahl der Lebewesen einer Population zur Zeit $t$ angibt.

Eine erste Idee ist, die zeitliche Änderung der Population mit der Größe der Population zu koppeln. Das könnte bedeuten, dass im Falle einer großen Population auch ein großes Wachstum vorliegt. Dies würde der Erfahrung entsprechen, dass viele Lebewesen mehr Nachwuchs produzieren als wenige. Die zeitliche Änderung der Population ist durch die Ableitung von $x(t)$ nach $t$ gegeben. Im ersten Modell soll diese Ableitung also proportional zur Größe der Population sein, das heißt es gilt

$$\frac{dx}{dt} = rx, \quad r \in \mathbb{R}, \tag{6.1}$$

$$x(0) = x_0. \tag{6.2}$$

Hierbei ist der Parameter $r$ der Proportionalitätsfaktor, der die unterschiedlichen Wachstumsraten für Populationen unterschiedlicher Lebewesen beschreibt. Dieser Faktor ist vorgegeben, er wird gegebenfalls durch Experimente bestimmt. Außerdem ist in (6.2) zur Vervollständigung des mathematischen Modells die Populationsgröße zum Anfangszeitpunkt gegeben, der zweite Parameter des Modells. Das mathematische Modell (6.1), (6.2) ist ein Anfangswertproblem mit einer gewöhnlichen Differentialgleichung.

Man kann Differentialgleichungen nur in Spezialfällen analytisch lösen. Bei (6.1), (6.2) handelt es sich um einen solchen Spezialfall (einen der einfachsten). Man kann hier die sogenannte Trennung der Variablen verwenden. Dabei erlaubt man, mit den Differentialen $dx$ und $dt$ so wie mit Zahlen zu rechnen. Man sortiert die Terme der Gleichung (6.1) so um, dass alle Terme, die nur von $x$ abhängen auf der einen Seite stehen, während alle Terme, die nur von $t$ abhängen auf die andere Seite kommen.

[1] nach [Son01]

Der Proportionalitätsfaktor hängt weder von $x$ noch von $t$ ab und kann auf einer beliebigen Seite der Gleichung stehen. Wendet man dieses Verfahren auf (6.1) an, so erhält man

$$\frac{dx}{x} = r\ dt.$$

Nun integriert man beide Seiten dieser Gleichung unbestimmt. Beim Integrieren erhält man auf beiden Seiten Integrationskonstanten, die zu einer einzigen Konstanten zusammengefasst werden

$$\int \frac{dx}{x} = \int r\ dt \quad \Longrightarrow \quad \ln|x| = rt + K,\ \ K \in \mathbb{R}.$$

Da uns die Population $x(t)$ interessiert, muss man nach $x(t)$ auflösen. Dafür wendet man auf beiden Seiten der letzten Gleichung die Exponentialfunktion an. Man erhält

$$x(t) = Ce^{rt}, \quad \text{mit } C \in \mathbb{R}.$$

Die Konstante $C$ kann für das konkrete Anfangswertproblem (6.1), (6.2) mit Hilfe der Anfangsbedingung festgelegt werden. Aus (6.2) folgt

$$x_0 = x(0) = C.$$

Damit lautet die Lösung des Anfangswertproblems (6.1), (6.2)

$$x(t) = x_0 e^{rt}. \tag{6.3}$$

Das Verhalten der Lösung hängt natürlich vom Proportionalitätsfaktor $r$ ab. Man kann drei Fälle unterscheiden:

1. $r = 0$. Das heißt, es gibt kein Wachstum. Man erhält aus (6.3) $x(t) = x_0$ für alle Zeiten. Die Anzahl der Lebewesen in der Population verändert sich nicht. Stirbt eines, wird gleichzeitig ein neues geboren und umgekehrt.
2. $r > 0$. Man hat positives Wachstum. Es gilt

   $$\lim_{t\to\infty} x(t) = \infty,$$

   das heißt die Anzahl der Lebewesen der Population wächst unbeschränkt.
3. $r < 0$. Man hat negatives Wachstum. In dem Fall folgt aus (6.3)

   $$\lim_{t\to\infty} x(t) = 0,$$

   das bedeutet, dass die Population für $t \to \infty$ ausstirbt.

Diese drei Fälle sind in Abbildung 6.1 illustriert.

Das einfache Modell (6.1), (6.2) erweist sich als nicht sehr realistisch. Es spiegelt bekannte Zusammenhänge nicht wider. Zum Beispiel vermehren sich Bakterien in einer Petrischale nicht mehr so gut, wenn die Population eine gewisse Größe erreicht hat, weil beispielsweise Knappheit an Nahrung herrscht. Andererseits ist das Wachstum größer, wenn genügend Platz in der Schale ist. In einem realistischen Modell würde daher positives Wachstum nur bis zu einer gewissen Grenze existieren. Dann würde die Sterblichkeitsrate der Population überwiegen, das heißt negatives Wachstum, bis irgendwann wieder positives Wachstum einsetzen kann, und so weiter.

### 6.1.2 Ein diskretes Modell

Bevor ein realistischeres Modell eingeführt wird, betrachten wir noch eine diskrete Variante des einfachen Modells (6.1), (6.2). Diskret bedeutet, dass die Zeit nicht

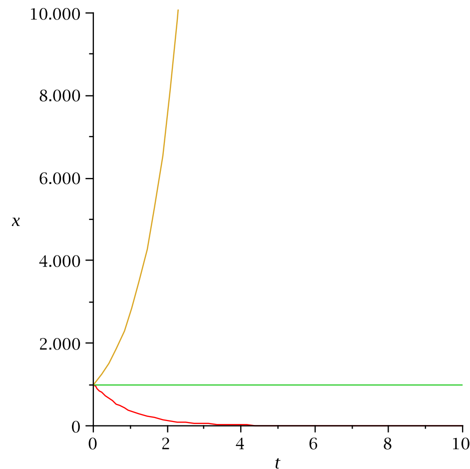


Abbildung 6.1: Lösungen (6.3) des einfachen Wachstumsmodells für unterschiedliche Wachstumsraten, $x_0 = 1000$.

mehr als kontinuierliche Variable betrachtet wird, sondern nur noch gewisse Zeitpunkte

$$0, \Delta t, 2\Delta t, 3\Delta, \ldots$$

mit $\Delta t > 0$ betrachtet werden (wie auf einer digitalen Uhr). Da die Zeit nicht mehr kontinuierlich ist, kann man nicht mehr ableiten. Man muss die Ableitung in (6.1) geeignet ersetzen. Das geschieht mit Hilfe des Differenzenquotienten

$$\frac{\Delta x}{\Delta t} := \frac{x(t + \Delta t) - x(t)}{\Delta t}. \tag{6.4}$$

Ist $x(t)$ zweimal stetig differenzierbar, so erhält man aus der Taylor–Entwicklung

$$x(t + \Delta t) = x(t) + \Delta t \frac{dx}{dt}(t) + \mathcal{O}\left((\Delta t)^2\right).$$[2]

Durch Umstellung ergibt sich

$$\frac{\Delta x}{\Delta t} = \frac{dx}{dt}(t) + \mathcal{O}(\Delta t).$$

Das bedeutet, dass der Differenzenquotient (6.4) eine Approximation erster Ordnung an die Ableitung ist.

Wir wählen die Bezeichnung $x^{(0)} = x(0), x^{(1)} = x(\Delta t)$ und so weiter. Der Einfachheit halber betrachten wir $\Delta t = 1$, was man immer durch eine geeignete Entdimensionierung des realen Problems erreichen kann. Wählt man auf der rechten Seite von (6.1) die Anzahl der Lebewesen der Population vom vorangegangenen Zeitpunkt $x^{(n)}$ , so erhält zur Berechnung der Anzahl der Lebewesen zum nächsten Zeitpunkt die Differenzengleichung 1. Ordnung

$$x^{(n+1)} - x^{(n)} = rx^{(n)}, \quad n = 0, 1, 2, \ldots \tag{6.5}$$

$$x^{(0)} = x_0. \tag{6.6}$$

Durch Umstellen erhält man

$$x^{(n+1)} = (1 + r)x^{(n)}.$$

[2]Eine Funktion $f(t)$ ist $\mathcal{O}(t)$, wenn es eine Konstante $C > 0$ gibt, so dass $|f(t)| \leq C\,|t|$ gilt, für $|t|$ hinreichend klein.

Rekursives Einsetzen ergibt

$$x^{(n+1)} = (1+r)x^{(n)} = (1+r)^2 x^{(n-1)} = \ldots = (1+r)^{n+1} x^{(0)}. \tag{6.7}$$

Mit (6.7) lässt sich das Lösungsverhalten von Modell (6.5), (6.6) studieren:

1. $r > 0$. In diesem Fall wächst die Anzahl der Lebewesen immer noch unbeschränkt.
2. $r = 0$. Auch hier ändert sich nichts im Vergleich zum kontinuierlichen Modell (6.1), (6.2). Die Anzahl der Lebewesen bleibt konstant.
3. $-2 < r < 0$. In diesem Fall gilt $\lim_{n\to\infty} x^{(n)} = 0$, die Population stirbt also aus.
4. $r = -2$. In diesem Fall alterniert die Anzahl der Lebewesen zwischen $x^{(0)}$ und dem negativen Wert $-x^{(0)}$. Dieser Fall ist unrealistisch.
5. $r < -2$. In diesem Fall existiert kein eigentlicher Grenzwert. Es treten jedoch negative Anzahlen von Lebewesen auf, was wiederum in der Realität nicht vorkommt.

Insgesamt stellt man fest:

1. Das Lösungsverhalten des kontinuierlichen und des diskreten Modells unterscheidet sich (hier für $r \leq -2$),
2. Das Lösungsverhalten des diskreten Modells ist auch nicht realistischer als das Lösungsverhalten des kontinuierlichen Modells.

### 6.1.3 Numerische Verfahren zur Lösung des Kontinuumsmodells

Wir haben gesehen, dass Kontinuumsmodelle physikalischer Prozesse zu Gleichungen führt, in denen Funktionen gesucht sind. Sind in diesen Gleichungen Ableitungen der gesuchten Funktion enthalten, spricht man von Differentialgleichungen. Handelt es sich bei den Funktionen um skalare Funktionen einer Veränderlichen $u : (a,b) \to \mathbb{R}$, so spricht man von gewöhnlichen Differentialgleichungen. Diese werden im Laufe des Studiums noch ausführlich behandelt.

Die Herleitung von gewöhnlichen Differentialgleichungen durch Modellierung ist ein Teil der Beschreibung von Naturvorgängen, siehe (6.1),(6.2). Der zweite Teil besteht darin, diese Gleichungen zu lösen. Das geht im allgemeinen nicht so einfach wie im Abschnitt 6.1.1.

Der einfachste Typ einer gewöhnlichen Differentialgleichung wurde bereits in der Schule behandelt: Gegeben ist eine Funktion $f : (a,b) \to \mathbb{R}$. Gesucht ist eine Funktion $u : (a,b) \to \mathbb{R}$, so dass

$$u'(x) = f(x)$$

gilt. Die allgemeine (abstrakte) Lösung ist das unbestimmte Integral

$$u(x) = \int f(x)\, dx. \tag{6.8}$$

Bekanntes aus der Schule über das Integral:

- Es gibt Integrationsregeln, die man probieren kann (Substitutionen, partielle Integration).
- Diese funktionieren jedoch nur bei speziellen Funktionen.
- Mathematische Software kann weiterhelfen (MAPLE, MATHEMATICA, ...).
- *Integration im allgemeinen kompliziert !*

**Beispiel 6.1** Gesucht ist die Stammfunktion von $f(x) = \sqrt{x + \sqrt{x}}$. Man erhält mit MAPLE

$$u(x) = \int \sqrt{x + \sqrt{x}}\, dx = \left(\frac{2}{3}x^{5/4} + \frac{1}{6}x^{3/4} - \frac{1}{4}x^{1/4}\right)\sqrt{\sqrt{x}+1} + \frac{1}{4}\text{arsinh}\left(x^{1/4}\right).$$

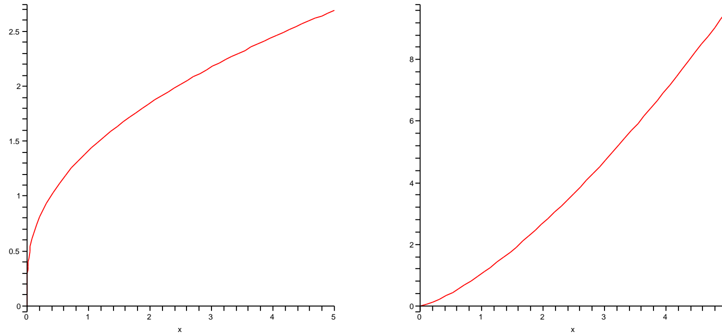


Abbildung 6.2: Integrand und Stammfunktion zum Beispiel 6.1.

□

**Beispiel 6.2** Gesucht ist die Stammfunktion von $f(x) = \sqrt{x^2 + \sqrt{x}}$. Man erhält mit MAPLE

$$u(x) = \int \sqrt{x^2 + \sqrt{x}}\, dx = \frac{4}{5}x^{5/4}\text{hypergeom}\left(\left[\frac{-1}{2}, \frac{5}{6}\right], \left[\frac{11}{6}\right], -x^{3/2}\right)$$

Die Stammfunktion kann nur durch eine spezielle Funktion, die sogenannte hypergeometrische Funktion, dargestellt werden !

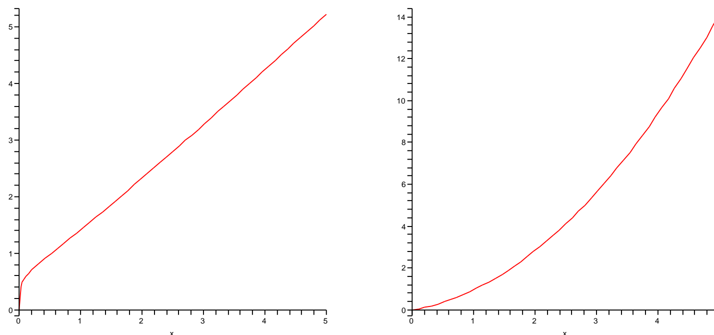


Abbildung 6.3: Integrand und Stammfunktion zum Beispiel 6.2.

□

**Beispiel 6.3** Gesucht ist die Stammfunktion von $f(x) = \sqrt{x + 1 + \sqrt{x}}$. Bei diesem Integranden hilft auch MAPLE nicht weiter. Trotzdem möchte man eine Vorstellung von einer Stammfunktion haben. Dazu dienen numerische Verfahren.

□

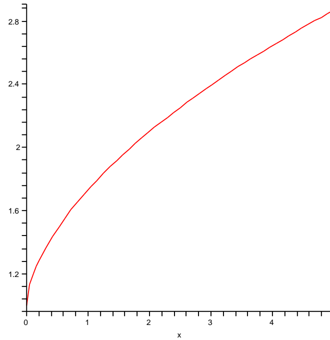


Abbildung 6.4: Integrand zum Beispiel 6.3.

Schon beim unbestimmten Integral gibt es Fälle, wo man die Lösung nicht analytisch findet. Bei Differentialgleichungen ist das der allgemeine Fall. Es gibt nur wenige, einfache Typen, die eine geschlossene analytische Darstellung der Lösung ermöglichen, wie (6.1), (6.2).

Ein Beispiel für eine gewöhnliche Differentialgleichung, die analytisch nicht auflösbar ist, ist

$$u'(x) = x^2 + u^2(x). \tag{6.9}$$

Man kann zeigen, dass eine Lösung dieser Differentialgleichung existiert, aber dass diese Lösung nicht mit elementaren Funktionen und Integration darstellbar ist. In solchen Fällen helfen nur numerische Verfahren zur Approximation der Lösung.

Betrachte die allgemeine gewöhnliche Differentialgleichung 1. Ordnung

$$u'(x) = f(x, u(x)) \quad x \in (a, b), \quad u(a) = u_0. \tag{6.10}$$

Das einfachste numerische Verfahren zur Approximation der Lösung von (6.10) ist das explizite Euler[3]–Verfahren. Zunächst zerlegt man $[a, b]$ in $n$ (gleich große) Teilintervalle mit den Punkten

$$a = x_1 < x_2 < \ldots < x_n < x_{n+1} = b, \quad x_i - x_{i-1} = h,$$

siehe Abbildung 6.5. Die numerische Approximation der Lösung wird mit $u^h$ bezeichnet.

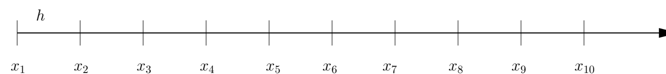


Abbildung 6.5: Zerlegung des Intervalls für numerische Verfahren.

Man kennt

- den Funktionswert von $u$ in $x_1 : u(x_1) = u_0$,
- die Ableitung von $u$ in $x_1 : u'(x_1) = f(x_1, u(x_1))$.

Die Idee besteht nun darin, in Richtung dieser Ableitung bis $x_2$ zu gehen, wobei man den Funktionswert auf dieser Geraden als Approximation für $u(x_2)$ nimmt

$$u^h(x_2) := u(x_1) + hf(x_1, u(x_1)),$$

siehe Abbildung 6.6. Dabei macht man im allgemeinen einen Fehler : $u^h(x_2) \neq u(x_2)$ !

[3] Leonhard Euler (1707 – 1783)

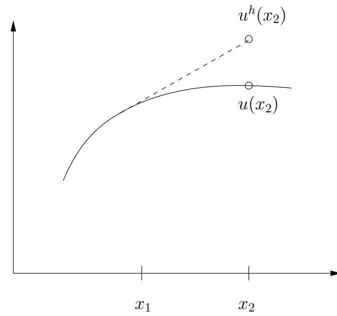


Abbildung 6.6: Prinzip des expliziten Euler–Verfahrens.

Man fährt nach dem gleichen Prinzip fort und erhält das explizite Euler–Verfahren

$$\begin{aligned} u^h(x_1) &= u(x_1), \\ u^h(x_i) &= u^h(x_{i-1}) + hf(x_{i-1}, u^h(x_{i-1})), \quad i = 2, \ldots, n+1. \end{aligned} \tag{6.11}$$

Ein anderes Verfahren, das sogenannte implizite Euler–Verfahren, erhält man, wenn man anstelle des bekannten Anstieges $f(x_{i-1}, u^h(x_{i-1}))$ den unbekannten Anstieg $f(x_i, u^h(x_i))$ nimmt

$$\begin{aligned} u^h(x_1) &= u(x_1), \\ u^h(x_i) &= u^h(x_{i-1}) + hf(x_i, u^h(x_i)), \quad i = 2, \ldots, n+1. \end{aligned} \tag{6.12}$$

Bei diesem Verfahren muss man zur Berechnung von $u^h(x_i)$ im allgemeinen eine nichtlineare Gleichung lösen. Das ist teurer als das explizite Euler–Verfahren.

Aus mathematischer Sicht muss man folgende Fragen zu den Verfahren untersuchen, siehe spätere Vorlesungen:

- Funktionieren die Verfahren immer ? Wenn nicht, unter welchen Bedingungen funktionieren sie ?
- Wie genau sind die Ergebnisse ?
- Wie schnell sind die Berechnungen ?
- Wie verändern sich die Ergebnisse, wenn man das Gitter verändert ?
- Gibt es bessere Verfahren, das heißt, Verfahren die genauer auf dem gleichen Gitter bei vergleichbarem Aufwand sind ?

**Beispiel 6.4** Wir betrachten eine Gleichung vom Typ (6.9)

$$u'(x) = x^2 + u^2(x), \quad u(0) = 0 \quad \text{in } [0,1].$$

Der Iterationsschritt beim expliziten Euler–Verfahren lautet

$$u^h(x_i) = u^h(x_{i-1}) + h\left(x_{i-1}^2 + (u^h(x_{i-1}))^2\right)$$

und beim impliziten Euler–Verfahren

$$u^h(x_i) = u^h(x_{i-1}) + h\left(x_i^2 + (u^h(x_i))^2\right).$$

In jedem Schritt des impliziten Euler–Verfahrens muss man eine quadratischen Gleichung lösen

$$0 = h(u^h(x_i))^2 - u^h(x_i) + \left(u^h(x_{i-1}) + hx_i^2\right).$$

*MATLAB–DEMO*

□

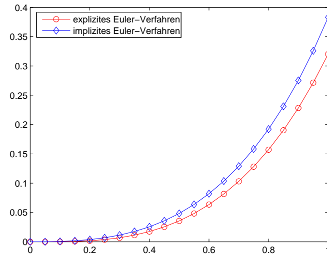


Abbildung 6.7: Approximation der Lösung von Beispiel 6.4 in $[0, 1]$, $h = 0.05$.

## 6.2 Ein realistischeres Modell

Bei einem realistischeren Modell muss zum einen die Population, wenn sie zu groß wird, wegen Überbevölkerung wieder kleiner werden. Wenn zum anderen die Population eine gewisse Schranke unterschreitet, dann ist wieder genug Platz zum Wachsen da und die Population muss wieder größer werden.

### 6.2.1 Ein Kontinuumsmodell

Ein kontinuierliches Modell, in welchem versucht wird, den obigen Anforderungen gerecht zu werden, hat die Gestalt

$$\frac{dx}{dt} = rx(L - x), \quad r, L \in \mathbb{R},\ L > 0, \tag{6.13}$$

$$x(0) = x_0. \tag{6.14}$$

Dabei ist $L$ eine charakteristische Anzahl der Lebewesen in der Population. Betrachte nämlich $r = 1$:

1. Ist $x = L$, dann verschwindet die rechte Seite von (6.13), das bedeutet $dx/dt = 0$, und die Größe der Population ändert sich nicht mehr.
2. Ist $x > L$, dann ist $dx/dt < 0$ und die Population wird kleiner.
3. Ist $x < L$, dann ist $dx/dt > 0$ und die Population wächst.

Die Lösung des Anfangswertproblems (6.13), (6.14) ist

$$x(t) = \frac{Lx_0 e^{Lrt}}{L - x_0 + x_0 e^{Lrt}}, \tag{6.15}$$

was man durch Einsetzen überprüfen kann. Die Lösungen $x(t)$ sind für $r = 1$, $L = 1$ und unterschiedliche Werte von $x_0$ in Abbildung 6.8 dargestellt. Man stellt fest, dass unabhängig vom Anfangswert, die Anzahl der Lebewesen in der Population dem Wert $L = 1$ zustrebt. Man erhält ein Gleichgewicht für die Anzahl der Lebewesen. Das stellt man auch für andere positive Wachstumsraten und andere Werte von $L$ fest. Die Populationsdynamik von Modell (6.13), (6.14) entspricht also nicht der Realität.

### 6.2.2 Ein diskretes Modell

Wie beim einfachen Modell wird der Differentialquotient durch den Differenzenquotienten ersetzt. Der Einfachheit halber rechnen wir wieder mit $\Delta t = 1$ und mit $L = 1$. Man erhält aus (6.13), (6.14) die Differenzengleichung

$$x^{(n+1)} = x^{(n)} + rx^{(n)}\left(1 - x^{(n)}\right).$$

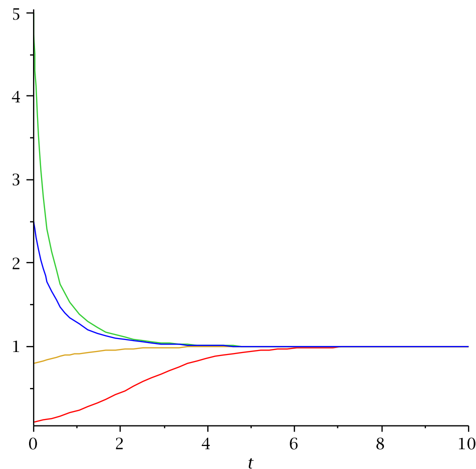


Abbildung 6.8: Lösungen (6.15) des realistischen Wachstumsmodells für $L = 1$, $r = 1$ und unterschiedliche Anfangswerte $x_0$.

Durch Umstellen erhält man

$$x^{(n+1)} = (1+r)x^{(n)} - r\left(x^{(n)}\right)^2, \quad n = 0, 1, 2, \ldots \tag{6.16}$$

$$x^{(0)} = x_0. \tag{6.17}$$

Diese Gleichung nennt man logistische Differenzengleichung.

Zur Lösung der logistischen Differenzengleichung verwendet man am besten einen Computer. Wir verwenden als Anfangsbedingung $x_0 = 0.1$ und experimentieren mit verschiedenen Werten der Wachstumsrate $r$. Die Ergebnisse sind in Abbildung 6.9 dargestellt. Die diskreten Ergebnisse sind dabei durch Linien verbunden. Für $r = 1.5$ unterscheidet sich die Lösung des diskreten Modells nicht wesentlich von der Lösung des kontinuierlichen Modells. Das ist jedoch für $r = 2.5$ schon anders. Man erhält für diese Wachstumsrate eine oszillierende Lösung mit konstanter Amplitude. Erhöht man die Wachstumsrate weiter, verliert man noch die Regelmäßigkeit der Lösung. Für $r = 3$ erhält man eine chaotisch oszillierende Lösung. Diese Lösung entspricht den Erwartungen an die Veränderungen der Anzahl von Lebewesen in eine Population.

In Abbildung 6.9 sieht man, dass unterschiedliche Wachstumsraten $r$ zu qualitativ vollkommen unterschiedlichen Lösungen führen. Um das Verhalten der Lösung bezüglich der Wachstumsrate näher zu studieren, sind in Abbildung 6.10 die Iterierten $x^{(5000)}, \ldots x^{(5120)}$ der logistischen Differenzengleichung für $r \in [1.9, 3]$ eingezeichnet. Man sieht:

1. Bis etwa $r = 2$ bekommt man nur die stationäre Lösung.
2. Danach treten Lösungen auf, bei denen sich immer zwei Werte abwechseln (oszillierende Lösungen mit konstanter Amplitude).
3. Ab etwa $r = 2.45$ treten oszillierende Lösungen mit vier Werten auf.
4. Ab etwa $r = 2.55$ treten oszillierende Lösungen mit acht Werten auf.
5. Für größere Werte von $r$ sieht das Verhalten im Bild chaotisch aus. Das dem nicht so ist, zeigt der Ausschnitt auf der rechten Seite von Abbildung 6.10. Dieser Ausschnitt sieht ähnlich wie das Gesamtbild aus. Die kleinen Skalen verhalten sich offenbar ähnlich wie die großen Skalen.

Das diskrete realistischere Modell ist wesentlich reicher strukturiert als das kontinuierliche realistischere Modell. Für hinreichend große Wachstumsraten erhält man

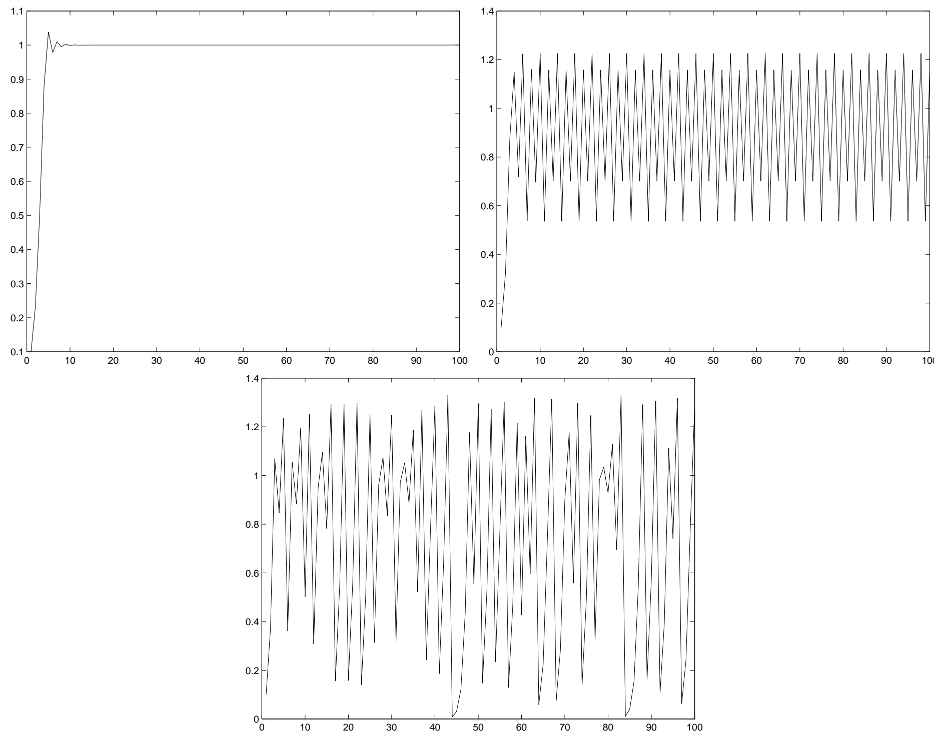


Abbildung 6.9: Lösungen von (6.16), (6.17) mit $x_0 = 0.1$ und $r = 1.5$, $r = 2.5$ und $r = 3$ (von links nach rechts und oben nach unten).

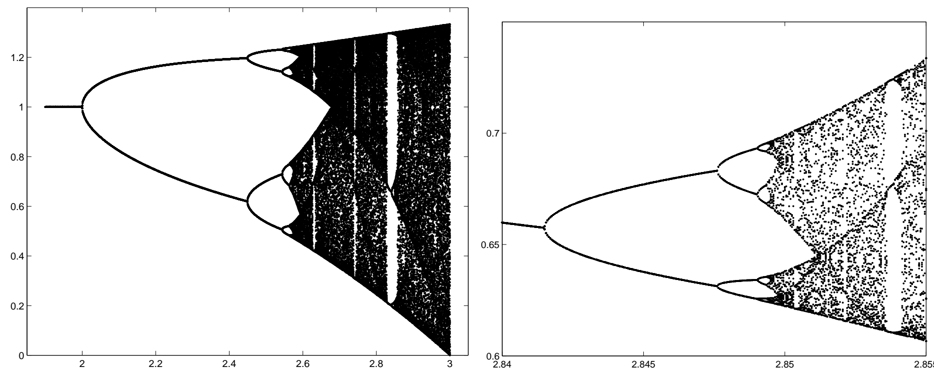


Abbildung 6.10: Lösungen von (6.16), (6.17) mit $x_0 = 0.1$ und $r \in [1.9, 3]$ (links), rechts Ausschnitt.

Lösungen, die den Vorstellungen an die Entwicklung der Anzahl von Lebewesen in eine Population entsprechen.

**Bemerkung 6.5** Die logistische Differenzengleichung (6.16), (6.17) kann auch als explizites Euler–Verfahren des kontinuierlichen Modells (6.13), (6.14) aufgefasst werden. In späteren Vorlesungen wird gezeigt werden, dass das explizite Euler–Verfahren in bestimmten Situationen instabil ist, siehe auch in den Übungen. Die logistische Differenzengleichung ist gerade eine solche Situation. Deshalb gibt es solch große Unterschiede zwischen dem kontinuierlichen und dem diskreten Modell. $\square$

# Kapitel 7

# Wärmeleitung

Dieses[1] Kapitel behandelt die mathematische Modellierung von Wärmeleitungsprozessen. Es werden die Grundprinzipien der Thermodynamik und wichtige Konzepte, wie Diffusion und Konvektion, eingeführt.

## 7.1 Thermodynamik

Das Grundkonzept der Thermodynamik ist jenes der Wärme, das einer ungeordneten Bewegung von Molekülen entspricht. Dieser Bewegung ist eine kinetische Energie zugeordnet, die als Wärmeenergie bezeichnet wird. Die Temperatur ist ein lineares Maß für den Mittelwert dieser Energie. Seien $m$ $[kg]$ die Masse der Moleküle und $v$ $[m/s]$ der Betrag ihrer Geschwindigkeit, dann ist der Druck $p$ $[N/m^2]$ (Kraft pro Fläche) durch

$$pV = \frac{2}{3} N \overline{E_{\text{kin}}} = \frac{2}{3} N \left( \frac{m}{2} \overline{v^2} \right)$$

beschrieben, wobei $V$ $[m^3]$ das Volumen, $N$ die Anzahl der Teilchen und $\overline{E_{\text{kin}}}$ $[J] = [Nm]$ die mittlere kinetische Energie der Teilchen bezeichnen. Verwendet man die Zustandsgleichung für das ideale Gas

$$pV = NkT$$

mit der Temperatur $T$ $[K]$ und der Boltzmann[2]–Konstanten $k = 1.38\ 10^{-23}\ J/K$, so ergibt sich

$$T = \frac{m}{3k} \overline{v^2}.$$

Die wichtigen Konzepte der Thermodynamik sind energetischer Natur:

- Die innere Energie $U$ $[J]$ bezeichnet die kinetische Energie der Teilchen des betrachteten Systems, die Energie der chemischen Bindungen der Teilchen des Systems und ähnliche Effekte.
- Die Enthalpie $H$ $[J]$ ist die Summe aus innerer Energie und Volumenarbeit, das heißt

$$H = U + pV.$$

Die Erhaltung der Energie wird im ersten Hauptsatz der Thermodynamik beschrieben. Dieser besagt, dass die Änderung $\Delta U$ der inneren Energie gleich der Summe aus zugeführter Wärmemenge $\Delta Q$ und geleisteter Arbeit $-\Delta W$ ist. Da die Arbeit durch $W = pV$ gegeben ist, folgt

$$H = U + W.$$

[1] nach [Bur07], [Wla72]

[2] Ludwig Boltzmann (1844 – 1906)

Dann kann die Energieerhaltung als

$$\Delta\,(U+W) = \Delta H = \Delta Q \tag{7.1}$$

geschrieben werden.

Um die Unordnung im System zu beschreiben, verwendet man die Entropie $S$ $[J/K]$, siehe zweiter Hauptsatz der Thermodynamik. Die Entropie ist durch die Relation

$$\Delta Q = -T\Delta S$$

beschrieben. Das bedeutet, die Entropie ist gleich der zugeführten Wärmemenge pro Temperatur. Nach (7.1) ist

$$\Delta S = -\frac{\Delta H}{T}.$$

Der zweite Hauptsatz der Thermodynamik besagt nun, dass bei einem reversiblen Prozess $\Delta S = 0$ gilt und bei einem irreversiblen Prozess $\Delta S > 0$.

## 7.2 Transport

Im Rahmen der kinetischen Gastheorie kann Wärmeleitung als Energietransport durch die Teilchen interpretiert werden. Neben der Energie, können auch Masse und Impuls transportiert werden. Diese Effekte sind bei Strömungen von Interesse.

Der Einfachheit halber betrachten wir als Gebiet einen Stab, der als eindimensionale Strecke $\Omega = (a,b)$ modelliert wird. Zur makroskopischen Beschreibung der Wärmeleitung durch Transport werden die kontinuierlichen Dichten, $h$ für die Enthalpie und $u$ für die Temperatur, als Funktionen für positive Zeiten

$$h, u \;:\; \mathbb{R}^+ \times \Omega \to \mathbb{R}^+$$

betrachten. Wir sagen zu den Dichten auch kurz Enthalpie beziehungsweise Temperatur.

Die Grundlage der Modellierung ist der erste Hauptsatz der Thermodynamik. Man betrachtet ein beliebiges Teilgebiet $\omega \subset \Omega$, $\omega = (\alpha,\beta)$. Dann ist die zeitliche Änderung der Enthalpie in $\omega$ gleich der zugeführten Wärmemenge, (7.1). Wärmezufuhr (auch negative) kann durch in $\omega$ befindliche Wärmequellen oder durch Wärmefluss über den Rand von $\omega$ erfolgen. Die Wärmequellen in $\Omega$ werden durch die Dichte $f(t,x) \;:\; \mathbb{R}^+ \times \Omega \to \mathbb{R}$ und der Wärmefluss wird durch den „Flussvektor“ $q \;:\; \mathbb{R}^+ \times \Omega \to \mathbb{R}$ beschrieben. Man erhält

$$\frac{d}{dt}H(t,\omega) = \frac{d}{dt}\int_\omega h(t,x)\;dx = \int_\omega f(t,x)\;dx + q(t,\beta) - q(t,\alpha). \tag{7.2}$$

Die Formel der partiellen Integration liefert

$$\int_\omega \frac{\partial}{\partial x}q(t,x)\;dx = q(t,\beta) - q(t,\alpha).$$

Einsetzen dieser Beziehung in (7.2), Vertauschung von Differentiation nach der Zeit und Integration im Ort sowie Umstellen ergibt

$$\int_\omega \left(\frac{\partial}{\partial t}h(t,x) - \frac{\partial}{\partial x}q(t,x) - f(t,x)\right)\;dx = 0.$$

Diese Beziehung gilt für ein beliebig gewähltes Teilgebiet $\omega$. Das kann nur der Fall sein, wenn der Integrand gleich Null ist, also

$$\frac{\partial h}{\partial t} - \frac{\partial q}{\partial x} = f \quad \text{in } (0,t)\times(a,b). \tag{7.3}$$

Gleichung (7.3) wird Transportgleichung genannt. Die rechte Seite $f$ ist eine bekannte Funktion, welche die Dichten der Wärmequellen beschreibt. Die Funktionen $h$ und $q$ sind unbekannt. In der Form (7.3) ist die Beschreibung des Wärmetransports unabhängig von der Temperatur. Man benötigt nun noch Materialgesetze, die noch eine Relation zur Temperatur herstellen.

## 7.3 Materialgesetze

Die Beziehung zwischen der Enthalpie und der Temperatur kann in vielen Fällen als linear modelliert werden

$$h(t, x) = \rho c u(t, x), \tag{7.4}$$

wobei $\rho$ $[kg/m^3]$ die Dichte und $c$ $[J/(kg\ K)] = [W\ s/(kg\ K)]$ die spezifische Wärmekapazität des betrachteten Materials sind. Im einfachsten Fall sind $\rho$ und $c$ Konstanten. In manchen Situationen ist es aber wichtig, diese Größen als veränderlich zu betrachten, beispielsweise $\rho = \rho(x, u)$, $c = c(x, u)$. Das tritt beispielsweise auf, wenn man eine Mischung mehrerer Materialen zu modellieren hat, die unterschiedliche Dichten und Wärmekapazitäten besitzen. Die effektive Dichte und Wärmekapazität sind dann dann ortsabhängige Funktionen, die durch das Material an der jeweiligen Position bestimmt sind. Ein anderer Fall ist, dass manche Materialen sich start ausdehnen, wenn die Temperatur ansteigt. Dann verändert sich deren Dichte und es ist wichtig, $\rho = \rho(u)$ zu betrachten.

Die Beziehung zwischen dem Wärmefluss $q$ und der Temperatur $u$ wird im allgemeinen durch Diffusion bestimmt. Das bedeutet, die Teilchen bewegen sich (mikroskopisch mittels einer Brown[3]'schen Bewegung) bevorzugt in die Richtung des stärksten Temperaturgefälles, um lokale Schwankungen der Temperatur auszugleichen. Das lokal stärkste Temperaturgefälle kann mit Hilfe der ersten Ableitung bestimmt werden. Es wird durch das sogenannte Fick[4]'sche Gesetz oder Fourier[5]'sche Abkühlungsgesetz modelliert

$$q(t, x) = \lambda \frac{\partial}{\partial x} u(t, x), \tag{7.5}$$

wobei $\lambda > 0$ $[W/(m\ K)]$ den Wärmeleitkoeffizienten bezeichnet. Die spezielle Modellierung von $\lambda$ hängt wieder von der betrachteten Situtation ab. Im allgemeinen wird $\lambda = \lambda(x, u)$ sein, manchmal muss man auch eine Abhängigkeit vom Temperaturgefälle $\frac{\partial u}{\partial x}$ berücksichtigen.

## 7.4 Die Wärmeleitungsgleichung

Wir betrachten der Einfachheit halber den Fall konstanter, skalarer Werte von $\rho, c$ und $\lambda$. Dann erhält man mit dem Einsetzen von (7.4) und (7.5) in (7.3) die lineare Differentialgleichung

$$\frac{\partial u}{\partial t} - D \frac{\partial^2 u}{\partial x^2} = f \quad \text{in } \mathbb{R}^+ \times (a, b), \tag{7.6}$$

wobei $D = \lambda/(c\rho)$ $[m^2/s]$ der Temperatur–Leitwert ist.

Man weiss aus der Theorie zu Gleichungen der Gestalt (7.6), dass die Lösung dann eindeutig bestimmt ist, wenn man für die Temperatur des Systems am Anfang,

[3]Robert Brown (1773 – 1858)
[4]Adolf Fick (1829 – 1901)
[5]Jean Baptiste Joseph Fourier (1768 – 1830)

das heißt zum Zeitpunkt $t = 0$ und zusätzlich auf dem Rand, dass heißt in den Punkten $a$ und $b$, geeignete Bedingungen für alle Zeiten vorgibt. Die Anfangsbedingung hat die Form

$$u(0, x) = u_0(x), \quad x \in (a, b),$$

für eine gegebene Anfangstemperatur $u_0(x)$.

Randbedingungen kann man unterschiedlich vorgeben. Dazu betrachten wir den Wärmefluss über den Rand und nehmen an, dass außerhalb von $(a, b)$ eine Umgebungstemperatur $u^*$ gegeben ist. Im allgemeinen erfolgt die Wärmeübertragung mit der Umgebung durch Strömung (Konvektion). Dabei wird die Wärme in ein oder aus einem Fluid beziehungsweise Gas übertragen, indem das Fluid beziehungsweise Gas die Oberfläche eines anderen Volumens überströmt und dabei ein Temperaturausgleich erfolgt. Da der Wärmefluss über den Rand die Temperaturdifferenz ausgleichen muss, erhält man, zum Beispiel im Punkt $b$

$$q(t, b) = -\alpha \left(u - u^*\right)(t, b)$$

mit einem positiven Wärmeübergangskoeffizienten $\alpha = \alpha(x, u, u^*)$. Ersetzt man den Wärmefluss durch die Temperatur, (7.5), erhält man die sogenannte Robin[6]–Randbedingung

$$\lambda \frac{\partial u}{\partial x}(t, b) = -\alpha \left(u - u^*\right)(t, b).$$

Interessant sind die Grenzwerte von $\beta = \alpha/\lambda$:

- Für $\beta \to 0$ erhält man die sogenannte homogene Neumann[7]–Randbedingung $\frac{\partial u}{\partial x}(t, b) = 0$. Diese Randbedingung besagt, dass kein Austauch von Wärme mit der Umgebung erfolgt. Dies ist bei einem isolierten Rand der Fall.
- Für $\beta \to \infty$ erhält man die Dirichlet[8]–Randbedingung $u(t, b) = u^*(t, b)$. Diese Randbedingung besagt, dass der Wärmeaustausch mit der Umgebung so stark ist, dass sich die Temperatur am Rand des Stabs der Umgebungstemperatur anpasst.

Im Punkt $a$ sind die Dirichlet–Randbedingung $u(t, a) = u^*(t, a)$, die homogene Neumann–Randbedingung $\frac{\partial u}{\partial x}(t, a) = 0$ und die Robin–Randbedingung $\lambda \frac{\partial u}{\partial x}(t, a) = \alpha \left(u - u^*\right)(t, a)$.

Man beachte, dass man für $f = 0$, das heißt man hat keine Temperaturquellen und –senken, und im Fall eines isolierten Randes ein abgeschlossenes System erhält, in dem die Energieerhaltung gilt

$$\frac{d}{dt} H(t, (a, b)) = \int_a^b \frac{\partial h}{\partial t}(t, x)\ dx = q(b) - q(a) = 0,$$

vergleiche (7.2).

Nun kann man die Wärmeleitungsgleichung (7.6) skalieren und in eine dimensionslose Form bringen. Man wählt eine typische Länge $l$ für das Gebiet und eine zunächst noch unbestimmte Zeitskala $\tau$ und transformiert die Variablen zu

$$\tilde{x} = \frac{x}{l}, \quad \tilde{t} = \frac{t}{\tau}.$$

Als nächstes wird die Temperatur mittels einer Abschätzung $T_0$ für die auftretende Minimaltemperatur und einer Abschätzung $\Delta T$ für die Temperaturschwankung transformiert

$$\tilde{u} = \frac{u - T_0}{\Delta T}.$$

[6]Gustave Robin (1855 – 1897)
[7]Carl Gottfried Neumann (1832 – 1925)
[8]Johann Peter Gustav Lejeune Dirichlet (1805 – 1859)

Mit der Kettenregel

$$\begin{aligned}
\frac{\partial \tilde{u}}{\partial \tilde{t}} &= \frac{\partial}{\partial t}\left(\frac{u-T_0}{\Delta T}\right)\frac{\partial t}{\partial \tilde{t}} = \frac{\tau}{\Delta T}\frac{\partial u}{\partial t}, \\
\frac{\partial \tilde{u}}{\partial \tilde{x}} &= \frac{\partial}{\partial x}\left(\frac{u-T_0}{\Delta T}\right)\frac{\partial x}{\partial \tilde{x}} = \frac{l}{\Delta T}\frac{\partial u}{\partial x}, \\
\frac{\partial^2 \tilde{u}}{\partial \tilde{x}^2} &= \frac{\partial}{\partial x}\left(\frac{l}{\Delta T}\frac{\partial u}{\partial x}\right)\frac{\partial x}{\partial \tilde{x}} = \frac{l^2}{\Delta T}\frac{\partial^2 u}{\partial x^2},
\end{aligned}$$

erhält man nun aus (7.6) die skalierte Wärmeleitungsgleichung

$$\frac{\partial \tilde{u}}{\partial \tilde{t}} = \frac{\tau}{\Delta T}\frac{\partial u}{\partial t} = \frac{\tau}{\Delta T}\left(D\frac{\partial^2 u}{\partial x^2} + f\right) = \frac{\tau}{\Delta T}\frac{\Delta T}{l^2}D\frac{\partial^2 \tilde{u}}{\partial \tilde{x}^2} + \tilde{f} = \frac{D\tau}{l^2}\frac{\partial^2 \tilde{u}}{\partial \tilde{x}^2} + \tilde{f} \qquad (7.7)$$

mit $\tilde{f} = \tau f / \Delta T$. Die Randbedingung skaliert sich zu

$$\begin{aligned}
\frac{\partial}{\partial \tilde{x}}\tilde{u}\left(\frac{t}{\tau}, \frac{b}{l}\right) &= \frac{\alpha l}{\lambda \Delta T}\Big((\Delta T u - T_0) - (\Delta T u^* - T_0)\Big)\left(\frac{t}{\tau}, \frac{b}{l}\right) \\
&= -\frac{\alpha l}{\lambda}\left(\tilde{u} - \tilde{u}^*\right)\left(\frac{t}{\tau}, \frac{b}{l}\right)
\end{aligned}$$

und die Anfangsbedingung zu $\tilde{u}(\tilde{x}, 0) = \tilde{u}_0(\tilde{x})$.

In der Wärmeleitungsgleichung (7.7) gibt es noch zwei effektive Parameter, $\tau$ und $l$. Es liegt nun nahe, die Zeitskala $\tau$ so zu wählen, dass der dimensionslose Diffusionskoeffizient gleich Eins ist, das heißt $\tau = l^2/D$. Dann folgt aus (7.7)

$$\frac{\partial \tilde{u}}{\partial \tilde{t}} - \frac{\partial^2 \tilde{u}}{\partial \tilde{x}^2} = \tilde{f} = \frac{l^2}{D\Delta T}f \quad \text{in } \mathbb{R}^+ \times \left(\frac{a}{l}, \frac{b}{l}\right). \qquad (7.8)$$

Es verbleibt noch der dimensionslose Wärmeübergangskoeffizient $\beta = \alpha l / \lambda$ in der Randbedingung als Parameter.

**Bemerkung 7.1 Stationäre Wärmeleitungsgleichung.** Falls $\tilde{u}$ zeitlich konstant ist, erhält man die stationäre Wärmeleitungsgleichung

$$-\tilde{u}'' = \tilde{f} \quad \text{in } (a, b). \qquad (7.9)$$

Das ist die sogenannte Poisson[9]-Gleichung. Die homogene Form, das heißt $\tilde{f} = 0$, wird Laplace[10]-Gleichung genannt. Die Gleichung (7.9) kann man im Prinzip durch zweimaliges Integrieren im Ort lösen, währendessen das bei der Gleichung (7.8) nicht mehr funktioniert. □

**Bemerkung 7.2 Modellfehler.** Die Differentialgleichungen (7.8) und (7.9) modellieren die Wärmeausbreitung in einem Stab. Es stellt sich auch hier die Frage, wie gut diese Modelle sind. Der Modellfehler besitzt unter anderem folgende Bestandteile:

- Der Stab ist nicht ein– sondern dreidimensional. Man findet im Prinzip auf die gleiche Art und Weise wie oben die Wärmeleitungsgleichung im dreidimensionalen Gebiet. Im Unterschied zur eindimensionalen Gleichung treten dann Ableitungen in alle drei Raumrichtungen auf, womit man eine sogenannte partielle Differentialgleichung erhält.
- Bei den linearen Ansätzen (7.4) und (7.5) werden Terme höherer Ordnung vernachlässigt.

[9] Siméon Denis Poisson (1781 – 1840)
[10] Pierre Simon Laplace (1749 – 1829)

- Das Material ist im allgemeinen nicht vollständig homogen.
- Die physikalischen Konstanten kennt man nur bis zu einer gewissen Genauigkeit.
- Die Anfangsbedingung kennt man im allgemeinen nur punktweise.
- Die Randbedingungen kann man auch nur zu einer gewissen Genauigkeit steuern.

□

# Teil III

# Die Programmiersprache C

# Kapitel 8

# Einführung

Die[1] Programmiersprache C ist eine der am häufigsten verwendeten Programmiersprachen in Wissenschaft und Technik. Sie ist sehr viel näher an der Maschine (dem Computer) angesiedelt als zum Beispiel MATLAB.

C ist eine Compiler–Sprache. Das Übersetzen der Programme besteht aus zwei Komponenten, dem eigentlichen Compilieren und dem Linken. Der Compiler erzeugt aus dem Quelltext einen für den Rechner lesbaren Objektcode. Der Linker erstellt das ausführbare Programm, indem er in die vom Compiler erzeugte Objektdatei Funktionen (siehe auch Kapitel 12) aus Bibliotheken (Libraries) einbindet. Der Begriff Compiler wird häufig auch als Synonym für das gesamte Entwicklungssystem (Compiler, Linker, Bibliotheken) verwendet.

Diese Vorgehensweise ist in Gegensatz zu MATLAB, wo die Befehle während der Laufzeit eingelesen und dann abgearbeitet werden. Die Herangehensweise von MATLAB ist deutlich langsamer.

Für die Bearbeitung der Übungsaufgaben werden Editor und C–Compiler bereitgestellt. Es handelt sich dabei um frei erhältliche (kostenlose) Programme (`gcc`), die auf LINUX-Betriebssystemen arbeiten. Es handelt sich bei C um eine weitgehend standardisierte Programmiersprache. Jedoch ist C–Compiler nicht gleich C–Compiler. Die Vergangenheit hat gezeigt, dass nicht alle Programme unter verschiedenen Compilern lauffähig sind. Wer mit einem anderen als mit dem `gcc`–Compiler arbeitet, hat unter Umständen mit Schwierigkeiten bei der Kompatibilität zu rechnen.

## 8.1 Das erste C–Programm

Traditionell besteht das erste C–Programm darin, dass die Meldung *Hallo Welt* auf dem Bildschirm ausgegeben werden soll.
*Quelltext:* (HalloWelt.c):

```
/* HalloWelt.c */
#include <stdio.h>

int main()
{
  printf("Hallo Welt \n"); /* "\n new line */
  return 0;
}
```

[1] nach einer Vorlesung von Erik Wallacher

Folgende Strukturen finden wir in diesem ersten einfachen Programm vor:

- *Kommentare*
  werden mit `/*` eingeleitet und mit `*/` beendet. Sie können sich über mehrere Zeilen erstrecken und werden vom Compiler (genauer vom Präprozessor) entfernt.
- *Präprozessordirektiven*
  werden mit `#` eingeleitet. Sie werden vom Präprozessor ausgewertet. Die Direktive `#include` bedeutet, dass die nachfolgende Headerdatei einzufügen ist. Headerdateien haben die Dateinamenendung (Suffix.h). Die hier einzufügende Datei `stdio.h` enthält die benötigten Informationen zur standardmäßigen Ein– und Ausgabe von Daten (standard input/output).
- *Kommentare*
  Das Schlüsselwort `main` markiert den Beginn des Hauptprogramms, das heißt den Punkt, an dem die Ausführung der Anweisungen beginnt. Auf die Bedeutung der Klammer `()` wird später detaillierter eingegangen.
- Syntaktisch (und inhaltlich) zusammengehörende Anweisungen werden in Blöcken zusammengefasst. Dies geschieht durch die Einschließung eines Blocks in geschweifte Klammern:

```
{
  erste Anweisung;
  ...
  letzte Anweisung;
}
```

- Die erste Anweisung, die wir hier kennenlernen, ist `printf()`. Sie ist eine in `stdio.h` deklarierte Funktion, die Zeichenketten (Strings) auf dem Standardausgabegerät (Bildschirm) ausgibt. Die auszugebende Zeichenkette wird in Anführungsstriche gesetzt.
  Zusätzlich wird eine Escapesequenz angefügt: `\n` bedeutet, dass nach der Ausgabe des Textes *Hallo Welt* eine neue Zeile begonnen wird.
  *Anweisungen innerhalb eines Blocks werden mit Semikolon* `;` *abgeschlossen.*

Der übliche Suffix für C–Quelldateien ist .c und wir nehmen an, dass der obige Code in der Datei `HalloWelt.c` abgespeichert ist.
Die einfachste Form des Übersetzungsvorgangs ist die Verwendung des folgenden Befehls in der Kommandozeile:

```
gcc HalloWelt.c
```

gcc ist der Programmname des GNU-C-Compilers. Der Aufruf des Befehls erzeugt die ausführbare Datei `a.out` (`a.exe` unter Windows). Nach Eingabe von

`./a.out` (bzw. `./a.exe`)

wird das Programm gestartet. Auf dem Bildschirm erscheint die Ausgabe "Hallo Welt". Das Voranstellen von `./` kann weggelassen werden, falls sich das Arbeitsverzeichnis `./` im Suchpfad befindet. Durch Eingabe von

```
export PATH=$PATH:.
```

wird das Arbeitsverzeichnis in den Suchpfad aufgenommen. Einen anderen Namen für die ausführbare Datei erhält man mit der Option `-o`

```
gcc HalloWelt.c -o HalloWelt
```

## 8.2 Interne Details beim Compilieren (∗)

Der leicht geänderte Aufruf zum Compilieren

```
gcc -v HalloWelt.c
```

erzeugt eine längere Bilschirmausgabe, welche mehrere Phasen des Compilierens anzeigt. Im folgenden einige Tipps, wie man sich diese einzelnen Phasen anschauen kann, um den Ablauf besser zu verstehen:

a) Präprozessing:
Headerfiles (*.h) werden zur Quelldatei hinzugefügt (+ Makrodefinitionen, bedingte Compilierung)

```
gcc -E HalloWelt.c > HalloWelt.E
```

Der Zusatz > HalloWelt.E lenkt die Bildschirmausgabe in die Datei HalloWelt.E. Diese Datei HalloWelt.E kann mit einem Editor angesehen werden und ist eine lange C-Quelltextdatei.

b) Übersetzen in Assemblercode:
Hier wird eine Quelltextdatei in der (prozessorspezifischen) Programmiersprache Assembler erzeugt.

```
gcc -S HalloWelt.c
```

Die entstandene Datei HalloWelt.s kann mit dem Editor angesehen werden.

c) Objektcode erzeugen:
Nunmehr wird eine Datei erzeugt, welche die direkten Steuerbefehle, d.h. Zahlen, für den Prozessor beinhaltet.

```
gcc -c HalloWelt.c
```

Die Ansicht dieser Datei mit einem normalen Texteditor liefert eine unverständliche Zeichenfolge. Einblicke in die Struktur vom Objektcodedateien können mit Hilfe eines Monitors (auch ein Editor Programm) erfolgen.

```
hexedit HalloWelt.o
```

(Nur falls das Programm hexedit oder ein anderer Monitor installiert ist.)

d) Linken:
Verbinden aller Objektdateien und notwendigen Bibliotheken zum ausführbaren Programm *Dateiname.out* (*Dateiname.exe* unter Windows).

```
gcc -o Dateiname HalloWelt.c
```

[1] Mit ∗ gekennzeichnete Abschnitte werden in der Vorlesung nicht behandelt und sie werden in der Prüfung nicht abgefragt. Diese Abschnitte dienen Interessenten zur selbständigen Weiterbildung.

# Kapitel 9

# Variablen, Datentypen und Operationen

## 9.1 Deklaration, Initialisierung, Definition

Für die Speicherung und Manipulation von Ein- und Ausgabedaten sowie der Hilfsgrößen eines Algorithmus werden bei der Programmierung Variablen eingesetzt. Je nach Art der Daten wählt man einen von der jeweiligen Programmiersprache vorgegebenen geeigneten Datentyp aus. Vor ihrer ersten Verwendung müssen die Variablen durch Angabe ihres Typs und ihres Namens deklariert werden. In C hat die Deklaration die folgende Form:

```
Datentyp Variablenname;
```

Man kann auch mehrere Variablen desselben Typs auf einmal deklarieren, indem man die entsprechenden Variablennamen mit Komma auflistet:

```
Datentyp Variablenname1, Variablenname2, ...,VariablennameN;
```

Bei der Deklaration können einer Variablen auch schon Werte zugewiesen werden, das heißt eine Initialisierung der Variablen ist bereits möglich. Zusammen mit der Deklaration gilt die Variable dann als definiert.

Die Deklaration von Variablen sollte vor der ersten Ausführungsanweisung stattfinden. Dies ist bei den allermeisten Compilern nicht zwingend notwendig, dient aber der Übersicht des Quelltextes.

**Bemerkung 9.1 Variablennamen.** Bei der Vergabe von Variablennamen ist folgendes zu beachten:

- Variablennamen dürfen keine Umlaute enthalten. Als einziges Sonderzeichen ist der Unterstrich _ (engl. underscore) erlaubt.
- Variablennamen dürfen Zahlen enthalten, aber nicht mit ihnen beginnen.
- Groß- und Kleinschreibung von Buchstaben wird unterschieden.

□

## 9.2 Elementare Datentypen

Die Tabelle 9.1 gibt die Übersicht über die wichtigsten Datentypen in C.

Anhang A widmet sich speziell der Zahlendarstellung im Rechner. Insbesondere werden dort die Begriffe Gleitkommazahl und deren Genauigkeit erörtert.

| Schlüsselwort | Datentyp | Anzahl Bytes |
|---|---|---|
| char | Zeichen | 1 |
| int | ganze Zahl | 4 |
| float | Gleitkommazahl mit einfacher Genauigkeit | 4 |
| double | Gleitkommazahl mit doppelter Genauigkeit | 8 |
| void | leerer Datentyp | |

Tabelle 9.1: Elementare Datentypen. Die Bytelänge ist von Architektur zu Architektur unterschiedlich (hier: GNU-C-Compiler unter LINUX für die x86-Architektur. Siehe auch `sizeof()`).

**Beispiel 9.2 Deklaration, Initialisierung, Definition.**

```
#include <stdio.h>

int main()
{
    int a=4;
    /* Deklaration von a als ganze Zahl */
    /* + Initialisierung von a, das hei\3t a wird der Wert 4 zugewiesen */

    printf("Die int-Variable a wurde initialisiert mit %i\n" ,a );

    /* Die Formatangabe %i zeigt an, dass eine int-Variable
       ausgegeben wird */
    return 0;
 }
```

□

Der Datentyp **char** wird intern als ganzzahliger Datentyp behandelt. Er kann daher mit allen Operatoren behandelt werden, die auch für **int** verwendet werden. Erst durch die Abbildung der Zahlen von 0 bis 255 auf entsprechende Zeichen (ASCII-Tabelle) entsteht die Verknüpfung zu den Zeichen.
Einige dieser Datentypen können durch Voranstellen von weiteren Schlüsselwörtern modifiziert werden. Modifizierer sind:

- `signed/unsigned`: Gibt für die Typen int und char an, ob sie mit/ohne Vorzeichen behandelt werden (nur int und char).
- `short/long`: Reduziert/erhöht die Bytelänge des betreffenden Datentyps. Dabei wirkt sich short nur auf int und long nur auf double aus.
- `const`: Eine so modifizierte Variable kann initialisiert, aber danach nicht mehr mit einem anderen Wert belegt werden. Die Variable ist „schreibgeschützt“.

Bei den zulässigen Kombinationen ist die Reihenfolge

`const - signed/unsigned - long/short` *`Datentyp Variablenname`*

**Beispiel 9.3 Deklaration / Definition von Variablen.**
*Zulässig:*

```
int a;
signed char zeichen1;
unsigned short int b; oder äquivalent unsigned short b;
long double eps;
const int c=12;
```

Im letzten Beispiel wurde der Zuweisungsoperator = (s. Abschnitt 9.4) verwendet,

um die schreibgeschützte Variable `c` zu initialisieren. Variablen vom Typ `char` werden durch sogenannte Zeichenkonstanten initialisiert. Zeichenkonstanten gibt man an, indem man ein Zeichen in Hochkommata setzt, zum Beispiel

```
char zeichen1='A';
```

*Nicht zulässig:*

```
unsigned double d;
long char zeichen1;
char 1zeichen; /* unzulässiger Variablenname */
```

□

Die Funktion `sizeof()` liefert die Anzahl der Bytes zurück, die für einen bestimmten Datentyp benötigt werden. Sie hat als Rückgabewert den Typ `int`.

**Beispiel 9.4 C-Anweisung : sizeof().**

```
/* Beispiel: sizeof() */
# include <stdio.h>

int main()
{
   printf("Eine int-Variable benoetigt %i Bytes\n", sizeof(int));
   return 0;
}
```

□

## 9.3 Felder und Strings

### 9.3.1 Felder

Eine Möglichkeit, aus elementaren Datentypen weitere Typen abzuleiten, ist das Feld (Array). Ein Feld besteht aus $n$ Objekten des gleichen Datentyps. Die Deklaration eines Feldes ist von der Form

```
Datentyp Feldname[n];
```

Weitere Merkmale:

- Die Nummerierung der Feldkomponenten beginnt bei 0 und endet mit $n-1$.
- Die $i$–te Komponente des Feldes wird mit `Feldname[i]` angesprochen, $i = 0, \ldots, n-1$.
- Felder können bei der Deklaration initialisiert werden. Dies geschieht unter Verwendung des Zuweisungsoperators und der geschweiften Klammer.

**Beispiel 9.5 Felder.**

```
#include<stdio.h>

int main()
{
   float a[3]={3.2, 5, 6};
   /* Deklaration und Initialisierung eines (1 x 3) float-Feldes */

   printf("Die 0.-te Komponente von a hat den Wert %f\n" ,a[0] );
   /* Die Formatangabe %f zeigt an, dass eine float beziehungsweise
   double-Variable ausgegeben wird */
   return 0;
 }
```

□

### 9.3.2 Mehrdimensionale Felder

Es ist möglich, die Einträge eines Feldes mehrfach zu indizieren und so höherdimensionale Objekte zu erzeugen; für $d$ Dimensionen lautet die Deklaration dann

`Datentyp Feldname[`$n_1$`][`$n_2$`]...[`$n_d$`];`

**Beispiel 9.6 Deklaration und Initialisierung eines ganzzahligen 2 x 3–Feldes.**

```
#include <stdio.h>

int main()
{
   int a[2][3]={{1, 2, 3}, {4, 5, 6}};
   printf("Die [0,1].-te Komponente von a hat den Wert %i\n",a[0][1]);
   return 0;
}
```

□

### 9.3.3 Zeichenketten (Strings)

Eine Sonderstellung unter den Feldern nehmen die Zeichenketten (Strings) ein. Es handelt sich dabei um Felder aus Zeichen:

`char Stringname[Länge];`

Eine Besonderheit stellt dar, dass das Stringende durch die Zeichenkonstante `\0` markiert wird. Der String *Hallo* wird also durch

`char text[]={'H','a','l','l','o','\0'};`

initialisiert.

Ein String kann auch durch

`char text[]="Hallo";`

initialisiert werden. Dieser String hat auch die Länge 6, obwohl nur 5 Zeichen zur Initialisierung benutzt wurden. Das Ende eines Strings markiert immer die Zeichenkonstante `\0`.

**Beispiel 9.7 Deklaration und Initialisierung eines Strings.**

```
#include <stdio.h>

int main()
{
   char text[]="Hallo";
   printf("%s\n" ,text);

   /* Die Formatangabe %s zeigt an, dass ein String ausgegeben wird. */
   return 0;
}
```

□

## 9.4 Ausdrücke, Operatoren und mathematische Funktionen

Der Zuweisungsoperator

```
operand1 = operand2;
```

weist dem linken Operanden den Wert des rechten Operanden zu.

**Beispiel 9.8 Zuweisungsoperator.** Zum Beispiel ist im Ergebnis der Anweisungsfolge

```
#include <stdio.h>

int main()
{
       int x,y;
       x=2;
       y=x+4;
       printf("x=%i und y=%i\n",x,y);
       /* Formatangabe %i gibt dem printf-Befehl an,
        * dass an dieser Stelle eine Integervariable
        * ausgeben werden soll. */

       return 0;
}
```

der Wert von `x` gleich 2 und der Wert von `y` gleich 6. Hierbei sind `x`, `y`, `0`, `x+4` Operanden, wobei letzterer gleichzeitig ein Ausdruck, bestehend aus den Operanden `x`, `4` und dem Operator `+` ist. Sowohl `x=2` als auch `y=x+4` sind Ausdrücke. Erst das abschließende Semikolon `;` wandelt diese Ausdrücke in auszuführende Anweisungen.
□

Es können auch Mehrfachzuweisungen auftreten.

**Beispiel 9.9 Mehrfachzuweisung.** Die folgenden drei Zuweisungen sind äquivalent.

```
#include <stdio.h>

int main()
{
       int a,b,c;

       /* 1. Moeglichkeit */
       a = b = c = 123;
       /* 2. Moeglichkeit */
       a = (b = (c = 123));
       /* 3. Moeglichkeit (Standard) */
       c = 123;
       b=c;
       a=b;

       printf("a=%i, b=%i, c=%i\n",a,b,c);
       return 0;
}
```

□

### 9.4.1 Arithmetische Operatoren

*Unäre Operatoren.* Bei unären Operatoren tritt nur ein Operand auf.

| Operator | Beschreibung | Beispiel |
|---|---|---|
| - | Negation | -a |

*Binäre Operatoren.* Bei binären Operatoren treten zwei Operanden auf. Der Ergebnistyp der Operation hängt vom Operator ab.

| Operator | Beschreibung | Beispiel |
|---|---|---|
| + | Addition | a+b |
| - | Subtraktion | a-b |
| * | Multiplikation | a*b |
| / | Division (Achtung bei Integerwerten !!!) | a/b |
| % | Rest bei ganzzahliger Division (Modulooperation) | a%b |

**Achtung!!!** Die Division von Integerzahlen berechnet den ganzzahligen Anteil der Division, zum Beispiel liefert `8/3` das Ergebnis `2`. Wird jedoch einer der beiden Operanden in eine Gleitkommazahl umgewandelt, so erhält man das numerisch exakte Ergebnis. zum Beispiel `8.0/3` liefert `2.66666` als Ergebnis (siehe auch Kapitel 9.8).

Analog zur Mathematik gilt „Punktrechnung geht vor Strichrechnung“. Desweiteren werden Ausdrücke in runden Klammern zuerst berechnet.

**Beispiel 9.10 Arithmetische Operatoren.**

```
   % ermoeglicht das Setzen von mathematischen Ausdruecken
   % wird hier fuer die Referenz benutzt
#include <stdio.h>

int main()
{
       int a,b,c;
       double x;
       a=1;           /* a=1 */
       a=9/8;         /* a=1, Integerdivision */
       a=3.12;        /* a=3, abrunden wegen int-Variable */
       a=-3.12;       /* a=-3 oder -4, Compiler abhaengig */

       b=6;           /* b=6 */
       c=10;          /* c=10 */
       x=b/c;         /* x=0 */
       x=(double) b/c; /* x=0.6 siehe Kapitel 9.8 */
       x=(1+1)/2;     /* x=1 */
       x=0.5+1.0/2;   /* x=1 */
       x=0.5+1/2;     /* x=0.5 */
       x=4.2e12;      /* x=4.2*10^{12} wissenschaftl. Notation */

       return 0;
}
```

□

### 9.4.2 Vergleichsoperatoren

Vergleichsoperatoren sind binäre Operatoren. Der Ergebniswert ist immer ein Integerwert. Sie liefern den Wert 0, falls die Aussage falsch, und den Wert 1, falls die Aussage richtig ist.

| Operator | Beschreibung | Beispiel |
|---|---|---|
| `>` | größer | `a > b` |
| `>=` | größer oder gleich | `a >= b` |
| `<` | kleiner | `a < b/3` |
| `<=` | kleiner oder gleich | `a*b < =c` |
| `==` | gleich (Achtung bei Gleitkommazahlen !!!) | `a==b` |
| `!=` | ungleich (Achtung bei Gleitkommazahlen !!!) | `a!=3.14` |

**Achtung !!!** Ein typischer Fehler tritt beim Test auf Gleichheit auf, indem statt des Vergleichsoperators `==` der Zuweisungsoperator `=` geschrieben wird. Das Prüfen von Gleitkommazahlen auf (Un–)gleichheit kann nur bis auf den Bereich der Maschinengenauigkeit erfolgen und sollte daher vermieden werden.

**Beispiel 9.11 Vergleichsoperatoren.**

```
#include <stdio.h>

int main()
{
        int a,b;
        int aussage;
        float x,y;

        a=3;                    /* a=3 */
        b=2;                    /* b=2 */
        aussage = a>b;          /* aussage=1 ; entspricht wahr */
        aussage = a==b;         /* aussage=0 ; entspricht falsch */

        x=1.0+1.0e-8;           /* x=1 + 1.0 *10^{-8} */
        y=1.0+2.0e-8;           /* y=1 + 2.0 *10^{-8} */
        aussage = (x==y);       /* aussage=0 oder 1 ; entspricht wahr,
                                   falls eps > 10^{-8}, obwohl x ungleich y */

        return 0;
}
```

□

### 9.4.3 Logische Operatoren

Es gibt nur einen unären logischen Operator

| Operator | Beschreibung | Beispiel |
|---|---|---|
| `!` | logische Negation | `! (3>4)   /* Ergebnis = 1; entspricht wahr */` |

und zwei binäre logische Operatoren.

| Op. | Beschreibung | Beispiel |
|---|---|---|
| `&&` | logisches UND | `(3>4) && (3<=4)  /* Ergebnis = 0; entspricht falsch */` |
| || | logisches ODER | (3>4) || (3<=4)  /* Ergebnis = 1; entspricht wahr */ |

Die Wahrheitstafeln für das logische UND und das logische ODER sind aus der Algebra bekannt.

### 9.4.4 Bitorientierte Operatoren (∗)

Bitorientierte Operatoren sind nur auf `int`–Variablen (beziehungsweise `char`–Variablen) anwendbar. Um die Funktionsweise zu verstehen, muss man zunächst die Darstellung von Ganzzahlen innerhalb des Rechners verstehen.

Ein Bit ist die kleinste Informationseinheit mit genau zwei möglichen Zuständen:

$$\left\{ \begin{array}{l} \text{bit ungesetzt} \\ \text{bit gesetzt} \end{array} \right. \equiv \left\{ \begin{array}{l} 0 \\ 1 \end{array} \right. \equiv \left\{ \begin{array}{l} \text{falsch} \\ \text{wahr} \end{array} \right. .$$

Ein Byte besteht aus 8 Bit. Eine `short int`–Variable besteht aus 2 Byte. Damit kann also eine `short int`–Variable $2^{16}$ Werte annehme. Das erste Bit bestimmt das Vorzeichen der Zahl. Gesetzt bedeutet - (negativ), nicht gesetzt entspricht + (positiv).

**Beispiel 9.12 (Short)-Integerdarstellung im Rechner.**

| Darstellung im Rechner (binär) | Dezimal |
|---|---|
| $\underbrace{\underbrace{0}_{+}\ 0000000}_{\text{1. Byte}}\ \underbrace{00001010}_{\text{2. Byte}}$ | $2^3 + 2^1 = 10$ |
| $\underbrace{\underbrace{1}_{-}\ 1111111}_{\text{1. Byte}}\ \underbrace{11011011}_{\text{2. Byte}}$ | $-(2^5 + 2^2) - 1 = -37$ |

□

**Unäre bitorientierte Operatoren**

| Operator | Beschreibung | Beispiel |
|---|---|---|
| ∼ | Binärkomplement | ∼ `a` |

**Binäre bitorientierte Operatoren**

| Operator | Beschreibung | Beispiel |
|---|---|---|
| `&` | bitweises UND | a `& 1` |
| `\|` | bitweises ODER | a `\| 1` |
| ^ | bitweises exklusives ODER | a ^ `1` |
| `<<` | Linksshift der Bits von op1 um op2 Stellen | a `<< 1` |
| `>>` | Rechtsshift der Bits von op1 um op2 Stellen | `a >> 2` |

**Wahrheitstafel**

| x | y | x & y | x \| y | x ^ y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Beispiel 9.13 Bitorientierte Operatoren.**

```
#include <stdio.h>

int main()
{
        short int a,b,c;

        a=5;            /* 00000000 00000101 = 5 */
        b=6;            /* 00000000 00000110 = 6 */

        c= ~ b;         /* Komplement 11111111 11111001 =-(2^2+2^1)-1=-7 */
        c=a & b;        /* 00000000 00000101 = 5            */
                        /* bit-UND &                        */
                        /* 00000000 00000110 = 6            */
                        /* gleich                           */
                        /* 00000000 00000100 = 4            */

        c=a | b;        /* bit-ODER 00000000 00000111 = 7 */

        c=a^b;          /* bit-ODER exklusiv 00000000 00000011 = 3  */

        c=a << 2;       /* 2 x Linksshift 00000000 00010100 = 20   */

        c=a >> 1;       /* 1 x Rechtsshift & 00000000 00000010 = 2 */

        return 0;
}
```

□

### 9.4.5 Inkrement- und Dekrementoperatoren

**Präfixnotation**

| Notation | Beschreibung |
|---|---|
| `++operand` | operand=operand+1 |
| `--operand` | operand=operand-1 |

Inkrement– und Dekrementoperatoren in Präfixnotation liefern den inkrementierten beziehungsweise dekrementierten Wert als Ergebnis zurück.

**Beispiel 9.14 Präfixnotation.**

```
#include <stdio.h>

int main()
{
        int i,j;

        i=3;
        ++i; /* i=4 */
        j=++i; /* i=5, j=5 */

        /* oben angegebene Notation ist aequivalent zu */
        i=3;
        i=i+1;
        i=i+1;
```

```
        j=i;

        return 0;
}
```

□

**Postfixnotation**

| Notation | Beschreibung |
|---|---|
| `operand++` | operand=operand+1 |
| `operand--` | operand=operand-1 |

Inkrement– und Dekrementoperatoren in Postfixnotation liefern den Wert vor dem Inkrementieren beziehungsweise Dekrementieren zurück.

**Beispiel 9.15 Postfixnotation.**

```
#include <stdio.h>

int main()
{
        int i,j;

        i=3;
        i++; /* i=4 */
        j=i++; /* j=4 ,i=5 */

        /* oben angegebene Notatation ist aequivalent zu */

        i=3;
        i=i+1;
        j=i;
        i=i+1;

        return 0;
}
```

□

### 9.4.6 Adressoperator

Der Vollständigkeit halber wird der Adressoperator `&` schon in diesem Kapitel eingeführt, obwohl die Bedeutung erst in Kapitel 11 klar wird.

```
&datenobjekt;
```

### 9.4.7 Prioritäten von Operatoren

Es können beliebig viele Aussagen durch Operatoren verknüpft werden. Die Reihenfolge der Ausführung hängt von der Priorität der jeweiligen Operatoren ab. Operatoren mit höherer Priorität werden vor Operatoren niedriger Priorität ausgeführt. Haben Operatoren die gleiche Priorität so werden sie gemäß ihrer sogenannten Assoziativität von links nach rechts oder umgekehrt abgearbeitet.

**Prioritäten von Operatoren beginnend mit der Höchsten**

<table>
<tr><th>Priorität</th><th>Operator</th><th>Beschreibung</th><th>Assoz.</th></tr>
<tr><td>15</td><td>()</td><td>Funktionsaufruf</td><td>→</td></tr>
<tr><td></td><td>[]</td><td>Indizierung</td><td>→</td></tr>
<tr><td></td><td>-></td><td>Elementzugriff</td><td>→</td></tr>
<tr><td></td><td>.</td><td>Elementzugriff</td><td>→</td></tr>
<tr><td>14</td><td>+</td><td>Vorzeichen</td><td>←</td></tr>
<tr><td></td><td>-</td><td>Vorzeichen</td><td>←</td></tr>
<tr><td></td><td>!</td><td>Negation</td><td>←</td></tr>
<tr><td></td><td>~</td><td>Bitkomplement</td><td>←</td></tr>
<tr><td></td><td>++</td><td>Präfix-Inkrement</td><td>←</td></tr>
<tr><td></td><td>--</td><td>Präfix-Dekrement</td><td>←</td></tr>
<tr><td></td><td>++</td><td>Postfix-Inkrement</td><td>←</td></tr>
<tr><td></td><td>--</td><td>Postfix-Dekrement</td><td>←</td></tr>
<tr><td></td><td>&</td><td>Adresse</td><td>←</td></tr>
<tr><td></td><td>*</td><td>Zeigerdereferenzierung</td><td>←</td></tr>
<tr><td></td><td>(Typ)</td><td>Cast</td><td>←</td></tr>
<tr><td></td><td>sizeof()</td><td>Größe</td><td>←</td></tr>
<tr><td>13</td><td>*</td><td>Multiplikation</td><td>→</td></tr>
<tr><td></td><td>/</td><td>Division</td><td>→</td></tr>
<tr><td></td><td>%</td><td>Modulo</td><td>→</td></tr>
<tr><td>12</td><td>+</td><td>Addition</td><td>→</td></tr>
<tr><td></td><td>-</td><td>Subtraktion</td><td>→</td></tr>
<tr><td>11</td><td><<</td><td>Links-Shift</td><td>→</td></tr>
<tr><td></td><td>>></td><td>Rechts-Shift</td><td>→</td></tr>
<tr><td>10</td><td><</td><td>kleiner</td><td>→</td></tr>
<tr><td></td><td><=</td><td>kleiner gleich</td><td>→</td></tr>
<tr><td></td><td>></td><td>größer</td><td>→</td></tr>
<tr><td></td><td>>=</td><td>größer gleich</td><td>→</td></tr>
<tr><td>9</td><td>==</td><td>gleich</td><td>→</td></tr>
<tr><td></td><td>!=</td><td>ungleich</td><td>→</td></tr>
<tr><td>8</td><td>&</td><td>bitweises UND</td><td>→</td></tr>
<tr><td>7</td><td>^</td><td>bitweises exklusivers ODER</td><td>→</td></tr>
<tr><td>6</td><td>|</td><td>bitweises ODER</td><td>→</td></tr>
<tr><td>5</td><td>&&</td><td>logisches UND</td><td>→</td></tr>
<tr><td>4</td><td>||</td><td>logisches ODER</td><td>→</td></tr>
<tr><td>3</td><td>:?</td><td>Bedingung</td><td>←</td></tr>
<tr><td>2</td><td>=</td><td>Zuweisung</td><td>←</td></tr>
<tr><td></td><td>*=, /=, +=</td><td>Zusammengesetzte Zuweisung</td><td>←</td></tr>
<tr><td></td><td>-=, &=, ^=</td><td>Zusammengesetzte Zuweisung</td><td>←</td></tr>
<tr><td></td><td>|=, <<=, >>=</td><td>Zusammengesetzte Zuweisung</td><td>←</td></tr>
<tr><td>1</td><td>,</td><td>Komma-Operator</td><td>→</td></tr>
</table>

Im Zweifelsfall kann die Priorität durch Klammerung erzwungen werden.

**Beispiel 9.16 Prioritäten von Operatoren.**

```
#include <stdio.h>

int main()
{
	int a=-4, b=-3, c;

	c=a<b<-1;            /* c=0 ; falsch */
	c=a<(b<-1);          /* c=1 ; wahr */
	c=a ==-4 && b == -2;  /* c=0 ; falsch */

	return 0;
}
```

Die erste Anweisung wird von links nach rechts abgearbeitet. Dabei ist zunächst `a<b == 1` (wahr). Im nächsten Schritt ist aber `1<-1 == 0` (falsch). Die Abarbeitung von rechts erzwingt man mit der Klammer (zweite Zeile). In der dritten Zeile ist der rechte Term neben `&&` falsch. □

## 9.5 Operationen mit vordefinierten Funktionen

### 9.5.1 Mathematische Funktionen

Im Headerfile `math.h` werden unter anderem Deklarationen der in Tabelle 9.2 zusammengefassten mathematischen Funktionen und Konstanten bereitgestellt:

| Funktion/Konstante | Beschreibung |
|---|---|
| `sqrt(x)` | Wurzel von $x$ |
| `exp(x)` | $e^x$ |
| `log(x)` | natürlicher Logarithmus von $x$ |
| `pow(x,y)` | $x^y$ |
| `fabs(x)` | Absolutbetrag von $x : \|x\|$ |
| `fmod(x,y)` | realzahliger Rest von $x/y$ |
| `ceil(x)` | nächste ganze Zahl $\geq x$ |
| `floor(x)` | nächste ganze Zahl $\leq x$ |
| `sin(x), cos(x), tan(x)` | trigonometrische Funktionen |
| `asin(x), acos(x), atan(x)` | trigonometrische Umkehrfunktionen |
| `M_E` | Eulersche Zahl $e$ |
| `M_PI` | $\pi$ |

Tabelle 9.2: Mathematische Funktionen

Für die Zulässigkeit der Operation, das heißt den Definitionsbereich der Argumente, ist der Programmierer verantwortlich, siehe Dokumentationen (`man`). Ansonsten werden Programmabbrüche oder unsinnige Ergebnisse produziert.

**Beispiel 9.17 Mathematische Funktionen und Konstanten.**

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

int main()
```

```
{
        float x,y,z;
        x=4.5;          /* x=4.5 */
        y=sqrt(x);      /* y=2.121320, was ungefaehr = sqrt(4.5) */
        z=M_PI;         /* z=3.141593, was ungefaehr = pi */

        return 0;
}
```

□

### 9.5.2 Funktionen für Zeichenketten (Strings)

Im Headerfile `string.h` werden unter anderem die Deklarationen der folgenden Funktionen für Strings bereitgestellt:

| Funktion | Beschreibung |
|---|---|
| `strcat(s1,s2)` | anhängen von `s2` an `s1` |
| `strcmp(s1,s2)` | lexikographischer Vergleich der Strings `s1` und `s2` |
| `strcpy(s1,s2)` | kopiert `s2` auf `s1` |
| `strlen(s)` | Anzahl der Zeichen in String `s` ( = `sizeof(s)-1`) |
| `strchr(s,c)` | sucht Zeichenkonstante (Character) `c` in String `s` |

Tabelle 9.3: Funktionen für Strings

**Beispiel 9.18 Funktionen für Zeichenketten (Strings).**

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

int main()
{
        int i;
        char s1[]="Hallo";    /* reserviert 5+1 Byte im Speicher fuer s1
                              und belegt sie mit H,a,l,l,o,\0     */
        char s2[]="Welt";     /* reserviert 4+1 Byte im Speicher f\"ur s2 */
        char s3[100]="Hallo"; /* reserviert 100 Byte im Speicher f\"ur s3
                               * und belegt die ersten 6 mit H,a,l,l,o,\0 */

        /* !!!NICHT ZULAESSIG!!! (Kann zu Programmabsturz fuehren) *** */
        strcat(s1,s2);        /* Im reservierten Speicherbereich von s1
                              * steht nun H,a,l,l,o,W
                              * Der Rest von s2 wird irgendwo in den
                              * Speicher geschrieben */
        /* ZULAESSIG */
        strcat(s3,s2);        /* Die ersten 10 Bytes von s3 sind nun
                               * belegt mit H,a,l,l,o,W,e,l,t,\0
                               * Der Rest ist zufaellig beschrieben */
        strcpy(s1,s2);        /* Die ersten 5 Bytes von s1 sind nun
                               * belegt mit W,e,l,t,\0 */
        i=strlen(s2);         /* i=4 */
        i=strcmp(s2,s3);      /* i=15, Unterschied zwischen 'W' und 'H' in
                               * ASCII*/
        return 0;
```

```
}
```

□

**Achtung!** Der Umgang mit Strings ist problematisch, zum Beispiel wird bei dem Befehl `strcat(s1,s2)` der String `s2` an `s1` angehängt. Dadurch wird der Speicherbedarf für String `s1` vergrößert. Wurde bei der Deklaration von `s1` zu wenig Speicherplatz reserviert (allokiert) schreibt der Computer die überschüssigen Zeichen in einen nicht vorher bestimmten Speicherbereich. Dies kann unter Umständen sogar zum Absturz des Programms führen – das Ergebnis können seltsame und schwer zu findende Fehler im Programm sein, die teilweise nicht immer auftreten (siehe auch Beipiel 9.18).

## 9.6 Zusammengesetzte Anweisungen

Wertzuweisungen der Form

```
op1=op1 operator op2;
```

können zu

```
op1 operator = op2;
```

verkürzt werden.
Hierbei ist `operator` $\in$ {`+`,`-`,`*`,`/`,`%`,`|`,`^`,`<<`,`>>`}.

**Beispiel 9.19 Zusammengesetzte Anweisungen.**

```
#include <stdio.h>

int main()
{
        int i=7,j=3;

        i += j; /* i=i+j; */
        i >>= 1; /* i=i >> 1 (i=i/2), bitorientierte Operation */
        j *= i; /* j=j*i */

        return 0;
}
```

□

## 9.7 Nützliche Konstanten

Für systemabhängige Zahlenbereiche, Genauigkeiten und so weiter ist die Auswahl der Konstanten aus Tabelle 9.4 und Tabelle 9.5 recht hilfreich. Sie stehen dem Programmierer durch Einbinden der Headerdateien `float.h` beziehungsweise `limits.h` zur Verfügung.

Weitere Konstanten können in der Datei `float.h` nachgeschaut werden. Der genaue Speicherort dieser Datei ist abhängig von der gerade verwendeten Version des gcc und der verwendeten Distribution. Die entsprechenden Headerfiles können auch mit dem Befehl

```
find /usr -name float.h -print
```

gesucht werden. Dieser Befehl durchsucht den entsprechenden Teil des Verzeichnisbaums (`/usr`) nach der Datei namens `float.h`.

Tabelle 9.4: Konstanten aus float.h

| Konstante | Beschreibung |
|---|---|
| FLT_DIG | Anzahl gültiger Dezimalstellen für `float` |
| FLT_MIN | kleinste, darstellbare positive `float` Zahl |
| FLT_MAX | größte, darstellbare positive `float` Zahl |
| FLT_EPSILON | kleinste positive `float` Zahl `eps` mit `1.0+eps` $\neq$ `1.0` |
| DBL_ | wie oben für `double` |
| LDBL_ | wie oben für `long double` |

Tabelle 9.5: Konstanten aus limits.h

| Konstante | Beschreibung |
|---|---|
| INT_MIN | kleinste, darstellbare `int` Zahl |
| INT_MAX | größte, darstellbare `int` Zahl |
| SHRT_ | wie oben für `short int` |

## 9.8 Typkonversion (cast)

**Beispiel 9.20 Abgeschnittene Division.**

```
#include <stdio.h>

int main()
{
	int a=10, b=3;
	float quotient;
	quotient = a/b;        /* quotient = 3 */
	quotient = (float) a/b; /* quotient = 3.3333 */

	return 0;
}
```

Nach der Zuweisung `a/b` hat die Variable `quotient` den Wert 3.0, obwohl sie als Gleitkommazahl deklariert wurde! Ursache: Resultat der Division zweier `int`–Variablen ist standardmäsig wieder ein `int`–Datenobjekt.

Abhilfe schaffen hier Typumwandlungen (engl.: casts). Dazu setzt man den gewünschten Datentyp in Klammern vor das umzuwandelnde Objekt, im obigen Beispiel:

```
quotient = (float) a/b;
```

Hierdurch wird das Ergebnis mit den Nachkommastellen übergeben. □

**Achtung!** bei Klammerung von Ausdrücken! Die Anweisung

```
quotient = (float) (a/b);
```

führt wegen der Klammern die Division komplett im `int`–Kontext durch und der Cast bleibt wirkungslos.

**Bemerkung 9.21** Die im ersten Beispiel gezeigte abgeschnittene Division erlaubt in Verbindung mit dem Modulooperator `%` eine einfache Programmierung der Division mit Rest. □

Ist einer der Operanden eine Konstante, so kann man auch auf Casts verzichten: Statt

```
quotient = (float) 10/b;
```

kann man die Anweisung

```
quotient = 10.0/b;
```

verwenden.

## 9.9 Standardein– und –ausgabe

**Eingabe:** Das Programm fordert benötigte Informationen/Daten vom Benutzer an.

**Ausgabe:** Das Programm teilt die Forderung nach Eingabedaten dem Benutzer mit und gibt (Zwischen–) Ergebnisse aus.

### 9.9.1 Ausgabe

Die Ausgabe auf das Standardausgabegerät (Terminal, Bildschirm) erfolgt mit der `printf()`–Bibliotheksfunktion. Die Anweisung ist von der Form

```
printf(Formatstringkonstante, Argumentliste);
```

Die Argumentliste ist eine Liste von auszugebenden Objekten, jeweils durch ein Komma getrennt (Variablennamen, arithmetische Ausdrücke etc.). Die Formatstringkonstante enthält neben Text zusätzliche spezielle Zeichen: spezielle Zeichenkonstanten (Escapesequenzen) und Formatangaben.

| **Zeichenkonstante** | **erzeugt** |
|---|---|
| `\n` | neue Zeile |
| `\t` | Tabulator |
| `\v` | vertikaler Tabulator |
| `\b` | Backspace |
| `\\` | Backslash \ |
| `\?` | Fragezeichen ? |
| `\'` | Hochkomma |
| `\''` | Anführungsstriche |

Die Formatangaben spezifizieren, welcher Datentyp auszugeben ist und wie er auszugeben ist. Sie beginnen mit `%`. Die folgende Tabelle gibt einen Überblick über die wichtigsten Formatangaben:

| `Formatangabe` | `Datentyp` |
|---|---|
| `%f, %g` | `float, double` |
| `%i, %d` | `int, short` |
| `%u` | `unsigned int` |
| `%o` | `int, short oktal` |
| `%x` | `int, short hexadezimal` |
| `%c` | `char` |
| `%s` | Zeichenkette (String) |
| `%li, %ld` | `long` |
| `%Lf` | `long double` |
| `%e` | `float, double` wissenschaftl. Notation |

Durch Einfügen eines Leerzeichens nach `%` wird Platz für das Vorzeichen ausgespart. Nur negative Vorzeichen werden angezeigt. Fügt man stattdessen ein `+` ein, so wird das Vorzeichen immer angezeigt.

Weitere Optionen kann man aus Beispiel 9.22 entnehmen oder man erhält sie mit `man sprintf`.

**Beispiel 9.22 Ausgabe von Gleitkommazahlen.**

```
#include <stdio.h>

int main()
{
	const double pi=3.14159265;

	printf("Pi = %f\n",pi);
	printf("Pi = % f\n",pi);
	printf("Pi = %+f\n",pi);
	printf("Pi = %.3f\n",pi);
	printf("Pi = %.7e\n",pi);

	return 0;
}
```

erzeugen die Bildschirmausgabe

Pi = 3.141593
Pi =  3.141593
Pi = +3.141593
Pi = 3.142
Pi = 3.1415927e+00

□

### 9.9.2 Eingabe

Für das Einlesen von Tastatureingaben des Benutzers steht unter anderem die Bibliotheksfunktion `scanf()` zur Verfügung. Ihre Verwendung ist auf den ersten Blick identisch mit der von `printf()`.

```
scanf(Formatstringkonstante, Argumentliste);
```

Die Argumentliste bezieht sich auf die Variablen, in denen die eingegebenen Werte abgelegt werden sollen, wobei zu beachten ist, dass in der Argumentliste nicht die Variablen selbst, sondern ihre *Adressen* anzugeben sind. Dazu verwendet man den Adressoperator `&`, siehe Abschnitt 11.

**Beispiel 9.23 Einlesen einer ganzen Zahl.**

```
#include <stdio.h>

int main()
{
	int a;

	printf("Geben Sie eine ganze Zahl ein: ");
	scanf("%i",&a);
	printf("a hat nun den Wert : %i\n",a);
```

```
    return 0;
}
```

Die eingegebene Zahl wird als `int` interpretiert und an der Adresse der Variablen `a` abgelegt. □

Die anderen Formatangaben sind im Wesentlichen analog zu `printf()`. Eine Ausnahme ist das Einlesen von `double`– und `long double`–Variablen. Statt `%f` sollte man hier

- `%lf` für `double`,
- `%Lf` für `long double`,

verwenden. Das Verhalten variiert je nach verwendetem C–Compiler.

**Achtung!** Handelt es sich bei der einzulesenden Variable um ein Feld (insbesondere String) oder eine Zeiger Variable (siehe Kapitel 11), so entfällt der Adressoperator `&` im `scanf()`–Befehl.

Ein Beispiel:

```
char text[100];
scanf("%s",text);
```

Die Funktion `scanf` ist immer wieder eine Quelle für Fehler.

```
int zahl;
char buchstabe;

scanf("%i", &zahl);
scanf("%c", &buchstabe);
```

Wenn man einen solchen Code laufen lässt, wird man sehen, dass das Programm den zweiten `scanf`–Befehl scheinbar einfach überspringt. Der Grund ist die Art, wie `scanf` arbeitet. Die Eingabe des Benutzers beim ersten `scanf` besteht aus zwei Teilen: einer Zahl (sagen wir 23) und der Eingabetaste (die wir mit `\n` bezeichnen). Die Zahl 23 wird in die Variable `zahl` kopiert, das `\n` steht aber immer noch im sogenannten Tastaturpuffer. Beim zweiten `scanf` liest der Rechner dann sofort das `\n` aus und geht davon aus, dass der Benutzer dieses `\n` als Wert für die Variable `buchstabe` wollte. Vermeiden kann man dies mit einem auf den ersten Blick komplizierten Konstrukt, das dafür deutlich flexibler ist.

```
int zahl;
char buchstabe;
char tempstring[80];

/* wir lesen eine ganze Zeile in den String tempstring von stdin -
 * das ist die Standardeingabe
 */
fgets(tempstring, sizeof(tempstring), stdin);

/* Wir haben jetzt einen ganzen String, wie teilen wir ihn auf?
 * => mit der Funktion sscanf
 */
sscanf(tempstring, "%d", &zahl);

/* und nun nochmal fuer den Buchstaben */
```

```
fgets(tempstring, sizeof(tempstring), stdin);
sscanf(tempstring, "%c", &buchstabe);
```

Der Rückgabewert von `fgets` ist ein Zeiger; der obige Code überprüft nicht, ob dies ein NULL Zeiger ist – diese Überprüfung ist in einem Programm natürlich Pflicht! Die Funktionen `fgets` und `sscanf` sind in `stdio.h` deklariert.

# Kapitel 10

# Programmflusskontrolle

## 10.1 Bedingte Ausführung

Bei der bedingten Ausführung werden Ausdrücke auf ihren Wahrheitswert hin überprüft und der weitere Ablauf des Programms davon abhängig gemacht. C sieht hierfür die Anweisungen `if` und `switch` vor.

### 10.1.1 Die if()–Anweisung

Die allgemeine Form der Verzweigung (Alternative) ist

```
if (logischer Ausdruck)
{
      Anweisungen A;
}
else
{
      Anweisungen B;
}
```

und zählt ihrerseits wiederum als Anweisung. Der `else`–Zweig kann weggelassen werden (einfache Alternative). Folgt nach dem `if`– bzw. `else`–Befehl nur eine Anweisung, so muss diese nicht in einen Block (geschweifte Klammern) geschrieben werden.

**Beispiel 10.1 Signum-Funktion.** Die Signum-Funktion gibt das Vorzeichen an:

$$y(x) = \begin{cases} 1 & x > 0, \\ 0 & x = 0, \\ -1 & x < 0. \end{cases}$$

```
int main() /* Signum Funktion */
{
        float x,y;

        if (x>0.0)
        {
                y=1.0;
        }
        else
        {
                if (x == 0.0)
```

```
            {
                    y=0.0;
            }
            else
            {
                    y=-1.0;
            }
        }
        return 0;
}
```

Die Kurzform des obigen Programms ist

```
#include <stdio.h>

int main() /* Signum Funktion */
{
        float x,y;

        printf("Geben Sie eine Zahl ein: ");
        scanf("%f",&x);

        if (x>0)
            y=1;
        else
            if (x == 0)
               y=0;
            else
               y=-1;
        printf("Das Vorzeichen von x = %f ist %.0f\n",x,y);
        return 0;
}
```

□

### 10.1.2 Die switch()–Anweisung

Zur Unterscheidung von mehreren Fällen ist die Verwendung von `switch–case`–Kombinationen bequemer. Mit dem Schlüsselwort `switch` wird ein zu überprüfender Ausdruck benannt. Es folgt ein Block mit `case`–Anweisungen, die für die einzelnen möglichen Fälle Anweisungsblöcke vorsehen. Mit dem Schlüsselwort `default` wird ein Anweisungsblock eingeleitet, der dann auszuführen ist, wenn keiner der anderen Fälle eingetreten ist (optional).

```
  switch (Ausdruck)
  {
        case Fall 1:
        {
                Anweisungen fuer Fall 1;
                break;
        }
        ...
        case Fall n:
        {
                Anweisungen fuer Fall n;
                break;
        }
        default:
        {
```

```
                Anweisungen fuer alle anderen Faelle;
                break;
        }
 }
```

**Achtung!!!** Man beachte, dass der Anweisungsblock jedes `case`–Falles mit `break` abgeschlossen werden muss! Ansonsten wird in C der nächste `case`–Block abgearbeitet. Das ist anders als in MATLAB!

**Beispiel 10.2 switch–Anweisung.**

```
#include <stdio.h>

int main()
{
        int nummer;
        printf("Geben Sie eine ganze Zahl an: ");
        scanf("%i",&nummer);

        printf("Namen der Zahlen aus {1,2,3} \n");
        switch (nummer)
        {
                case 1:
                {
                        printf("Eins = %i \n", nummer);
                        break;
                }
                case 2:
                {
                        printf("Zwei = %i \n", nummer);
                        break;
                }
                case 3:
                {
                        printf("Drei = %i \n", nummer);
                        break;
                }
                default:
                {
                        printf("Die Zahl liegt nicht in der Menge {1,2,3} \n");
                        break;
                }
        }
        return 0;
}
```

□

## 10.2 Schleifen

Schleifen dienen dazu, die Ausführung von Anweisungsblöcken zu wiederholen. Die Anzahl der Wiederholungen ist dabei an eine Bedingung geknüpft. Zur Untersuchung, ob eine Bedingung erfüllt ist, werden Vergleichs– und Logikoperatoren aus Kapitel 9.4 benutzt.

### 10.2.1 Der Zählzyklus (for–Schleife)

Beim Zählzyklus steht die Anzahl der Zyklendurchläufe a–priori fest, der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus. Die allgemeine Form ist

```
        for (ausdruck1; ausdruck2; ausdruck3)
        {
                Anweisungen;
        }
```

**Beispiel 10.3 Summe der natürlichen Zahlen von 1 bis n.** Vergleich dazu auch Algorithmus 2.5.

```
#include <stdio.h>

int main()
{
        int i,summe,n;
        char tempstring[80];
        /* Einlesen der oberen Schranke n
         * von der Tastatur */
        printf("Obere Schranke der Summe : ");
        fgets(tempstring, sizeof(tempstring), stdin);
        sscanf(tempstring, "%i", &n);

        summe=0;       /* Setze summe auf 0                */
        for (i=1; i<=n; i=i+1)
        {
                summe=summe+i;
        }
        printf("Summe der Zahlen von 1 bis %i ist %i \n",n,summe);
        return 0;
}
```

Im obigen Programmbeispiel ist `i` die Laufvariable des Zählzyklus, welche mit `i=1` (`ausdruck1`) initialisiert wird, mit `i=i+1` (`ausdruck3`) weitergezählt und in `i <= n` (`ausdruck2`) bezüglich der oberen Grenze der Schleifendurchläufe getestet wird. Im Schleifeninneren `summe=summe+i;` (`anweisung`) erfolgen die eigentlichen Berechnungsschritte des Zyklus. Die Summationsvariable muss vor dem Eintritt in den Zyklus initialisiert werden. □

**Beispiel 10.4 Kompakte Programmierung der Summe der natürlichen Zahlen von 1 bis n.** Eine kompakte Version dieser Summationsschleife (korrekt, aber sehr schlecht lesbar) wäre:

```
for (summe=0, i=1; i<=n; summe+=i, i++);
```

Man unterscheidet dabei zwischen den Abschluss einer Anweisung `;` und dem Trennzeichen `;` in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet. □

Der `ausdruck2` ist stets ein logischer Ausdruck und `ausdruck3` ist ein arithmetischer Ausdruck zur Manipulation der Laufvariablen. Die Laufvariable kann eine einfache Variable vom Typ `int`, `float` oder `double` sein.

**Achtung!!!** Vorsicht bei der Verwendung von Gleitkommazahlen (`float`, `double`) als Laufvariable. Dort ist der korrekte Abbruchtest wegen der internen Zahlendarstellung unter Umständen nicht einfach zu realisieren.

Die folgenden Beipiele 10.5, 10.6 verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tipps zu deren Umgehung.

**Beispiel 10.5** Ausgabe der diskreten Knoten $x_i$ des Intervalls $[a, b]$, welches in $n$ gleichgroße Teilintervalle zerlegt wird, das heißt

$$x_i = a + ih,\ i = 0, \ldots, n \ \text{ mit } \ h = \frac{b-a}{n}.$$

```
#include <stdio.h>

int main()
{
	float a,b,xi,h;
	int n;
	char tempstring[80];

	a=0.0;   /* Intervall [a,b] wird initialisiert */
	b=1.0;   /* mit [0,1] */

	printf("Geben Sie die Anzahl der Teilintervalle an: ");
	fgets(tempstring, sizeof(tempstring), stdin);
	sscanf(tempstring, "%i", &n);
	h=(b-a)/n;

	n=1;   /* n wird nun als Hilfsvariable verwendet */
	for (xi=a; xi<=b; xi=xi+h)
	{
		printf("%i.te Knoten : %f \n",n,xi);
		n=n+1;
	}
	return 0;
}
```

Da Gleitkommazahlen nur eine limitierte Anzahl gültiger Ziffern besitzt, wird es oft passieren, dass der letzte Knoten $x_n = b$ nicht exakt ausgegeben oder gar nicht ausgegeben wird. Auswege sind:

1.) Falls $x_n$ gar nich ausgegeben wird: Änderung des Abbruchtests in $\texttt{xi} <= \texttt{b} + \texttt{h}/2.0$, jedoch ist $x_n$ dann immer noch fehlerbehaftet.

2.) Keine fortlaufende Addition, sondern Berechnung der Knoten immer vom ersten Knoten ausgehend, verwende dazu Zyklus mit `int`–Variable:

```
for (i=0; i<=n; i++)
{
  xi=a+i*h;
  printf("'\%i.te Knoten : \%f \bs n"',n,xi);
}
```

□

Die gemeinsame Summation kleinerer und größerer Zahlen kann ebenfalls zu Ungenauigkeiten führen, vergleiche auch Algorithmus 2.2 zur Lösung der quadratischen Gleichung.

**Beispiel 10.6** Im diesem Beispiel wird die Summe $\sum_{i=1}^{n} 1/i^2$ auf zwei verschiedene Arten berechnet:

$$\sum_{i=1}^{n} \frac{1}{i^2} = \sum_{i=1}^{n} \frac{1}{(i*i)} = \sum_{i=1}^{n} \frac{\left(\frac{1}{i}\right)}{i}.$$

Der Reihenwert ist $\pi^2/6 = 1.644934068...$. Bei der zweiten Summe werden im Programm die Summanden in umgekehrter Reihenfolge aufsummiert!

```
#include <stdio.h>
#include <math.h>
#include <limits.h> /* enth\"alt die Konstante INT_MAX */

int main()
{
       float summe1 = 0.0,summe2 = 0.0;
       int i,n;
       char tempstring[80];

       printf("Der erste Algorithmus wird ungenau f\"ur n bei ca. %f \n",
              ceil(sqrt(1.0/1.0e-6)) );
       /* siehe Kommentar 1.Schranke */

       printf("Weitere Fehler ergeben sich f\"ur n >= %f, \n",
              ceil(sqrt(INT_MAX)) );
       /* siehe Kommentar 2.Schranke */

       printf("Geben Sie die obere Summationsschranke n an : ");
       fgets(tempstring, sizeof(tempstring), stdin);
       sscanf(tempstring, "%i", &n);

       for (i=1; i<=n; i++)
       {
              /* 1. Schranke f\"ur i */
              /* Der Summand 1.0/(i*i) wird bei der Addition */
              /* nich mehr ber\"ucksichtigt, falls 1.0/(i*i) < 1.0e-6 */

              /* 2. Schranke f\"ur i */
              /* Das Produkt i*i ist als int-Variable nicht */
              /* mehr darstellbar, falls i*i > INT_MAX */

              summe1=summe1+1.0/(i*i);
       }

       for (i=n; i>=1; i--)
       {
              summe2=summe2+1.0/i/i;
       }

       printf("Der erste Algorithmus liefert das Ergebnis : %f \n",summe1);
       printf("Der zweite Algorithmus liefert das Ergebnis : %f \n",summe2);
       return 0;
}
```

Das numerische Resultat in `summe2` ist genauer, da dort zuerst alle kleinen Zahlen addiert werden, welche bei `summe1` wegen der beschränkten Anzahl gültiger Ziffern keinen Beitrag zur Summation mehr liefern können. Gleichzeitig ist zu beachten,

dass die Berechnung von `i*i` nicht mehr in `int`–Zahlen darstellbar ist für `i*i > INT_MAX`. Dagegen erfolgt die Berechnung `1.0/i/i` vollständig im Bereich von Gleitkommazahlen. □

### 10.2.2 Abweisender Zyklus (while–Schleife)

Beim abweisenden Zyklus steht die Anzahl der Durchläufe nicht a–priori fest. Der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus.

Die allgemeine Form ist

```
while(logischer Ausdruck)
{
       Anweisungen;
}
```

**Beispiel 10.7 while–Schleife.** Für eine beliebige Anzahl von Zahlen soll das Quadrat berechnet werden. Die Eingabeserie wird durch die Eingabe von 0 beendet.

```
#include <stdio.h>

int main()
{
       float zahl;
       char tempstring[80];

       printf("Geben Sie eine Zahl ein ('0' f\"ur Ende) : ");
       fgets(tempstring, sizeof(tempstring), stdin);
       sscanf(tempstring, "%f", &zahl);

       while (zahl != 0.0)
       {
              printf("%f hoch 2 = %f \n",zahl,zahl*zahl);
              printf("Geben Sie eine Zahl ein ('0' f\"ur Ende) : ");
              fgets(tempstring, sizeof(tempstring), stdin);
              sscanf(tempstring, "%f", &zahl);
       }
       return 0;
}
```

□

### 10.2.3 Nichtabweisender Zyklus (do–while–Schleife)

Beim nichtabweisenden Zyklus steht die Anzahl der Durchläufe nicht a–priori fest. Der Abbruchtest erfolgt nach dem Durchlauf eines Zyklus. Somit durchläuft der nichtabweisende Zyklus mindestens einmal die Anweisungen im Zyklusinneren.

Die allgemeine Form ist

```
do
{
       Anweisungen;
}
while(logischer Ausdruck);
```

**Beispiel 10.8 do–while–Schleife.** Es wird solange eine Zeichenkette von der Tastatur eingelesen, bis die Eingabe eine nichtnegative ganze Zahl ist.

```
#include <stdio.h>
#include <math.h>   /* F\"ur pow */

int main()
{
       int  i, n, zahl=0;
       char text[100];

       do
       {
              printf("Geben Sie eine nichtnegative ganze Zahl ein : ");
              fgets(text, sizeof(text), stdin);
              sscanf(text, "%s",text);

              /* Es wird nacheinander gepr\"uft ob text[i] */
              /* eine Ziffer ist. Besteht die Eingabe */
              /* nur aus Ziffern, dann wird abgebrochen */
              i=0;
              while ('0' <= text[i] && text[i] <= '9')
              {      /* ASCII */
                     i=i+1;
              }
              if (text[i] != '\0')
              {
                     printf("%c ist keine Ziffer \n",text[i]);
              }
       }
       while (text[i] != '\0');

       /* Umwandlung von String zu Integer */
       n=i;  /* Die L\"ange des Strings == i */
       for (i=0;i<=n-1;i++)
       {
              zahl=zahl+ (text[i]-'0')*pow(10,n-1-i);
       }
       printf("Die Eingabe %s die nichtnegativen ganze Zahl %i\n",text,zahl);
       return 0;
}
```

Intern behandelt der Computer Zeichenkonstanten wie `int`–Variablen. Die Zuweisung erfolgt über die ASCII–Tabelle. So entsprechen zum Beispiel die Zeichenkonstanten '0',...,'9' den Werten 48,..,57.

Zur Umwandlung von Strings in `int`–Variablen kann man auch den Befehl `strtol` nutzen, siehe `man strtol`. □

## 10.3 Anweisungen zur unbedingten Steuerungsübergabe

- `break` Es erfolgt der sofortige Abbruch der nächstäußeren `switch`–, `while`–, `do`–`while`– oder `for`–Anweisung.
- `continue` Abbruch des aktuellen und Start des nächsten Zyklus einer `while`–, `do`–`while`– oder `for`–Schleife.
- `goto marke` Fortsetzung des Programms an der mit `marke` markierten Anweisung

**Achtung!!!** Die goto–Anweisung sollte sparsam (besser gar nicht) verwendet werden, da sie dem strukturierten Programmieren zuwider läuft und den gefürchteten Spaghetticode erzeugt. In den Übungen (und in der Klausur) ist die goto–Anweisung zur Lösung der Aufgaben **nicht erlaubt**.

# Kapitel 11

# Zeiger (Pointer)

Bislang war beim Zugriff auf eine Variable nur ihr Inhalt von Interesse. Dabei war es unwichtig, wo (an welcher Speicheradresse) der Inhalt abgelegt wurde. Ein neuer Variablentyp, der Pointer (Zeiger), speichert Adressen unter Berücksichtigung des dort abgelegten Datentyps.

## 11.1 Adressen

Das folgende Programm demonstriert, wie man Speicheradressen von Variablen ermittelt.

**Beispiel 11.1 Adressen von Variablen.**

```
#include <stdio.h>

int main()
{
	int a=16;
	int b=4;
	double f=1.23;
	float g=5.23;

	/* Formatangabe %u steht f\"ur unsigned int */
	/* &a = Adresse von a */
	printf("Wert von a = %i, \t Adresse von a = %u\n",a,(unsigned int)&a);
	printf("Wert von b = %i, \t Adresse von b = %u\n",b,(unsigned int)&b);
	printf("Wert von f = %f, \t Adresse von f = %u\n",f,(unsigned int)&f);
	printf("Wert von g = %f, \t Adresse von g = %u\n",g,(unsigned int)&g);
	return 0;
}
```

Nach dem Start des Programms erscheint folgende Ausgabe:

```
Wert von a = 16,        Adresse von a = 2289604
Wert von b = 4,         Adresse von b = 2289600
Wert von f = 1.230000,  Adresse von f = 2289592
Wert von g = 5.230000,  Adresse von g = 2289588
```

□

**Bemerkung 11.2 Zu Beispiel 11.1.**

1.) Dieses Programm zeigt die Werte und die Adressen der Variablen `a,b,f,g` an. Die Adressangaben sind abhängig vom System und Compiler und variieren dementsprechend.

2.) Der Adressoperator `&` im `printf()`–Befehl sorgt dafür, dass nicht der Inhalt der jeweiligen Variable ausgegeben wird, sondern die Adresse der Variable im Speicher. Die Formatangabe `%u` dient zur Ausgabe von vorzeichenlosen Integerzahlen (`unsigned int`). Dieser Platzhalter ist hier nötig, da der gesamte Wertebereich der Ganzzahl ausgeschöpft werden soll und negative Adressen nicht sinnvoll sind.

3.) In den letzten Zeilen wird angegeben, dass die Variable `g` auf der Speicheradresse 2289588 liegt und die Variable `f` auf der Adresse 2289592. Die Differenz beruht auf der Tatsache, dass die Variable `g` vom Typ `float` zur Speicherung `sizeof(float)=4` Bytes benötigt. Auch bei den anderen Variablen kann man erkennen, wieviel Speicherplatz sie aufgrund ihres Datentyps benötigen.

□

## 11.2 Pointervariablen

Eine Pointervariable (Zeigervariable) ist eine Variable, deren Wert (Inhalt) eine Adresse ist. Die Deklaration erfolgt durch:

```
Datentyp *Variablenname;
```

Das nächste Programm veranschaulicht diese Schreibweise.

**Beispiel 11.3**

```
#include <stdio.h>

int main()
{
       int a=16;
       int *pa;      /* Deklaration von int Zeiger pa - pa ist ein Zeiger auf
                      * eine Integer*/
       double f=1.23;
       double *pf;   /* Deklaration von double Zeiger pf */

       pa=&a; /* Zeiger pa wird die Adresse von a zugewiesen */
       pf=&f; /* Zeiger pf wird die Adresse von f zugewiesen */

       printf("Variable a : Inhalt = %i\t Adresse = %u\t Gr"osse %i \n"
       ,a,(unsigned int)&a,sizeof(a));
       printf("Variable pa : Inhalt = %u\t Adresse = %u\t Gr"osse %i \n"
       ,(unsigned int)pa,(unsigned int)&pa,sizeof(pa));
       printf("Variable f : Inhalt = %f\t Adresse = %u\t Gr"osse %i \n"
       ,f,(unsigned int)&f,sizeof(f));
       printf("Variable pf : Inhalt = %u\t Adresse = %u\t Gr"osse %i \n"
       ,(unsigned int)pf,(unsigned int)&pf,sizeof(pf));
       return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Variable a : Inhalt = 16      Adresse = 2289604    Gr"osse 4
Variable pa : Inhalt = 2289604 Adresse = 2289600   Gr"osse 4
Variable f : Inhalt = 1.230000 Adresse = 2289592   Gr"osse 8
Variable pf : Inhalt = 2289592 Adresse = 2289588   Gr"osse 4
```

□

**Bemerkung 11.4 Zu Beispiel 11.3.**

1.) Da Pointervariablen wieder eine Speicheradresse besitzen, ist die Definition eines Pointers auf einen Pointer nicht nur sinnvoll, sondern auch nützlich (siehe Beispiel 11.9).

2.) Die Größe des benötigten Speicherplatzes für einen Pointer ist unabhängig vom Typ der ihm zu Grunde liegt, da der Inhalt stets eine Adresse ist. Der hier verwendete Rechner (32–Bit–System) hat einen Speicherplatzbedarf von 4 Byte (= 32 Bit).

□

## 11.3 Adressoperator und Zugriffsoperator

Der unäre Adressoperator `&` (Referenzoperator)

```
&Variablenname;
```

bestimmt die Adresse der Variable. Der unäre Zugriffsoperator `*` (Dereferenzoperator)

```
*pointer;
```

erlaubt den (indirekten) Zugriff auf den Inhalt, auf den der Pointer zeigt. Die Daten können wie Variablen manipuliert werden.

**Beispiel 11.5**

```
#include <stdio.h>

int main()
{
        int a=16;
        int b;
        int *p; /* Deklaration von int Zeiger p */

        p=&a;  /* Zeiger p wird die Adresse von a zugewiesen */
        b=*p;  /* b = Wert unter Adresse p = a = 16 */

        printf("Wert von b = %i = %i = Wert von *p \n",b,*p);
        printf("Wert von a = %i = %i = Wert von *p \n",a,*p);
        printf("Adresse von a = %u = %u = Wert von p\n"
        ,(unsigned int)&a,(unsigned int)p);
        printf("Adresse von b = %u != %u = Wert von p\n\n"
        ,(unsigned int)&b,(unsigned int)p);

        *p=*p+2; /* Wert unter Adresse p wird um 2 erh\"oht */
                 /* das heisst a=a+2                      */

        printf("Wert von b = %i != %i = Wert von *p \n",b,*p);
        printf("Wert von a = %i = %i = Wert von *p \n",a,*p);
        printf("Adresse von a = %u = %u = Wert von p\n"
        ,(unsigned int)&a,(unsigned int)p);
        printf("Adresse von b = %u != %u = Wert von p\n\n"
        ,(unsigned int)&b,(unsigned int)p);
        return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Wert von b = 16 = 16 = Wert von *p
Wert von a = 16 = 16 = Wert von *p
Adresse von a = 2289604 = 2289604 = Wert von p
Adresse von b = 2289600 != 2289604 = Wert von p

Wert von b = 16 != 18 = Wert von *p
Wert von a = 18 = 18 = Wert von *p
Adresse von a = 2289604 = 2289604 = Wert von p
Adresse von b = 2289600 != 2289604 = Wert von p
```

□

## 11.4 Zusammenhang zwischen Zeigern und Feldern

Felder nutzen das Modell des linearen Speichers, das heißt ein im Index nachfolgendes Element ist auch physisch im unmittelbar nachfolgenden Speicherbereich abgelegt. Dieser Fakt erlaubt die Interpretation von Zeigervariablen als Feldbezeichner und umgekehrt.

**Beispiel 11.6 Zeiger und Felder.**

```
#include <stdio.h>

int main()
{
       float ausgabe;
       float f[4]={1,2,3,4};
       float *pf;


       pf=f;  /* \"Aquivalent w\"are die Zuweisung pf=&f[0] */

       /* Nicht Zul\"assige Operationen mit Feldern */
       /* f=g;
        * f=f+1; */

       /* \"Aquivalente Zugriffe auf Feldelemente */
       ausgabe=f[3];
       ausgabe=*(f+3);
       ausgabe=pf[3];
       ausgabe=*(pf+3);
       return 0;
}
```

□

**Bemerkung 11.7 Zu Beispiel 11.6.**

1.) Das Beispiel zeigt, dass der Zugriff auf einzelne Feldelemente für Zeiger und Felder identisch ist, obwohl es sich um unterschiedliche Datentypen handelt.

2.) Die arithmetischen Operatoren `+` und `-` haben bei Zeigern und Feldern auch den gleichen Effekt. Der Ausdruck `pf + 3` liefert als Wert

    `(Adresse in pf) + 3 * sizeof(Typ)`
    (und nicht `(Adresse in pf) + 3` !!!)

3.) Der Zuweisungsoperator `=` ist für Felder nicht anwendbar, das heißt `f=ausdruck;` ist nicht zulässig. Einzelne Feldelemente können jedoch wie gewohnt manipuliert werden, zum Beispiel `f[2]=g[3]` ist zulässig. Die Zuweisung `pf=pf+1` hingegen bewirkt, dass `pf` nun auf `f[1]` zeigt.

4.) Ein weiterer Unterschied zwischen Feldvariablen und Pointervariablen ist der benötigte Speicherplatz. Im Beipiel liefert `sizeof(pf)` den Wert 4 und `sizeof(f)` den Wert 16 (=4 `sizeof(float)`).

5.) Die folgenden Operatoren sind auf Zeiger anwendbar:

- Vergleichsoperatoren: `==, !=, <, >, <=, >=`,
- Addition `+` und Subtraktion `-`,
- Inkrement `++`, Dekrement `--` und zusammengesetzte Operatoren `+=`, `-=`.

## 11.5 Dynamische Felder mittels Zeiger

Bisher wurde die Länge von Feldern bereits bei der Deklaration beziehungsweise Definition angegeben. Da viele Aufgaben und Probleme stets nach demselben Prinzip ausgeführt werden können, möchte man die Feldlänge gerne als Parameter und nicht als feste Größe in die Programmierung einbeziehen. Die benötigten Datenobjekte werden dann in der entsprechenden Größe und damit mit entsprechend optimalem Speicherbedarf erzeugt.
Für Probleme dieser Art bietet C mehrere Funktionen (in der Headerdatei `malloc.h`), die den notwendigen Speicherplatz zur Laufzeit verwalten. Dazu zählen:

| | |
|---|---|
| `malloc()` | reserviert Speicher einer bestimmten Größe |
| `calloc()` | reserviert Speicher einer bestimmten Größe und initialisiert die Feldelemente mit 0 |
| `realloc()` | erweitert einen reservierten Speicherbereich |
| `free()` | gibt den Speicherbereich wieder frei |

Die Funktionen `malloc()`, `calloc()` und `realloc()` versuchen, den angeforderten Speicher bereitzustellen (allokieren), und liefern einen Pointer auf diesen Bereich zurück. Konnte die Speicheranforderung nicht erfüllt werden, wird ein Null–Pointer (`NULL` in C, das heißt Pointer zeigt auf die 0–Adresse im Speicher) zurückliefert. Die Funktion `free()` enthält als Argument einen so definierten Pointer und gibt den zugehörigen Speicherbereich wieder frei.

**Beispiel 11.8 Norm des Vektors (1,...,n).** Das folgende Programm demonstriert die Reservierung von Speicher durch die Funktion `malloc()` und die Freigabe durch `free()`.

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
int main()
{
        int n, i;
        float *vektor, norm=0.0;

        printf("Geben Sie die Dimension n des Vektorraums an: ");
        scanf("%i",&n);

        /* Dynamische Speicher Reservierung */
```

```
        vektor = (float *) malloc(n*sizeof(float));

        if (vektor == NULL) /* Genuegend Speicher vorhanden? */
        {
                printf("Nicht genuegend Speicher vorhanden \n");
                return 1;
                /* Programm beendet sich und gibt einen Fehlerwert zurueck */
        }
        else
        {
                /* Initialisierung des Vektors */
                /* Norm des Vektors */
                for (i=0;i<n;i=i+1)
                {
                        vektor[i]=i+1;
                        norm=norm+vektor[i]*vektor[i];
                }
                norm=sqrt(norm);

                /* Freigabe des Speichers */
                free(vektor);
                printf("Die Norm des eingegebenen Vektors (1,...,%i) ist : %f \n"
                ,n,norm);
        }
        return 0;
}
```

□

Ein zweidimensionales dynamisches Feld, eine Matrix, lässt sich einerseits durch ein eindimensionales dynamisches Feld darstellen, als auch durch einen Zeiger auf ein Feld von Zeigern. Dies sieht für eine Matrix von $m$ Zeilen und $n$ Spalten wie folgt aus:

**Beispiel 11.9 Dynamisches 2D Feld.**

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

int main()
{
        int n,m,i,j;
        double **p;    /* Zeiger auf Zeiger vom Typ double */

        printf("Geben Sie die Anzahl der Zeilen der Matrix an: ");
        scanf("%i",&m);
        printf("Geben Sie die Anzahl der Spalten der Matrix an: ");
        scanf("%i",&n);

        /* Allokiert Speicher fuer Zeiger auf die Zeilen der Matrix */
        p=(double **) malloc(m*sizeof(double*));

        for (i=0;i<m;i++)
        {
                /* Allokiert Speicher fuer die Zeilen der Matrix */
                p[i]= (double *) malloc(n*sizeof(double));
        }

        for (i=0;i<m;i++) /* Initialisierung von Matrix p */
```

```
        {
                for (j=0;j<n;j++)
                {
                        p[i][j]=(i+1)*(j+1);
                        printf("%f ",p[i][j]);
                }
                printf("\n");
        }

        for (i=0;i<m;i++)
        {
                free(p[i]); /* Freigabe der Zeilen */
        }
        free(p);        /* Freigabe der Zeilenzeiger */
        return 0;
}
```

□

Zuerst muss der Speicher auf die Zeilenpointer allokiert werden, erst danach kann der Speicher für die einzelnen Zeilen angefordert werden. Beim Deallokieren des Speichers müssen ebenfalls alle Spalten und danach alle Zeilen wieder freigegeben werden. Für den Fall $m = 3$ und $n = 4$ veranschaulicht das Bild die Ablage der Daten im Speicher.

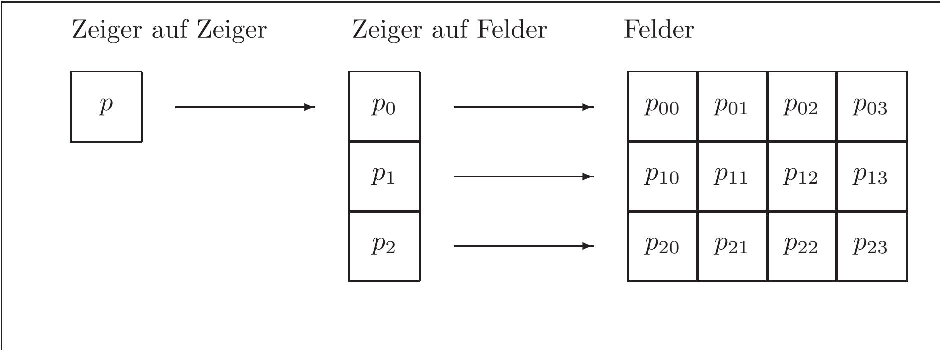


Abbildung 11.1: Dynamisches 2D Feld mit Zeiger auf Zeiger

**Achtung !** Es gibt keine Garantie, dass die einzelnen Zeilen der Matrix hintereinander im Speicher angeordnet sind. Somit unterscheidet sich die Speicherung des dynamischen 2D–Feldes von der Speicherung des statischen 2D–Feldes (siehe Kapitel 9.3.2), obwohl die Syntax des Elementzugriffes `p[i][j]` identisch ist. Dafür ist diese Matrixspeicherung flexibler, da die Zeilen auch unterschiedliche Längen haben dürfen. Insbesondere findet das dynamische 2D–Feld Anwendung zur Speicherreservierung bei der Bearbeitung von dünnbesetzten Matrizen.

Will man sich sicher sein, dass die Matrixeinträge im Speicher hintereinander kommen, so allokiere man zuerst den Speicher für die gesamte Matrix und weise dann den Zeigern auf die Zeilen die entsprechenden Zeilenanfänge zu.

# Kapitel 12

# Funktionen

Ein C–Programm gliedert sich ausschließlich in Funktionen. Beispiele für Funktionen wurden bereits vorgestellt:

- Die Funktion `main()`, die den Ausführungsbeginn des Programms markiert und somit das Hauptprogramm darstellt,
- Bibliotheksfunktionen, die häufiger benötigte höhere Funktionalität bereitstellen (zum Beispiel `printf()`, `scanf()`, `sqrt()`, `strcpy()` etc).

C verfügt nur über einen sehr kleinen Sprachumfang, stellt jedoch eine Vielzahl an Funktionen in Bibliotheken für fast jeden Bedarf bereit. Was aber, wenn man eine Funktion für eine ganz spezielle Aufgabe benötigt und nichts Brauchbares in den Bibliotheken vorhanden ist? Ganz einfach: Man schreibt sich diese Funktionen selbst.

## 12.1 Deklaration, Definition und Rückgabewerte

Die Deklaration einer Funktion hat die Gestalt

```
Datentyp Funktionsname(Datentyp1,...,DatentypN);
```

Die Deklaration beginnt mit dem Datentyp des Rückgabewertes, gefolgt vom Funktionsnamen. Es folgt eine Liste der Datentypen der Funktionsparameter. Bei der Deklaration können auch die Namen für die Funktionsparameter vergeben werden:

```
Datentyp Funktionsname(Datentyp1 Variable1,...,DatentypN VariableN);
```

Die Deklaration legt jedoch nur das Allernotwendigste fest, so ist zum Beispiel noch nichts darüber gesagt, was mit den Funktionsparametern im Einzelnen geschieht und wie der Rückgabewert gebildet wird. Diese wichtigen Aspekte werden in der Definition der Funktion behandelt:

```
Datentyp Funktionsname(Datentyp1 Variable1,...,DatentypN VariableN)
{
  Deklaration der Funktionsvariablen;
  Anweisungen;
}
```

Der Funktionsrumpf besteht gegebenenfalls aus Deklarationen von weiteren Variablen (zum Beispiel für Hilfsgrößen oder den Rückgabewert) sowie Anweisungen.

Folgendes ist bei Funktionen zu beachten:

- Die Deklaration beziehungsweise Definition von Funktionen wird außerhalb jeder anderen Funktion, speziell `main()`, vorgenommen.
- Funktionen müssen vor ihrer Verwendung zumindestens deklariert sein. Unterlässt man dies, so nimmt der Compiler eine implizite Deklaration mit Standardrückgabewert `int` vor, was eine häufige Ursache für Laufzeitfehler darstellt.
- Deklaration/Definition können in beliebiger Reihenfolge erfolgen.
- Deklaration und Definition müssen konsistent sein, das heißt die Datentypen für Rückgabewert und Funktionsparameter müssen übereinstimmen.

**Beispiel 12.1 Funktion.**

```
#include <stdio.h>

/***********************************/
/* Deklaration der Maximum-Funktion */
/***********************************/
float maximum (float, float);

/*****************************************************/
/* Hauptprogramm                                     */
/*****************************************************/
int main()
{
	float a=3.0,b=2.0;
	printf("Das Maximum von %f und %f ist %f\n",a,b,maximum(a,b));
	return 0;
}

/*********************************************************/
/* Definition der Maximum-Funktion                       */
/*********************************************************/
float maximum (float x, float y)
{
	/* Die Funktion maximum() erstellt Kopien
	 * der Funktionswerte (reserviert neuen Speicher)
	 * und speichert sie in x beziehungsweise y */
	float maxi;

	if (x>y) maxi=x;
	else maxi=y;

	return maxi;          /* R\"uckgabewert der Funktion */
}
```

□

## 12.2 Lokale und globale Variablen

Variablen lassen sich nach ihrem Gültigkeitsbereich unterteilen:

- *lokale Variablen:* Sie gelten in dem Anweisungsblock, zum Beispiel Funktionenrumpf oder Schleifenrumpf, in dem sie deklariert wurden. Für diesen Block gelten sie als lokal. Bei der Ausführung des Programms existieren die Variablen bis zum Verlassen des Anweisungsblocks.
- *globale Variablen:* Sie werden außerhalb aller Funktionen deklariert/definiert, zum Beispiel direkt nach den Präprozessordirektiven, und sind zunächst im gesamten Programm einschließlich aller Funktionen gültig. Dies bedeutet

speziell, dass jede Funktion sie verändern kann. Achtung: unvorhergesehener Programmablauf ist möglich!

Variablen die auf einer höheren Ebene (global oder im Rumpf einer aufrufenden Funktion) deklariert/definiert wurden, können durch Deklaration gleichnamiger lokaler Variablen im Rumpf einer aufgerufenen Funktion „verdeckt“ werden. In diesem Zusammenhang spricht man von Gültigkeit beziehungsweise Sichtbarkeit von Variablen.

**Beispiel 12.2 Gültigkeitsbereich von Variablen.**

```
/***********************************************/
/* Im Beispiel wird 4 x die Variable a deklariert
/***********************************************/
#include <stdio.h>

/************************************/
/* Deklaration der Funktion summe() */
/************************************/
int  summe (int, int);

/*******************************************************/
/* Deklaration von globalen Variablen                */
/*******************************************************/
int a = 1;

/*******************************************************/
/* Hauptprogramm                                     */
/*******************************************************/
int main()
{
    printf("start %d\n",a);
    int a=2;   /* a in main() = 2 und ueberdeckt globales a*/
    printf("1.stelle %d\n",a);
    {
        int a=2; /* lokales a in main() = 2 und ueberdeckt a in main() */
        printf("2.stelle %d\n",a);
        a=a+1;   /* lokales a in main() wird um 1 erh\"oht */
        printf("3.stelle %d\n",a);
        a=summe(a,a); /* lokales a in main() = 2 x lokales a in main()
                       * Gleichzeitig wird das globale a in
                       * der Funktion summe um 1 erh\"oht */
        /* lokales a in main() wird gel\"oscht */
        printf("4.stelle %d\n",a);
    }
    printf("5.stelle %d\n",a);
    a=summe(a,a);      /* a in main = 2 x lokales a in main()
                        * Gleichzeitig wird das globale a in
                        * der Funktion summe um 1 erh\"oht */
    printf("6.stelle %d\n",a);

    return 0;
}

/***************************************************************/
/* Definition Summen-Funktion */
/***************************************************************/
int summe (int x, int y)
{
    /* Die Funktion summe() erstellt Kopien
```

```
     * der Funktionswerte (reserviert neuen Speicher)
     * und speichert sie in x beziehungsweise y */

    printf("summe: 1.stelle %d\n",a);
    a=a+1; /* globales a wird um 1 erh\"oht */
    printf("summe: 2.stelle %d\n",a);

    int a=0;   /* a in der Funktion summe() */
    printf("summe: 3.stelle %d\n",a);
    a=x+y;            /* a in der Funktion summe = x+y */
    printf("summe: 4.stelle %d\n",a);

    return a;  /* a in der Funktion wird zur\"uckgegeben */
               /* a,x,y in der Funktion summe werden gel\"oscht */
}
```

Die Ausgabe ist

```
start 1
1.stelle 2
2.stelle 2
3.stelle 3
summe: 1.stelle 1
summe: 2.stelle 2
summe: 3.stelle 0
summe: 4.stelle 6
4.stelle 6
5.stelle 2
summe: 1.stelle 2
summe: 2.stelle 3
summe: 3.stelle 0
summe: 4.stelle 4
6.stelle 4
```

Dieses Beispiel zeigt, dass man sehr gut aufpassen muss, wenn man den gleichen Namen für unterschiedliche Variablen nutzt. Auch wegen der Übersichtlichkeit empfiehlt es sich, für jede Variable einen anderen Namen zu verwenden. □

## 12.3 Call by value

Die Standardübergabe von Funktionsparametern geschieht folgendermaßen: An die Funktion werden Kopien der Variablen als Parameter übergeben und von dieser zur Verarbeitung genutzt. Die ursprüngliche Variable bleibt von den in der Funktion vorgenommenen Manipulationen unberührt, es sei denn, sie wird durch den Rückgabewert der Funktion überschrieben.

**Beispiel 12.3 Call by value I.**

```
#include <stdio.h>

/************************************/
/* Deklaration der Funktion setze() */
/************************************/
void setze (int);

/****************************/
/* Hauptprogramm           */
```

```
/****************************/
int main()
{
        int b=0;
        setze(b);
        printf("b=%i\n",b);
        return 0;
}

/***********************************/
/* Definition der Funktion setze () */
/***********************************/
void setze (int b)
{
        b=3;
}
```

Die Ausgabe lautet:

```
b=0
```

Die Funktion `setze()` hat nur eine Kopie der Variablen `b` als Parameter erhalten und auf einen neuen Wert gesetzt. Die eigentliche Variable im Hauptprogramm behielt ihren Wert. □

**Beispiel 12.4 Call by value II.**

```
#include <stdio.h>

/***********************************/
/* Deklaration der Funktion setze() */
/***********************************/
int setze (int);

/****************************/
/* Hauptprogramm          */
/****************************/
int main()
{
        int b=0;
        b=setze(b);
        printf("b=%i\n",b);
        return 0;
}

/***********************************/
/* Definition der Funktion setze () */
/***********************************/
int setze (int b)
{
        b=3;
        return b;
}
```

Die Ausgabe lautet:

```
b=3
```

Die Funktion `setze()` hat wieder nur eine Kopie der Variablen `b` als Parameter erhalten und auf einen neuen Wert gesetzt. Durch das Zurückliefern und die Zuweisung an die eigentliche Variable `b` wurde die Änderung wirksam. □

## 12.4 Call by reference

Bei *call by reference* wird der Funktion nicht eine Kopie der Variablen selbst, sondern in Form eines Pointers auf die Variable eine *Kopie der Adresse der Variablen* übergeben. Über die Kenntnis der Variablenadresse kann die Funktion den Variableninhalt manipulieren. Hierzu kommen beim Aufruf der Funktion der Adressoperator und im Funktionsrumpf der Inhaltsoperator in geeigneter Weise zum Einsatz.

**Beispiel 12.5 Call by reference.**

```
#include <stdio.h>

/************************************/
/* Deklaration der Funktion setze() */
/************************************/
void setze (int *);

/*****************************/
/* Hauptprogramm           */
/*****************************/
int main()
{
       int b=0;
       setze(&b);
       printf("b=%i\n",b);
       return 0;
}

/************************************/
/* Definition der Funktion setze () */
/************************************/
void setze (int *b)
{
       *b=3;
}
```

Die Ausgabe lautet:

```
b=3
```

Der Funktion `setze()` wird ein Zeiger auf eine `int`–Variable übergeben und sie verwendet den Inhaltsoperator `*`, um den Wert der entsprechenden Variablen zu verändern. Im Hauptprogramm wird der Zeiger mit Hilfe des Adressoperators `&` erzeugt. □

## 12.5 Rekursive Programmierung

Bisher haben Funktionen ihre Aufgabe in einem Durchgang komplett erledigt. Eine Funktion kann ihre Aufgabe aber manchmal auch dadurch erledigen, dass sie sich selbst mehrmals aufruft und jedes Mal nur eine Teilaufgabe löst. Das führt zu rekursiven Aufrufen.

**Beispiel 12.6 Rekursive Programmierung.** Das folgende Programm berechnet $x^k$, $x \in \mathbb{R}$, $k \in \mathbb{Z}$, rekursiv.

```
#include <stdio.h>

/* Deklaration potenz-Funktion */
```

```
double potenz (double, int);

/* Hauptprogramm */
int main()
{
        double x;
        int k;
        printf("Zahl x : ");  scanf("%lf",&x);
        printf("Potenz k : "); scanf("%i",&k);
        printf("x^k = %f \n",potenz(x,k));

        return 0;
}

/* Definition potenz-Funktion */
double potenz (double x, int k)
{
        if (k<0) /* Falls k < 0 berechne (1/x)^(-k) */
        {
                return potenz(1.0/x,-k);
        }
        else
        {
                if (k==0) /* Rekursionsende */
                {
                        return 1;
                }
                else
                {
                        return x*potenz(x,k-1); /* Rekursionsaufruf */
                }
        }
}
```

Dieser Rekursion liegt die Darstellung

$$x^k = \underbrace{x(x(x(\ldots 1)))}_{k}$$

zu Grunde. Die Funktion `potenz()` ruft sich solange selbst auf, bis der Fall `k==0` eintritt. Das Ergebnis dieses Falls liefert sie an die aufrufende Funktion zurück. □

**Achtung !** Bei der rekursiven Programmierung ist stets darauf zu achten, dass der Fall des Rekursionsabbruchs (im Beispiel `k==0`) immer erreicht wird, da sonst die Maschine bis zum nächsten Stromausfall oder bis der Speicher voll ist (mit jedem Funktionsaufruf werden ja lokale Variablen angelegt) rechnet.

## 12.6 Kommandozeilen–Parameter

Ausführbaren Programmen können beim Aufruf Parameter übergeben werden, indem man nach dem Programmnamen eine Liste der Parameter (Kommandozeilen–Parameter) anfügt. In C–Programmen können so der Funktion `main` Parameter übergeben werden. Zur Illustration wird folgendes Beispiel betrachtet.

**Beispiel 12.7 Kommandozeilen–Parameter.**

```
/* 1  */ # include <stdio.h>
```

```
/* 2 */
/* 3 */ int main(int argc, char* argv[])
/* 4 */ {
/* 5 */       int zaehler;
/* 6 */       float zahl,summe=0;
/* 7 */
/* 8 */       for (zaehler=0;zaehler < argc ; zaehler++)
/* 9 */       {
/* 10 */             printf("Parameter %i = %s \n",zaehler,argv[zaehler]);
/* 11 */       }
/* 12 */       printf("\n");
/* 13 */       for (zaehler=1;zaehler < argc ; zaehler++)
/* 14 */       {
/* 15 */             sscanf(argv[zaehler],"%f",&zahl);
/* 16 */
/* 17 */             summe=summe+zahl;
/* 18 */       }
/* 19 */       printf("Die Summe der Kommandozeilen-Parameter : %f\n",summe);
/* 20 */
/* 21 */       return 0;
/* 22 */ }
```

Nach dem Start des obigen Programms zum Beispiel durch

```
a.out 1.4 3.2 4.5
```

erscheint folgende Ausgabe auf dem Bildschirm

```
Parameter 0 = a.out
Parameter 1 = 1.4
Parameter 2 = 3.2
Parameter 3 = 4.5

Die Summe der Kommandozeilen-Parameter : 9.100000
```

Die Angaben in der Kommandozeile sind an das Programm übergeben worden und konnten hier auch verarbeitet werde.

*Zeile 3:* `main` erhält vom Betriebssystem zwei Parameter. Die Variable `argc` enthält die Anzahl der übergebenen Parameter und `argv[]` die Parameter selbst. Die Namen der Variablen `argc` und `argv` sind natürlich frei wählbar, es hat sich jedoch eingebürgert, die hier verwendeten Bezeichnungen zu benutzen. Sie leiten sich von *arg*ument *c*ount und *arg*ument *v*alues ab.

Bei `argc` ist eine Besonderheit zu beachten. Hat diese Variable zum Beispiel den Wert 1 ist, so bedeutet das, dass kein Kommandozeilen–Parameter eingegeben wurde. Das Betriebssystem übergibt als ersten Parameter nämlich grundsätzlich den Namen des Programms selbst. Also erst wenn `argc` größer als 1 ist, wurde wirklich ein Parameter eingegeben.

Die Deklaration `char *argv[]` bedeutet: Zeiger auf Zeiger auf Zeichen. Man hätte auch `char **argv` schreiben können. Mit anderen Worten: `argv` ist ein Zeiger, der auf ein Feld zeigt, das wiederum Zeiger enthält. Diese Pointer zeigen schließlich auf die einzelnen Kommandozeilen–Parameter. Die leeren eckigen Klammern weisen darauf hin, dass es sich um ein Feld unbestimmter Größe handelt. Die einzelnen Argumente können durch Indizierung von `argv` angesprochen werden. `argv[1]` zeigt also auf das erste Argument ("1.4"), `argv[2]` auf "3.2"  und so weiter.

*Zeile 15:* Da es sich bei `argv[i]` um Strings handelt müssen die Eingabeparameter eventuell (je nach ihrer Bestimmung) in einen anderen Typ umgewandelt werden. Dies geschieht in diesem Beispiel mit Hilfe des `sscanf`–Befehls. □

## 12.7 Wie werden Deklarationen gelesen?

Eine Deklaration besteht grundsätzlich aus einem *Bezeichner* (Variablennamen oder Funktionsnamen), der durch einen oder mehrere *Zeiger*–, *Feld*– oder *Funktions*–Modifikatoren beschrieben wird. Wenn mehrere solcher Modifikatoren miteinander kombinieren, muss man darauf achten, dass Funktionen keine Funktionen oder Felder zurückgeben können und dass Felder auch keine Funktionen als Elemente haben können. Ansonsten sind alle Kombinationen erlaubt. Dabei haben Funktions– und Array–Modifikatoren Vorrang vor Zeiger–Modifikatoren. Durch Klammerung kann diese Rangfolge geändert werden.

Bei der Interpretation beginnt man am besten beim *Bezeichner* und liest nach rechts bis zum Ende beziehungsweise bis zu einer einzelnen rechten Klammer. Dann fährt man links vom Bezeichner mit eventuell vorhandenen Zeiger–Modifikatoren fort, bis das Ende oder eine einzelne linke Klammer erreicht wird. Dieses Verfahren wird für jede geschachtelte Klammer von innen nach außen wiederholt. Zum Schluss wird der Typ–Kennzeichner gelesen.

**Beispiel 12.8**

$$\underbrace{\texttt{char}}_{7}\quad \underbrace{*}_{6}\quad (\underbrace{*}_{4}\quad (\underbrace{*}_{2}\quad \underbrace{\texttt{Bezeichner}}_{1})\quad \underbrace{(\;)}_{3}\quad )\quad \underbrace{[20]}_{5}$$

`Bezeichner` (1) ist hier ein Zeiger (2) auf eine Funktion (3) ohne Eingabe–Argumente, die einen Zeiger (4) auf ein Feld mit 20 Elementen (5) zurückgibt, die Zeiger (6) auf `char`–Werte (7) sind! □

**Beispiel 12.9** Die folgenden vier Beispiele verdeutlichen den Einsatz von Klammern:

| | | |
|---|---|---|
| (i) | `char * a[10]` | `a` ist ein Feld der Größe 10 mit Zeigern auf `char`–Werte |
| (ii) | `char (* a)[10]` | `a` ist Zeiger auf ein Feld der Größe 10 mit `char`–Werte |
| (iii) | `char *a(int)` | `a` ist Funktion die als Eingabeparameter einen `int`–Wert verlangt und einen Zeiger auf `char`–Wert zurückgibt |
| (iv) | `char (*a)(int)` | `a` ist Zeiger, auf eine Funktion die als Eingabeparameter einen `int`–Wert verlangt und einen `char`–Wert zurückgibt |

□

## 12.8 Zeiger auf Funktionen

Manchmal ist es nützlich, Funktion an Funktionen zu übergeben. Dies kann mit Hilfe von Zeigern auf Funktionen realisiert werden.

**Beispiel 12.10 Zeiger auf Funktionen.** In diesem Beispiel wird der Funktion `trapez_regel` ein Zeiger auf die zu integrierende Funktion mitgeliefert. Dadurch ist es möglich, beliebige Funktionen mit Hilfe der Trapez–Regel numerisch zu integrieren. Die Intervallgrenzen und Anzahl der Stützstellen sollen dem Programm durch Kommandozeilen–Parameter übergeben werden.

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

/***************************************************************/
/* Deklaration der Funktion trapez_regel()             */
/* Eingabeparameter : 1.) Zeiger auf Funktion mit
 *                        Eingabeparameter double-Wert
```

```
 *                      und double-Rueckgabewert
 *                   2.) double fuer linke Intervallgrenze
 *                   3.) double fuer rechte Intervallgrenze
 *                   4.) int fuer Anzahl der Stuetzstellen
 * Rueckgabewert : double fuer das Integral
*************************************************************/
double trapez_regel(double (*f)(double), double, double, int);

/*********************************************************************/
/* Hauptprogramm                                                */
/*********************************************************************/
int main(int argc,char** argv)
{
       int n;
       double a,b,integral;

       /* Zeiger auf eine Funktion die als Rueckgabewert
        * eine double-Variable besitzt und als
        * Eingabe eine double-Variable verlangt    */
       double (*fptr)(double);

       if (argc<4)
       {
              printf("Programm ben\"otigt 3 Kommandozeilenparameter :\n");
              printf("1.) Linker Intervallrand (double)\n");
              printf("2.) Rechter Intervallrand (double)\n");
              printf("3.) Anzahl der Teilintervalle fuer");
              printf(" numerische Integration (int)\n");

              return 1;
       }
       else
       {
              sscanf(argv[1],"%lf",&a);
              sscanf(argv[2],"%lf",&b);
              sscanf(argv[3],"%i",&n);


              fptr=(double(*)(double)) cos;/* Zeiger fptr auf cos-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der cos-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
              printf(" Stuetzstellen)\n");
              printf("        \t %f (exakt)\n\n",sin(b)-sin(a));

              fptr=(double (*)(double)) sin;/*Zeiger fptr auf sin-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der sin-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
              printf(" Stuetzstellen)\n");
              printf("        \t %f (exakt)\n\n",cos(a)-cos(b));

              return 0;
       }
}
```

```
/******************************************************************/
/* Definition der Funktion trapez_regel()                        */
/******************************************************************/
double trapez_regel(double (*f)(double ),double a,double b,int n)
{
       n=n+1;
       int k;
       double h=(b-a)/n;
       double integral=0;

       for (k=0;k<=n-1;k++) integral=integral+h/2*(f(a+k*h)+f(a+(k+1)*h));

       return integral;
}
```

□

# Kapitel 13

# Strukturierte Datentypen (∗)

Über die normalen Datentypen hinaus gibt es in C weitere, komplexere Typen, die sich aus den einfacheren zusammensetzen. Außerdem besteht die Möglichkeit, eigene Synonyme für häufig verwendete Typen festzulegen.

| | |
|---|---|
| `Feld (array)` | Zusammenfassung von Elementen gleichen Typs |
| `Struktur (struct)` | Zusammenfassung von Elementen verschiedenen Typs |
| `Union (union)` | Überlagerung mehrerer Komponenten verschiedenen Typs auf dem gleichen Speicherplatz |
| `Aufzählungstyp (enum)` | Grunddatentyp mit frei wählbarem Wertebereich |

Der Typ Feld wurde bereits in Kapitel 9.3 vorgestellt.

## 13.1 Strukturen

Die Struktur definiert einen neuen Datentyp, welcher Komponenten unterschiedlichen Typs vereint. Die Länge einer solchen Struktur ist gleich der Gesamtlänge der einzelnen Bestandteile. Angewendet werden Strukturen häufig dann, wenn verschiedene Variablen logisch zusammengehören, wie zum Beispiel Name, Vorname, Straße etc. bei der Bearbeitung von Adressen. Die Verwendung von Strukturen ist nicht zwingend nötig, man kann sie auch durch die Benutzung von mehreren einzelnen Variablen ersetzen. Sie bieten bei der Programmierung jedoch einige Vorteile, da sie die Übersichtlichkeit erhöhen und die Bearbeitung vereinfachen.

### 13.1.1 Deklaration von Strukturen

Zur Deklaration einer Struktur benutzt man das Schlüsselwort `struct`. Ihm folgt der Name der Struktur. In geschweiften Klammern werden dann die einzelnen Variablen aufgeführt, aus denen die Struktur bestehen soll.
*Achtung:* Die Variablen innerhalb einer Stuktur können nicht initialisiert werden:

```
struct Strukturname
{
  Datendeklaration
};
```

Im folgenden Beispiel wird eine Stuktur mit Namen Student deklariert, die aus einer `int`–Variablen und einem String besteht.

**Beispiel 13.1 Deklaration einer Struktur.**

```
struct Student
       {
              int matrikel;
              char name[16];
       };
```

□

### 13.1.2 Definition von Strukturvariablen

Die Strukturschablone selbst hat noch keinen Speicherplatz belegt, sie hat lediglich ein Muster festgelegt, mit dem die eigentlichen Variablen definiert werden. Die Definition erfolgt genau wie bei den Standardvariablen (int a, double summe etc.), indem man den Variablentyp (struct Student) gefolgt von einem oder mehreren Variablennamen angibt, zum Beispiel:

```
struct Student peter, paul;
```

Hierdurch werden zwei Variablen peter, paul deklariert, die beide vom Typ struct Student sind. Neben dieser beschriebenen Methode gibt es noch eine weitere Form der Definition von Strukturvariablen:

**Beispiel 13.2 Deklaration einer Struktur und Definition der Strukturvariablen.**

```
struct Student
       {
              int matrikel;
              char name [16];
       } peter, paul;
```

In diesem Fall erfolgt die Definition der Variablen zusammen mit der Deklaration. Dazu müssen nur eine oder mehrere Variablen direkt hinter die Deklaration gesetzt werden. □

### 13.1.3 Felder von Strukturen

Eine Struktur wird häufig nicht nur zur Aufnahme eines einzelnen Datensatzes, sondern zur Speicherung vieler gleichartiger Sätze verwendet, beispielsweise mit

```
struct Student Studenten2006[1000];
```

Der Zugriff auf einzelne Feldelemente erfolgt wie gewohnt mit eckigen Klammern [].

### 13.1.4 Zugriff auf Strukturen

Um Strukturen sinnvoll nutzen zu können, muss man ihren Elementen Werte zuweisen und auf diese Werte auch wieder zugreifen können. Zu diesem Zweck kennt C den Strukturoperator '.', mit dem jeder Bestandteil einer Struktur direkt angesprochen werden kann.

**Beispiel 13.3 Deklaration, Definition und Zuweisung.**

```
#include <stdio.h>
#include <string.h> /* Fuer strcpy */

int main()
{
```

```
        /* Deklaration der Struktur Student */
        struct Student
        {
                int matrikel;
                char Vorname[16];
        };

        /* Definition eines Feldes der Struktur Student */
        struct Student Mathe[100];

        /* Zugriff auf die einzelnen Elemente */

        Mathe[0].matrikel=242834;
        strcpy(Mathe[0].Vorname,"Peter");

        Mathe[1].matrikel=343334;
        strcpy(Mathe[1].Vorname,"Paul");

        /* Der Zuweisungsoperator = ist auf Strukturen
           gleichen Typs anwendbar */

        Mathe[2]=Mathe[1];
        printf("Vorname von Student 2: %s\n",Mathe[2].Vorname);

        return 0;
}
```

□

### 13.1.5 Zugriff auf Strukturen mit Zeigern

Wurde eine Struktur deklariert, so kann man auch einen Zeiger auf diesen Typ wie gewohnt definieren, wie zum Beispiel

```
struct Student *p;
```

Der Zugriff auf den Inhalt der Adresse, auf die der Zeiger verweist, erfolgt wie üblich durch den Zugriffsoperator `*`. Wahlweise kann auch der Auswahloperator `->` benutzt werden.

**Beispiel 13.4 Zugriff mit Zeigern.**

```
#include <stdio.h>
#include <string.h> /* Fuer strcpy */

int main()
{
        /* Deklaration der Struktur Student */
        struct Student
        {
                int matrikel;
                char Strasse[16];
        };

        /* Definition Strukturvariablen peter, paul */
        struct Student peter, paul;
        struct Student *p;    /* Zeiger auf Struktur Student */

        p=&peter; /* p zeigt auf peter */
```

```
        /* Zugriff auf die einzelnen Elemente */

        (*p).matrikel=242834;
        strcpy((*p).Strasse,"Finkenweg 4");

        /* Alternativ */

        p=&paul;
        p->matrikel=423323;
        strcpy(p->Strasse,"Dorfgosse 2");

        printf("Paul, Matr.-Nr.: %i , Str.: %s\n "
        ,paul.matrikel,paul.Strasse);
        return 0;
}
```

□

### 13.1.6 Geschachtelte Strukturen

Bei der Deklaration von Strukturen kann innerhalb einer Struktur eine weitere Struktur eingebettet werden. Dabei kann es sich um eine bereits an einer anderen Stelle deklarierte Struktur handeln, oder man verwendet an der entsprechenden Stelle nochmals das Schlüsselwort struct und deklariert eine Struktur innerhalb der anderen.

**Beispiel 13.5 Geschachtelte Strukturen.**

```
#include <stdio.h>

int main()
{
        struct Punkt3D
        {
                float x;
                float y;
                float z;
        };

        struct Strecke3D
        {
                struct Punkt3D anfangspunkt;
                struct Punkt3D endpunkt;
        };

        struct Strecke3D s;

        /* Initialisierung einer Strecke */

        s.anfangspunkt.x=1.0;
        s.anfangspunkt.y=-1.0;
        s.anfangspunkt.z=2.0;
        s.endpunkt.x=1.1;
        s.endpunkt.y=1.2;
        s.endpunkt.z=1.4;
        printf("%f \n",s.endpunkt.z);
        return 0;
}
```

□

### 13.1.7 Listen

Wie in Abschnitt 13.1.6 bereits gezeigt wurde, können Strukturen auch andere (zuvor deklarierte) Strukturen als Komponenten enthalten. *Eine Struktur darf sich aber nicht selbst als Variable enthalten!*. Allerdings darf eine Struktur einen Zeiger auf sich selbst als Komponente beinhalten. Diese Datenstrukturen kommen zum Einsatz, wenn man nicht im voraus wissen kann, wieviel Speicher man für eine Liste von Datensätzen reservieren muss und daher die Verwendung von Feldern unzweckmäßig ist.

**Beispiel 13.6 Liste.**

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

int main()
{
        struct Student
        {
                char name[16];
                char familienstand[16];
                char geschlecht[16];
                struct Student *next;
        };

        struct Student Mathe[2];
        struct Student Bio[1];
        struct Student *startzeiger,*eintrag;

        /* Initialisierung der Mathe-Liste */
        strcpy(Mathe[0].name,"Kerstin");
        strcpy(Mathe[0].familienstand,"ledig\t");
        strcpy(Mathe[0].geschlecht,"weiblich");
        Mathe[0].next=&Mathe[1]; /* next zeigt auf den n\"achsten Eintag */

        strcpy(Mathe[1].name,"Claudia");
        strcpy(Mathe[1].familienstand,"verheiratet");
        strcpy(Mathe[1].geschlecht,"weiblich");
        Mathe[1].next=NULL; /* next zeigt auf NULL, d.h Listenende */

        /* Initialisierung der Bio-Liste */
        strcpy(Bio[0].name,"Peter");
        strcpy(Bio[0].familienstand,"geschieden");
        strcpy(Bio[0].geschlecht,"maennlich");
        Bio[0].next=NULL; /* next zeigt auf NULL, d.h Listenende */

        /* Ausgabe der Mathe-Liste */
        startzeiger=&Mathe[0];

        printf("Name\tFamilienstand\tGeschlecht\n\n");
        for (eintrag=startzeiger;eintrag!=NULL;eintrag=eintrag->next)
        {
                printf("%s\t%s\t%s\n"
                ,eintrag->name,eintrag->familienstand,eintrag->geschlecht);
        }
```

```
        /* Anh\"angen der Bio-Liste an die Mathe-Liste */
        Mathe[1].next=&Bio[0];

        /* Ausgabe der Mathe-Bio-Liste */
        printf("\n\nName\tFamilienstand\tGeschlecht\n\n");
        for (eintrag=startzeiger;eintrag!=NULL;eintrag=eintrag->next)
        {
                printf("%s\t%s\t%s\n"
                ,eintrag->name,eintrag->familienstand,eintrag->geschlecht);
        }
        return 0;
}
```

□

## 13.2 Unions

Während die Struktur sich dadurch auszeichnet, dass sie sich aus mehreren verschiedenen Datentypen zusammensetzt, ist das Charakteristische an Unions, dass sie zu verschiedenen Zeitpunkten jeweils einen bestimmten Datentyp aufnehmen können.

**Beispiel 13.7 Union.** Um die Klausurergebnisse von Studierenden zu speichern, müsste normalerweise zwischen Zahlen (Notenwerte im Falle des Bestehens) und Zeichenketten („nicht bestanden“) unterschieden werden. Verwendet man eine Unionvariable so kann man die jeweilige Situation flexibel handhaben

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

int main()
{
        union klausurresultat
                {
                        float note;
                        char nichtbestanden[16];
                };

        union klausurresultat ergebnis, *ergebnispointer;

        /* Zugriff mit pointer */
        ergebnispointer=&ergebnis;
        ergebnispointer->note=1.7;
        printf("Note %.1f :",ergebnispointer->note);
        strcpy(ergebnispointer->nichtbestanden,"bestanden");
        printf(" %s\n",ergebnispointer->nichtbestanden);

        /* Zugriff ohne pointer */
        ergebnis.note=5.0;
        printf("Note %.1f :",ergebnispointer->note);
        strcpy(ergebnis.nichtbestanden,"nicht bestanden");
        printf(" %s\n",ergebnispointer->nichtbestanden);

        return 0;
}
```

□

Der Speicherplatzbedarf einer Union richtet sich nach der größten Komponente (im Beispiel 13.7: 16 `sizeof(char)` = 16 Byte).

Gründe für das Benutzen von Unions fallen nicht so stark ins Auge wie bei Strukturen. Im wesentlichen sind es zwei Anwendungsfälle, in denen man sie einsetzt:

- Man möchte auf einen Speicherbereich auf unterschiedliche Weise zugreifen. Dies könnte bei der obigen Union doppelt der Fall sein. Hier kann man mit Hilfe der Komponente `nichtbestanden` auf die einzelnen Bytes der Komponente `note` zugreifen.
- Man benutzt in einer Struktur einen Bereich für verschiedene Aufgaben. Sollen beispielsweise in einer Struktur Mitarbeiterdaten gespeichert werde, so kann es sein, dass für den einen Angaben zum Stundenlohn in der Struktur vorhanden sein sollen, während der andere Speicherplatz für ein Monatsgehalt und der Dritte noch zusätzlich Angaben über Provisionen benötigt. Damit man nun nicht alle Varianten in die Struktur einbauen muss, wobei jeweils zwei unbenutzt blieben, definiert man einen Speicherbereich, in dem die jeweils benötigten Informationen abgelegt werden, als Union.

## 13.3 Aufzählungstyp

Der Aufzählungstyp ist ein Grundtyp mit frei wählbarem Wertebereich. Veranschaulicht wird dies am Beipiel der Wochtage.

**Beispiel 13.8 Aufzählungstyp.**

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

int main()
{
	enum tag
		{
			montag, dienstag, mittwoch, donnerstag,
			freitag, samstag, sonntag
		};

	enum tag wochentag;
	wochentag=montag;
	if (wochentag==montag) printf("Endlich wieder Vorlesungen!\n");
	return 0;
}
```

□

## 13.4 Allgemeine Typendefinition

Das Schlüsselwort `typedef` definiert neue Namen für bestehende Datentypen. Es erlaubt eine kürzere Schreibweise bei aufwendigen Deklarationen und kann Datentypen auf Wunsch aussagekräftigere Namen geben. Die Syntax für die Definition eines neuen Datentypnamens sieht wie folgt aus:

```
typedef typ Variablenname;
```

**Beispiel 13.9 Typendefinition.**

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

```
int main()
{
        struct vektor
                {
                        float x;
                        float y;
                        float z;
                };

        typedef char text[100];
        typedef struct vektor punkt;

        /* Deklaration der Variablen p und nachricht */
        punkt p;
        text nachricht;

        /* Initialisierung der Variablen p und nachricht */

        p.x=1.0;p.y=0.0;p.z=0.0;
        strcpy(nachricht,"nachricht ist eine Variable vom Typ text");
        printf("%s\n",nachricht);
        return 0;
}
```

Interessanterweise ist eine Variable vom Typ `text` nunmehr stets eine Zeichenkette der (max.) Länge 100. Weiterhin zeigt das Beispiel, dass auch strukturierten Datentypen eigene Typdefinitionen zugewiesen werden können. □

# Anhang A

# Computerarithmetik (∗)

## A.1 Zahlendarstellung im Rechner und Computerarithmetik

Prinzipiell ist die Menge der im Computer darstellbaren Zahlen endlich. Wie „groß“ diese Menge ist, hängt von der Rechnerarchitektur ab. So sind bei einer 32Bit-Architektur zunächst maximal $2^{32} = 4294967296$ ganze Zahlen (ohne Vorzeichen) darstellbar. Innerhalb dieser Grenzen stellt die Addition und Multiplikation von ganzen Zahlen kein Problem dar. Ganz anders verhält es sich mit rationalen und erst recht irrationalen Zahlen: Jede Zahl $x \neq 0$ lässt sich bekanntlich darstellen als

$$x = \operatorname{sgn}(x) B^N \sum_{n=1}^{\infty} x_{-n} B^{-n}, \tag{A.1}$$

wobei $2 \leq B \in \mathbb{N}, N \in \mathbb{Z}$ und $x_n \in \{0, 1, \ldots, B-1\}$ für alle $n \in \mathbb{N}$ (B-adische Entwicklung von $x$). Diese ist eindeutig, wenn gilt:

- $x_{-1} \neq 0$, falls $x \neq 0$,
- es existiert ein $n \in \mathbb{N}$ mit $x_{-n} \neq B - 1$.

Eine Maschinenzahl dagegen hat die Form

$$\tilde{x} = \operatorname{sgn}(x) B^N \sum_{n=1}^{l} x_{-n} B^{-n},$$

wobei die feste Größe $l \in \mathbb{N}$ die Mantissenlänge ist. Als Mantisse bezeichnet man den Ausdruck

$$m(x) = \sum_{n=1}^{l} x_{-n} B^{-n}. \tag{A.2}$$

Hieraus folgt sofort, dass irrationale Zahlen überhaupt nicht und von den rationalen Zahlen nur ein Teil im Rechner dargestellt werden. Man unterscheidet zwei Arten der Darstellung:

1. *Festkommadarstellung.* Sowohl die Anzahl $N$ der zur Darstellung verfügbaren Ziffern, als auch die Anzahl $N_1$ der Vorkommastellen ist fixiert. Die Anzahl der maximal möglichen Nachkommastellen $N_2$ erfüllt notwendigerweise

$$N = N_1 + N_2.$$

2. *Gleitkommadarstellung.* In (A.2) ist die Mantissenlänge l fixiert und der Exponent N ist begrenzt durch

$$N_- \leq N \leq N_+, \qquad N_-, N_+ \in \mathbb{Z}.$$

Alle Maschinenzahlen liegen dabei vom Betrag her im Intervall $(B^{N_- -1}, B^{N_+})$. Zur Normalisierung der Darstellung verlangt man, dass $x_{-1} \neq 0$, wenn $x \neq 0$. Zahlen, die kleiner als $B^{N_- -1}$ sind, werden vom Computer als 0 angesehen. Zahlen, die größer als $B^{N_+}$ sind, können nicht verarbeitet werden (Exponentenüberlauf).

Die Darstellung von irrationalen Zahlen erfordert eine Projektion auf die Menge der Maschinenzahlen, die so genannte Rundung. Man unterscheidet vier Rundungsarten:

1. *Aufrundung.* Zu $x \in \mathbb{R}$ wählt man die nächsthöhere Maschinenzahl $\tilde{x}$.

2. *Abrundung.* Zu $x \in \mathbb{R}$ wählt man die nächstniedrigere Maschinenzahl $\tilde{x}$.

3. *Rundung (im engeren Sinne).* Ausgehend von der Darstellung (A.1), setzt man

$$\tilde{x} := \operatorname{sgn}(x) B^N \begin{cases} \sum_{n=1}^{l} x_{-n} B^{-n}, & \text{falls } x_{-(l+1)} < B/2, \\ \sum_{n=1}^{l} x_{-n} B^{-n} + B^{-l}, & \text{falls } x_{-(l+1)} \geq B/2. \end{cases}$$

4. *Abschneiden.* Verwerfen aller $x_{-n}$ mit $n > l$ in der Darstellung A.1.

Die Größe

$$|\tilde{x} - x|$$

heißt absoluter, die Größe

$$\frac{|\tilde{x} - x|}{|x|}$$

heißt relativer Rundungsfehler.

Man wird erwarten, dass die Rundung im engeren Sinne die beste ist. In der Tat weist sie statistisch gesehen die geringsten Rundungsfehler auf. Für die Rundung (im engeren Sinne) gilt:

a) $\tilde{x}$ ist wieder eine Maschinenzahl.

b) Der absolute Fehler erfüllt

$$|\tilde{x} - x| \leq \frac{1}{2} B^{N-l}.$$

c) Der relative Fehler erfüllt

$$\frac{|\tilde{x} - x|}{|x|} \leq \frac{1}{2} B^{-l+1} =: eps.$$

Die Zahl *eps* wird auch als Maschinengenauigkeit bezeichnet.

## A.2 IEEE Gleitkommazahlen

Die IEEE (Institute of Electrical and Electronic Engineers) ist eine internationale Organisation, die binäre Formate für Gleitkommazahlen festlegt. Diese Formate werden von den meisten (aber nicht allen) heutigen Computern benutzt. Die IEEE definiert zwei unterschiedliche Formate mit unterschiedlicher Präzision: Einzel- und Doppelpräzision (single and double precision). Single precision wird in C von float Variablen und double precision von double Variablen benutzt.

Intel's mathematischer Co-Prozessor benutzt eine dritte, höhere Präzision, die sogenannte erweiterte Präzision (extended precision). Alle Daten im Co-Prozessor werden mit dieser erweiterten Präzision behandelt. Werden die Daten aus dem Co-Prozessor im Speicher ausgelagert, so werden sie automatisch umgewandelt in single bzw. double precision. Die erweiterte Präzision benutzt ein etwas anderes Format als die IEEE float- bzw double-Formate und werden in diesem Abschnitt nicht diskutiert.

**IEEE single precision.**

Single precision Gleitkommazahlen benutzen 32 Bits (4 Byte) um eine Zahl zu verschlüsseln:
**Bit 1-23**: Hier wird die Mantisse gespeichert, d.h. die Mantissenlänge $l$ beträgt bei single precision 23. Im single precision Format wird zur Mantisse noch Eins addiert.

**Beispiel A.1 IEEE Mantisse.**

$$\begin{aligned}&\overbrace{10110110100000000000000}^{\text{23 Bit}}\\ =&0.1011011101_2 + 1\\ =&1.1011011101_2\end{aligned}$$

□

**Bit 24-31**: Hier wird der binäre Exponent $N$ gespeichert. In Wirklichkeit wird der Exponent $N$ plus 127 gespeichert.
**Bit 32**: Gibt das Vorzeichen der Zahl an.

**Beispiel A.2 IEEE single precsion.** Welche Dezimalzahl wird duch den Binärcode $\underbrace{01000001}_{\text{Byte 3}}\ \underbrace{11011011}_{\text{Byte 2}}\ \underbrace{01000000}_{\text{Byte 1}}\ \underbrace{00000000}_{\text{Byte 0}}$ dargestellt?

$$\begin{array}{cccl} \text{Bit 32} & \text{Bit 31-24} & & \text{Bit 23-1}\\ \underbrace{0}_{\text{Zahl ist positiv}} & \underbrace{10000011}_{N=131-127=4} & & \underbrace{10110110100000000000000}_{\text{Mantisse}}\\ & & = & 1.1011011101_2 * 2^N\\ & & = & 11011.01101_2\\ & & = & 2^4+2^3+2^1+2^0+2^{-2}+2^{-3}+2^{-5}\\ & & = & 27.406250 \end{array}$$

Folgendes Beispielprogramm testet ob der Computer die float Zahl 27.040625 im IEEE single precision Format speichert.

```
# include <stdio.h>
# define text1 "Der Computer testet ob die float Zahl 27.0406250 \n"
# define text2 "im IEEE single Format gespeichert wird.\n\n"
# define text3 "          \t   Byte 3   Byte 2   Byte 1   Byte 0   \n\n"
# define text4 "27.0406250 = \t 01000001 11011011 01000000 00000000"
# define text5 " (im IEEE Format)\n"
# define text6 "         = \t      65      219      64       0"
# define text7 " (Byte als unsigned char)\n\n\n"
# define text8 "Ihr Rechner betrachtet die Zahl als : %f \n"

int main()
{
       unsigned char *p;
       float a;

       printf(text1);
       printf(text2);
       printf(text3);
       printf(text4);
       printf(text5);
       printf(text6);
       printf(text7);

       /*            Byte 3        Byte 2        Byte 1        Byte 0   */
       /*                                                               */
       /* bin\"ar :  01000001      11011011      01000000      00000000 */
       /* dezimal :  65            219           64            0        */

       p=(unsigned char *) &a;
       p[0]= 0;
       p[1]= 64;
       p[2]= 219;
       p[3]= 65;

       printf(text8,a);
       return 0;
}
```

□

### IEEE double precision

Double precision Gleitkommazahlen benutzen 64 Bits (8 Byte) um eine Zahl zu verschlüsseln:
**Bit 1-52**: Mantisse
**Bit 53-63**: Exponent
**Bit 64**: Vorzeichen
Mehr zu diesem Thema findet man zum Beispiel in [Car05].

## A.3 Computerarithmetik

Das Rechnen mit ganzen Zahlen ist (innerhalb der erwähnten Grenzen) kein Problem: Addition und Multiplikation sind kommutativ, assoziativ und das Distributivgesetz gilt.

**Beim Rechnen in der Fest- oder Gleitkommadarstellung gelten Assoziativ- und Distributivgesetz jedoch nicht!**

Ein weiterer Effekt, der zu beachten ist, ist die Auslöschung von führenden Stellen.

**Beispiel A.3 Auslöschnung.** Sei $B = 10$, $l = 3$. Sei $x = 0.9995$, $y = 0.9984$ und es soll im Computer die Differenz $x - y$ berechnet werden. Wie wirkt sich die Rundung auf das Resultat aus? Zunächst ist $\tilde{x} = 0.1 \cdot 10^1$ und $\tilde{y} = 0.998$. Daher ist

$$\tilde{x} - \tilde{y} = 0.2 \cdot 10^{-2},$$

das exakte Resultat ist

$$x - y = 0.11 \cdot 10^{-2}$$

und der relative Fehler ergibt sich zu 81.8% ! Dieser Effekt der Verstärkung des Rundungsfehlers tritt auf, wenn man fast identische Zahlen voneinander subtrahiert. □

Eine Operation mit ähnlich instabilen Verhalten ist die Division $x/y$ mit $x \ll y$. Addition zweier Zahlen mit gleichem Vorzeichen und Multiplikation sind jedoch gutartige Operationen.

# Literaturverzeichnis

[Bar04] Daniel J. Barrett. *Linux kurz und gut.* O'Reilly, 2004.

[Bur07] Martin Burger. Mathematische Modellierung. Skript zur Vorlesung, Universität Münster, Institut für Numerische und Angewandte Mathematik, 2006/07.

[Car05] P.A. Carter. *PC Assembly Language.* online unter http://www.drpaulcarter.com/pcasm/, 2005.

[CS74] C.C.Lin and L.A. Segel. *Mathematics Applied to Deterministic Problems in the Natural Sciences.* MacMillan New York, 1974.

[DS04] Timothy A. Davis and Kermit Sigmon. *MATLAB Primer.* Chapman & Hall/CRC, 7th edition, 2004.

[Seg72] L.A. Segel. Simplification and scaling. *SIAM Review*, 14:547 – 571, 1972.

[Son01] Thomas Sonar. *Angewandte Mathematik, Modellbildung und Informatik.* Vieweg, 2001.

[Wla72] W.S. Wladimirow. *Gleichungen der mathematischen Physik*, volume 74 of *Hochschulbücher für Mathematik.* VEB Deutscher Verlag der Wissenschaften Berlin, 1972.

# Index


', 14
.*, 15
./, 15
:, 14
<=, 16
<, 16
==, 16
>=, 16
>, 16
&&, 16
~=, 16
~, 16
\, 15
atan, 15
break, 17
case, 16
cd, 12
clear, 12
clf, 12
cos, 15
det, 15
disp, 13
edit, 12
eig, 15
elseif, 16
else, 16
exp, 15
eye, 14
format, 13
for, 17
help, 12
hold, 17
if, 16
inv, 15
i, 14
j, 14
log10, 16
log, 15
ls, 12
mesh, 17
norm, 15
otherwise, 17
pi, 14
plot, 17
pwd, 12
rand, 14
rank, 15
sin, 16
size, 14
sqrt, 16
switch, 16
tan, 16
what, 12
while, 17
who, 12
xor, 16
zeros, 14

Abrundung, 103
Abschneiden, 103
Adressoperator, 77, 78
Allokieren, 80
Alternative, 16
Anfangswertproblem, 28
Anweisung, 10
Arbeit, 38
array, 50
ASCII-Tabelle, 74
Aufrundung, 103
Aufzählungstyp, 94
Auslöschnung, 106

B-adische Entwicklung, 102
bdf, 8
Bezeichner, 91
Bibliotheksfunktionen, 83

C–Anweisung
    break, 74
    case, 68
    continue, 74
    do–while(), 73
    for(), 70
    goto, 74
    if(), 67
    switch(), 68
    while(), 73
C-Anweisung

calloc(), 80
free(), 80
malloc(), 80
printf(), 46, 63
realloc(), 80
scanf(), 64
sizeof(), 50
call by reference, 88
call by value, 86
cast, 62
cd, 8
chmod, 8
cp, 8

Datentyp
char, 49
signed, 49
unsigned, 49
const, 49
double, 49
long, 49
enum, 94, 100
float, 49
int, 49
short, 49
signed, 49
unsigned, 49
pointer, 76
struct, 94
strukturierte, 94
typedef, 100
union, 94, 99
void, 49
zeiger, 76
Datentypen, 48
definiert, 48
Deklaration, 48
deklarieren, 48
Dereferenzoperator, 78
Differentialgleichung
gewöhnliche, 28
Differenzengleichung, 30
logistische, 36
Differenzenquotient, 30
Diffusion, 40

Energie
kinetische, 38
Enthalpie, 38
Entropie, 39
enum, 100
env, 8
eps, 103
Euler–Verfahren
explizites, 33
implizit, 34
Exponentenüberlauf, 103

Felder, 50
dynamisch, 80
Festkommadarstellung, 102
find, 8
Fourier'sches Abkühlungsgesetz, 40
Funktion
affine, 23
call by reference, 88
call by value, 86
Funktionsparameter, 83
Funktionsrumpf, 83
Rückgabewert, 83
Funktionen, 83
Funktionsparameter, 83
Funktionsrumpf, 83

Gleitkommadarstellung, 103
grep, 8
gunzip, 8
gzip, 8

Header
float.h, 62
limits.h, 62
malloc.h, 80
math.h, 59
stdio.h, 46
string.h, 60
Headerdatei, 46

IEEE, 104
Initialisierung, 48

kill, 8
kinetische Energie, 38
Kommandozeilen–Parameter, 89
Kommentare, 46

Laplace-Gleichung, 42
libraries, 45
Listen, 98
ll, 8
ls, 8

M–File, 12
man, 8
Mantisse, 102
Mantissenlänge, 102
Maschinengenauigkeit, 103
Maschinenzahl, 102
Materialgesetze, 40

mkdir, 8
Modell
  diskret, 20
  Kontinuums–, 20
  qualitativ, 19
  quantitativ, 19
Modellfehler, 42
Monitor, 47
more , 8
mv, 8

Null–Pointer, 80

Operationen, 48
Operator
  Adressoperator, 77
  Assoziativität, 57
  Auswahloperator, 96
  bitorientiert, 55
  dekrement, 56
  Dereferenzoperator, 78
  inkrement, 56
  logisch, 54
  Priorität, 57
  Referenzoperator, 78
  Strukturoperator, 95
  Vergleichsoperator, 54
  Zugriffsoperator, 78
Output, 22

Parameter, 22
  effektiver, 25
Pointer, 76
Pointervariable, 77
Poisson-Gleichung, 42
Populationsdynamik, 28
Postfixnotation, 57
Präfixnotation, 56
Präprozessor, 46
Präprozessordirektiven, 46
Programmflusskontrolle, 67
ps, 8
pwd, 8

Rückgabewert, 83
Randbedingung
  Dirichlet, 41
  Neumann, 41
  Robin–, 41
Referenzoperator, 78
rekursiv, 88
rm, 8
rmdir, 8
Rundung, 103
Rundungsfehler, 103
  absolut, 103
  relativ, 103

Schlüsselwörter, 46
Schleife, 10
Schleifen, 69
  do–while, 73
  for, 70
  while, 73
Sensitivität, 26
Shell, 6
Signum-Funktion, 67
Skalen, 19
String, 50
Strings, 46
Struktur, 94
Suffix, 46

tail, 8
tar, 8
Temperatur, 38
Thermodynamik
  erster Hauptsatz, 38
  zweite Hauptsatz, 39
Transport, 39
Transportgleichung, 40
Trennung der Variablen, 28
typedef, 100
typeset, 8
Typkonversion, 62

underscore, 48
Union, 94
union, 99

Variable
  dimensionslos, 23
Variablen, 22, 48
  Gültigkeit, 85
  global, 84
  lokal, 84
  Sichtbarkeit, 85
Verzweigung, 10, 67

w, 8
Wärme, 38
Wärmeenergie, 38
Wärmekapazität
  spezifische, 40
Wärmeleitkoeffizient, 40
Wärmeleitungsgleichung, 40
Wachstum, 28
which, 8
who, 8

Wiederholung, 10

yppasswd, 8

Zeichenketten, 46
Zeichenkettten, 51
Zeiger, 76
Zeiger auf Funktionen, 91
Zeigervariable, 77
Zugriffsoperator, 78
Zyklus
    abweisend, 11, 73
    nichtabweisend, 11, 73
    Zählzyklus, 70