

# Modellierung und Programmierung

Volker John

Wintersemester 2006/07

# Inhaltsverzeichnis

<b>I</b>	<b>3</b>
<b>1 Modellierung</b>	<b>4</b>
1.1 Der Wurf . . . . .	4
1.2 Die eindimensionale Wärmeleitungsgleichung . . . . .	5
1.3 Gewöhnliche Differentialgleichungen . . . . .	7
1.4 Fazit . . . . .	11
<b>2 Das Betriebssystem Linux</b>	<b>12</b>
<b>3 Algorithmen</b>	<b>16</b>
<b>4 Einführung in MATLAB</b>	<b>19</b>
4.1 Allgemeines . . . . .	19
4.2 Bemerkungen zu Vektoren und Matrizen . . . . .	20
4.3 Matrizen . . . . .	21
4.4 Wichtige Funktionen in MATLAB . . . . .	22
4.5 Ablaufkontrolle . . . . .	23
4.6 Graphik . . . . .	24
<b>II Die Programmiersprache C</b>	<b>25</b>
<b>5 Einführung</b>	<b>26</b>
5.1 Das erste C-Programm . . . . .	26
5.2 Interne Details beim Compilieren (*) . . . . .	28
<b>6 Variablen, Datentypen und Operationen</b>	<b>29</b>
6.1 Deklaration, Initialisierung, Definition . . . . .	29
6.2 Elementare Datentypen . . . . .	29
6.3 Felder und Strings . . . . .	31
6.3.1 Felder . . . . .	31
6.3.2 Mehrdimensionale Felder . . . . .	32
6.3.3 Zeichenketten (Strings) . . . . .	32
6.4 Ausdrücke, Operatoren und mathematische Funktionen . . . . .	33
6.4.1 Arithmetische Operatoren . . . . .	34
6.4.2 Vergleichsoperatoren . . . . .	35
6.4.3 Logische Operatoren . . . . .	35
6.4.4 Bitorientierte Operatoren (*) . . . . .	36
6.4.5 Inkrement- und Dekrementoperatoren . . . . .	37
6.4.6 Adressoperator . . . . .	38
6.4.7 Prioritäten von Operatoren . . . . .	38
6.5 Operationen mit vordefinierten Funktionen . . . . .	40
6.5.1 Mathematische Funktionen . . . . .	40

6.5.2	Funktionen für Zeichenketten (Strings) . . . . .	41
6.6	Zusammengesetzte Anweisungen . . . . .	42
6.7	Nützliche Konstanten . . . . .	42
6.8	Typkonversion (cast) . . . . .	43
6.9	Standardein- und -ausgabe . . . . .	44
6.9.1	Ausgabe . . . . .	44
6.9.2	Eingabe . . . . .	45
<b>7</b>	<b>Programmflusskontrolle</b> . . . . .	<b>48</b>
7.1	Bedingte Ausführung . . . . .	48
7.1.1	Die if()-Anweisung . . . . .	48
7.1.2	Die switch()-Anweisung . . . . .	49
7.2	Schleifen . . . . .	50
7.2.1	Der Zählzyklus (for-Schleife) . . . . .	50
7.2.2	Abweisender Zyklus (while-Schleife) . . . . .	53
7.2.3	Nichtabweisender Zyklus (do-while-Schleife) . . . . .	54
7.3	Anweisungen zur unbedingten Steuerungsübergabe . . . . .	55
<b>8</b>	<b>Zeiger (Pointer)</b> . . . . .	<b>56</b>
8.1	Adressen . . . . .	56
8.2	Pointervariablen . . . . .	57
8.3	Adressoperator und Zugriffoperator . . . . .	58
8.4	Zusammenhang zwischen Zeigern und Feldern . . . . .	59
8.5	Dynamische Felder mittels Zeiger . . . . .	60
<b>9</b>	<b>Funktionen</b> . . . . .	<b>63</b>
9.1	Deklaration, Definition und Rückgabewerte . . . . .	63
9.2	Lokale und globale Variablen . . . . .	64
9.3	Call by value . . . . .	66
9.4	Call by reference . . . . .	68
9.5	Rekursive Programmierung . . . . .	68
9.6	Kommandozeilen-Parameter . . . . .	69
9.7	Wie werden Deklarationen gelesen? . . . . .	71
9.8	Zeiger auf Funktionen . . . . .	71
<b>10</b>	<b>Strukturierte Datentypen</b> . . . . .	<b>74</b>
10.1	Strukturen . . . . .	74
10.1.1	Deklaration von Strukturen . . . . .	74
10.1.2	Definition von Strukturvariablen . . . . .	75
10.1.3	Felder von Strukturen . . . . .	75
10.1.4	Zugriff auf Strukturen . . . . .	75
10.1.5	Zugriff auf Strukturen mit Zeigern . . . . .	76
10.1.6	Geschachtelte Strukturen . . . . .	77
10.1.7	Listen . . . . .	78
10.2	Unions . . . . .	79
10.3	Aufzählungstyp . . . . .	80
10.4	Allgemeine Typendefinition . . . . .	80
<b>A</b>	<b>Computerarithmetik (*)</b> . . . . .	<b>82</b>
A.1	Zahlendarstellung im Rechner und Computerarithmetik . . . . .	82
A.2	IEEE Gleitkommazahlen . . . . .	84
A.3	Computerarithmetik . . . . .	85

# Teil I

# Kapitel 1

## Modellierung

Durch Modellierung beschreibt man Vorgänge aus der Natur sowie industrielle Prozesse mit mathematischen Werkzeugen, zum Beispiel Gleichungen oder Ungleichungen. Modellierung geschieht durch Abstraktion, das heißt durch Vereinfachung und Verallgemeinerung der Realität.

### 1.1 Der Wurf

Ein in der Schule bereits detailliert untersuchter physikalischer Vorgang ist der Wurf.

Wir betrachten eine Punktmasse mit der konstanten Masse  $m$  [kg] und die Ebene, in der die Wurfkurve liegt. Das Ziel besteht darin, eine Formel für die Wurfkurve zu finden.

Der Wurf gehorcht dem Newtonschen<sup>1</sup> Grundgesetz der Dynamik

$$\mathbf{F} = \frac{d(m\mathbf{v})}{dt} =: (\dot{m}\mathbf{v}), \quad (1.1)$$

wobei  $\mathbf{F}$  [N] die wirkenden Kräfte sind,  $\mathbf{v}$  [m/s] die Geschwindigkeit und  $m\mathbf{v}$  [Ns] der Impuls. Der Punkt bezeichnet die Ableitung nach der Zeit. Damit besagt (1.1), dass die zeitliche Änderung des Impulsvektors der einwirkenden Kraft proportional ist und in die Richtung geschieht, in der jene Kraft angreift.

Bezeichne  $\mathbf{x} = (x, y)^T$  die Koordinaten der Ebene der Wurfkurve und  $\mathbf{v}(t) = (v_x(t), v_y(t))^T$  den Geschwindigkeitsvektor. Wir nehmen an, dass nur die Schwerkraft wirkt, also

$$\mathbf{F} = \begin{pmatrix} 0 \\ -mg \end{pmatrix},$$

wobei  $g \approx 9.81$  [m/s<sup>2</sup>] die Fallbeschleunigung ist. Da die Masse konstant ist, vereinfacht sich das Grundgesetz der Dynamik zu

$$\dot{\mathbf{v}} = \frac{\mathbf{F}}{m}, \quad (1.2)$$

wobei die zeitliche Ableitung der Geschwindigkeit die Beschleunigung ist. Die Geschwindigkeit erhält man nun durch Integration von (1.2)

$$\mathbf{v} = \frac{\mathbf{F}}{m}t + \mathbf{v}_0 \quad \text{oder} \quad \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} 0 \\ -g \end{pmatrix}t + \begin{pmatrix} v_{0x} \\ v_{0y} \end{pmatrix}. \quad (1.3)$$

Die Integrationskonstante  $\mathbf{v}_0$  muss durch eine Zusatzbedingung festgelegt, etwa durch die Geschwindigkeit zum Anfangszeitpunkt  $t = 0$  (Anfangsbedingung).

---

<sup>1</sup>Isaac Newton (1642 – 1727)

Nun ist die Geschwindigkeit die Ableitung des Weges nach der Zeit. Man erhält damit die Wurfkurve durch Integration von (1.3) nach der Zeit

$$\mathbf{x} = \frac{\mathbf{F}}{2m}t^2 + \mathbf{v}_0t + \mathbf{x}_0 \quad \text{oder} \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ -g/2 \end{pmatrix} t^2 + \begin{pmatrix} v_{0x} \\ v_{0y} \end{pmatrix} t + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}. \quad (1.4)$$

Auch hier muss die Integrationskonstante  $\mathbf{x}_0$  durch eine Zusatzbedingung festgelegt werden, etwa durch die Position des Wurfobjekts zum Anfangszeitpunkt.

Mit (1.4) hat man eine Darstellung der Wurfkurve erhalten, eine sogenannte Parameterdarstellung mit dem Parameter  $t$ . Löst man die obere Gleichung nach  $t$  (falls  $v_{0x} \neq 0$  ist) auf, setzt das Ergebnis in die zweite Gleichung ein und vereinfacht, erhält man

$$y = -\frac{g}{2v_{0x}^2}x^2 + \left(\frac{gx_0}{v_{0x}^2} + \frac{v_{0y}}{v_{0x}}\right)x + \left(y_0 - \frac{gx_0^2}{2v_{0x}^2} - \frac{v_{0y}x_0}{v_{0x}}\right). \quad (1.5)$$

Man erkennt, dass die Wurfkurve eine Parabel ist. *Übungsaufgaben: spezielle Würfe (AB erkennen und einsetzen), Merkmale dieser Würfe*

Damit haben wir das Ziel einer mathematischen Beschreibung der Wurfkurve erreicht. Ausgehend von einem physikalischen Grundgesetz, durch Nutzung von Definitionen von physikalischen Größen sowie zweimalige Integration, haben wir eine Formel gefunden. Die Frage ist, wie gut diese Formel ist. Das hängt von den Modellfehlern ab, die hier unter anderem durch folgende Dinge verursacht sind:

- Der richtige Wurfkörper ist dreidimensional und keine Punktmasse.
- Der richtige Körper 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<sup>2</sup>'sches oder Newton'sches Reibungsgesetz).
- Der reale Wurf findet nicht in einer Ebene statt.
- Die Fallbeschleunigung ist nur näherungsweise bekannt.

Unter diesen Voraussetzungen haben wir das explizite Ergebnis (1.5) erhalten.

## 1.2 Die eindimensionale Wärmeleitungsgleichung

Wir betrachten die Wärmeausbreitung in einem Stab, der als eindimensionale Strecke  $\Omega = (a, b)$  modelliert wird. Seien  $x \in (a, b)$  die Ortskoordinate,  $t \in \mathbb{R}$ ,  $t \geq 0$ , die Zeit,

- $u(t, x)$  – Temperatur zum Zeitpunkt  $t$  im Ort  $x$ ,  $[K]$ ,
- $\rho(x)$  – Dichte des Mediums aus dem der Stab besteht,  $[kg/m]$ ,
- $c(x)$  – spezifische Wärmekapazität des betrachteten Mediums,  $[J/(kg K)] = [W s/(kg K)]$ ,
- $k(x)$  – Wärmeleitkoeffizient des betrachteten Mediums,  $[W/(m K)]$ ,
- $F(t, x)$  – Intensität von Wärmequellen,  $[W/m]$ .

Wir betrachten das Wärmegleichgewicht in einem beliebigen Volumen  $V = (\alpha, \beta) \subset (a, b)$  im Zeitintervall  $(t, t + \Delta t)$ . Auf Grund des Fourierschen Gesetzes tritt durch den Rand von  $V$  die Wärmemenge,  $[J]$ ,

$$Q_1 = \int_t^{t+\Delta t} k(t, \beta)u'(t, \beta) - k(t, \alpha)u'(t, \alpha) dt$$

---

<sup>2</sup>George Gabriel Stokes (1819 – 1903)

in  $V$  ein, wobei der Strich die Ableitung nach  $x$  bezeichnet. Durch partielle Integration erhält man

$$Q_1 = \int_t^{t+\Delta t} \int_\alpha^\beta (k(t, x)u'(t, x))' dx dt.$$

In  $V$  entsteht die Wärmemenge

$$Q_2 = \int_t^{t+\Delta t} \int_\alpha^\beta F(t, x) dx dt.$$

Die Temperatur in einem Punkt  $x$  wächst in  $(t, t + \Delta t)$  in erster Näherung um den Wert

$$u(t + \Delta t, x) - u(t, x) \approx \dot{u}(t, x)\Delta t,$$

wobei der Punkt die Ableitung der Temperatur nach der Zeit bedeutet. Das ist der Wert, den die Tangente im Punkt  $(t, x)$ , im Punkt  $(t + \Delta t, x)$  annimmt. Man nimmt diesen linearen Ansatz, setzt also

$$u(t + \Delta t, x) - u(t, x) = \dot{u}(t, x)\Delta t. \quad (1.6)$$

Für das Anwachsen der Temperatur in  $V$  und für beliebiges  $\Delta t$  benötigt man damit die Wärmemenge

$$\begin{aligned} Q_3 &= \int_t^{t+\Delta t} \int_\alpha^\beta c(x)\rho(x) \frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} dx dt \\ &= \int_t^{t+\Delta t} \int_\alpha^\beta c(x)\rho(x)\dot{u}(t, x) dx dt. \end{aligned}$$

Es gilt nun  $Q_3 = Q_2 + Q_1$  und somit

$$\int_t^{t+\Delta t} \int_\alpha^\beta [c\rho\dot{u} - (ku')' - F](t, x) dx dt = 0.$$

Da das Volumen  $V$  beliebig gewählt wurde und  $\Delta t$  ebenfalls beliebig ist, muss der Integrand gleich Null sein (wird später in der Analysis-Vorlesung bewiesen). Man erhält die **Wärmeleitungsgleichung**

$$c\rho\dot{u} - (ku')' = F \text{ in } (0, T) \times (a, b).$$

Für ein homogenes Medium sind  $c, \rho, k$  konstant. Die Gleichung vereinfacht sich zu

$$\dot{u} - \varepsilon u'' = f \text{ in } (0, T) \times (a, b). \quad (1.7)$$

mit  $\varepsilon = k/(c\rho) > 0$  und  $f = F/(c\rho)$ . Das ist eine sogenannte Differentialgleichung, da Ableitungen der gesuchten Funktion auftreten.

Gesucht ist eine Funktion  $u(t, x)$ , die (1.7) erfüllt. Diese Funktion muss aber auch noch zusätzliche Bedingungen erfüllen. Die Beschreibung der Temperaturentstehung macht nur Sinn, wenn man weiß, wie die Temperaturverteilung  $u(0, x)$  zum Anfangszeitpunkt aussieht (Anfangsbedingung) und wenn man weiß, was im Laufe der Zeit an den Stabenden passiert (Randbedingungen). Beispiele für Randbedingungen:

- a) **Dirichlet<sup>3</sup> –Randbedingung.** Die Temperatur  $u$  am Rand wird vorgegeben, zum Beispiel

$$u(t, a) = g_1(t) \text{ in } (0, T).$$

---

<sup>3</sup>Johann Peter Gustav Lejeune Dirichlet (1805 – 1859)

- b) **Neumann<sup>4</sup>-Randbedingung.** Ein gegebener Wärmefluss wird am Rand aufrecht erhalten, zum Beispiel

$$k(t, a)u'(t, a) = g_2(t) \text{ in } (0, T).$$

Falls  $u$  zeitlich konstant ist, erhält man die **stationäre Wärmeleitungsgleichung**

$$-\varepsilon u'' = f \text{ in } (a, b). \quad (1.8)$$

Das ist die sogenannte **Poisson<sup>5</sup>-Gleichung**. Die homogene Form, d.h.  $f = 0$ , wird **Laplace<sup>6</sup>-Gleichung** genannt. Die Gleichung (1.8) kann man im Prinzip durch zweimaliges Integrieren im Ort lösen, währenddessen das bei der Gleichung (1.7) nicht mehr funktioniert.

Die Differentialgleichungen (1.7) und (1.8) modellieren die Wärmeausbreitung in einem Stab. Es stellt sich auch hier die Frage, wie gut diese Modelle sind. Der Modellfehler besitzt u.a. 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.
- Beim linearen Ansatz (1.6) werden Terme höherer Ordnung vernachlässigt.
- Das Material ist im allgemeinen nicht vollständig homogen.
- Die physikalischen Konstanten kennt man nur bis zu einer gewissen Genauigkeit.
- Die Anfangsbedingung kennt man i.a. nur punktweise.
- Die Randbedingungen kann man auch nur zu einer gewissen Genauigkeit steuern.

### 1.3 Gewöhnliche Differentialgleichungen

Wir haben gesehen, dass die Modellierung von physikalischen Prozessen 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  $u : (a, b) \rightarrow \mathbb{R}$ , so spricht man von gewöhnlichen Differentialgleichungen. Diese werden im Laufe des Studiums noch ausführlich behandelt. Hier dienen sie nur als Motivation für den Programmierungsteil der Vorlesung.

Die Herleitung von gewöhnlichen Differentialgleichungen durch Modellierung ist ein Teil der Beschreibung von Naturvorgängen. Der zweite Teil besteht darin, diese Gleichungen zu lösen.

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

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

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

$$u(x) = \int f(x) dx. \quad (1.9)$$

Bekanntes aus der Schule über das Integral:

<sup>4</sup>Carl Gottfried Neumann (1832 – 1925)

<sup>5</sup>Siméon Denis Poisson (1781 – 1840)

<sup>6</sup>Pierre Simon Laplace (1749 – 1829)

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

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

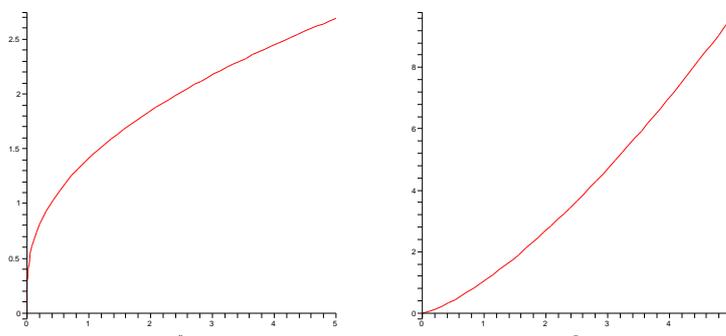


Abbildung 1.1: Integrand und Stammfunktion zum Beispiel 1.1.

□

**Beispiel 1.2** Gesucht ist die Stammfunktion von  $f(x) = \sqrt{x^2 + \sqrt{x}}$ . Man erhält mit MAPLE

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

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

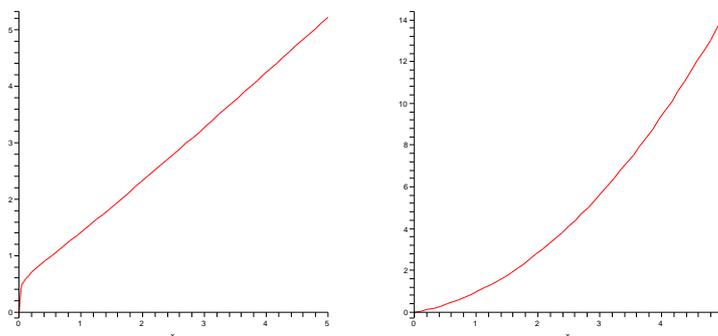


Abbildung 1.2: Integrand und Stammfunktion zum Beispiel 1.2.

□

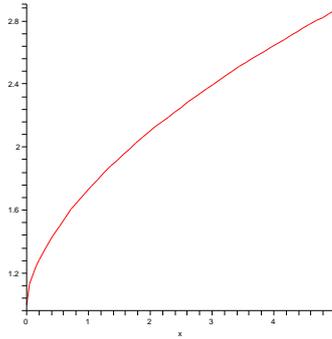


Abbildung 1.3: Integrand zum Beispiel 1.3.

**Beispiel 1.3** 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.  $\square$

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. Ein einfacher Typ ist die homogene, gewöhnliche Differentialgleichung erster Ordnung mit konstanten Koeffizienten. Gegeben ist  $\alpha \in \mathbb{R}$ . Gesucht ist eine Funktion  $u : (a, b) \rightarrow \mathbb{R}$  mit

$$u'(x) + \alpha u(x) = 0 \quad \forall x \in (a, b). \quad (1.10)$$

Für die Lösung von (1.10) gibt es ein einfaches Verfahren. Der Einfachheit halber wird  $u(x) > 0$  vorausgesetzt. Durch Umstellung erhält man

$$-\alpha = \frac{u'(x)}{u(x)} = (\ln(u(x)))'.$$

Integration ergibt

$$\ln(u(x)) = -\alpha x + c_0 \quad \implies \quad u(x) = c_1 e^{-\alpha x}.$$

Die Konstante  $c_1$  bestimmt sich durch eine Anfangsbedingung, die man zusätzlich zu (1.10) benötigt.

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

$$u'(x) = x^2 + u^2(x). \quad (1.11)$$

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

$$u'(x) = f(x, u(x)) \quad x \in (a, b), \quad u(a) = u_0. \quad (1.12)$$

Das einfachste numerische Verfahren zur Approximation der Lösung von (1.12) ist das explizite Euler<sup>7</sup>-Verfahren. Zunächst zerlegt man  $[a, b]$  in  $n$  (gleich große) Teilintervalle mit den Punkten

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

<sup>7</sup>Leonhard Euler (1707 – 1783)

siehe Abbildung 1.4. Die numerische Approximation der Lösung wird mit  $u^h$  bezeichnet.



Abbildung 1.4: 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 1.5. Dabei macht man im allgemeinen einen Fehler :  $u^h(x_2) \neq u(x_2)$  !

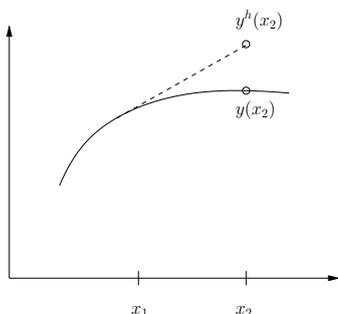


Abbildung 1.5: Prinzip des expliziten Euler-Verfahrens.

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

$$\begin{aligned} u^h(x_1) &= u(x_1), \\ u^h(x_i) &= u^h(x_{i-1}) + hf(x_{i-1}, u^h(x_{i-1})), \quad i = 2, \dots, n+1. \end{aligned} \quad (1.13)$$

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

$$\begin{aligned} u^h(x_1) &= u(x_1), \\ u^h(x_i) &= u^h(x_{i-1}) + hf(x_i, u^h(x_i)), \quad i = 2, \dots, n+1. \end{aligned} \quad (1.14)$$

Bei diesem Verfahren muss man zur Berechnung von  $u^h(x_i)$  i.a. 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 (genauer auf dem gleichen Gitter bei vergleichbarem Aufwand) ?

**Beispiel 1.4** Wir betrachten eine Gleichung vom Typ (1.11)

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

Der Iterationsschritt beim expliziten Euler-Verfahren lautet

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

und beim impliziten Euler-Verfahren

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

In jedem Schritt des impliziten Euler-Verfahrens muss man eine quadratische Gleichung lösen

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

*MATLAB-DEMO*

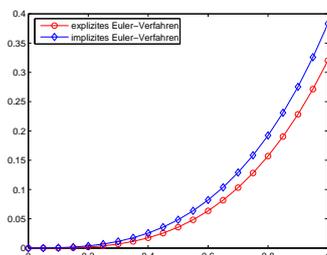


Abbildung 1.6: Approximation der Lösung von Beispiel 1.4 in  $[0, 1]$ ,  $h = 0.05$ .

□

## 1.4 Fazit

Die Modellierung von Vorgängen aus der Natur führt im allgemeinen auf Gleichungen, bei denen man Funktionen zu berechnen hat. Diese Gleichungen enthalten Ableitungen oder auch Integrale der gesuchten Funktionen und sind oft nichtlinear. Im allgemeinen wird man keine analytische Lösung finden können. Zur Approximation der Lösung kann man nur numerische Verfahren verwenden. Dafür muss man geeignete Algorithmen entwickeln (siehe zum Beispiel Vorlesung Praktische Mathematik) und diese Algorithmen auf Computern implementieren. Die Implementierung erfolgt unter Verwendung von Programmiersprachen, mit denen geeignete Programme geschrieben werden. Die Einführung in die Programmierung ist der Hauptbestandteil dieser Vorlesung.

## Kapitel 2

# 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.

Der Server vom Computer-Pool ist `pyramis.math.uni-sb.de`.

### Zugriffsrechte

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

```
-rw-r----- 1 john users 29444 2006-10-13 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 2006-10-12 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 2006-10-13 12:22 tmp.txt
```

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 Benutzeroberflächen (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. 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, emacs, Xemacs (sind selbsterklärend), vi ist ein Kommandozeile-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,
- `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 2.1. 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
```

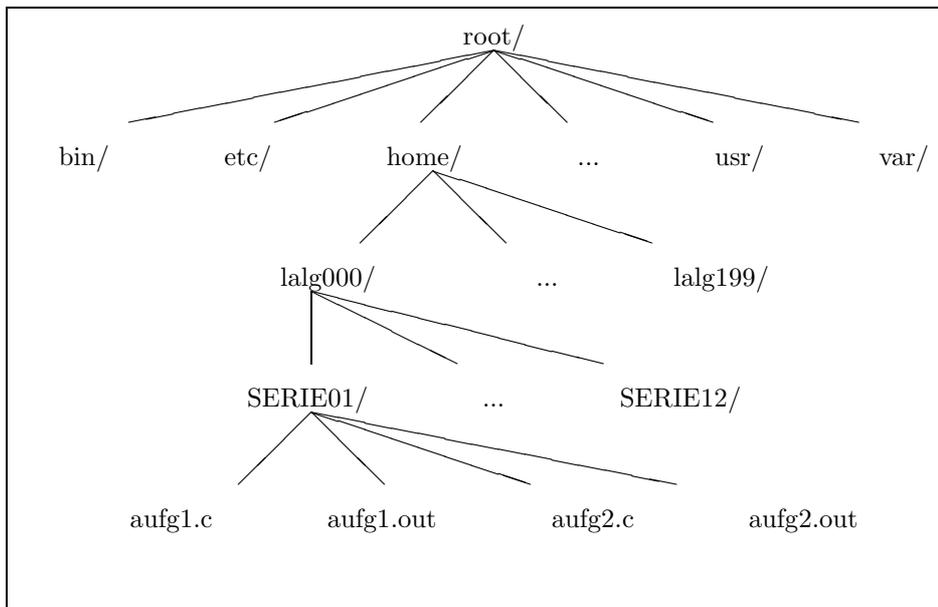


Abbildung 2.1: Beispiel für einen Verzeichnisbaum in Linux.

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

Von einem Hörer ist das Buch [Bar04] zur Einarbeitung in Linux empfohlen wurden.

# Kapitel 3

## 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 3.1** *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 3.2 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<sup>1</sup>sche Wurzelformel nutzt:

*Algorithmus 3.3 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

$$x_1 := \sqrt{-q}, \quad x_2 := -x_1.$$

---

<sup>1</sup>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.  $\square$

**Beispiel 3.4** *Summe von Zahlen.* Man solle die Summe der natürlichen Zahlen von 1000 bis 10000 bilden. Ein möglicher Algorithmus ist, diese nacheinander aufzuaddieren:

*Algorithmus 3.5*

- 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<sup>2</sup>

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

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 Ziel führen. Ziel ist es natürlich, den jeweils effizientesten Algorithmus zu verwenden.  $\square$

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 3.6** Am Anfang von Algorithmus 3.2 kommt die folgende Sequenz

$$\begin{aligned} a &= -p/2; \\ D &= a*a-q; \end{aligned}$$

(Hier wird MATLAB-Notation genutzt.)  $\square$

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

**Beispiel 3.7** Die einfache Alternative haben wir bereits in den Algorithmen 3.2 und 3.3 gesehen:

<sup>2</sup>Johann Carl Friedrich Gauss (1777 – 1855)

```

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

```

Im Algorithmus 3.3 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.  $\square$

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 3.8** Die Steuerung durch einen Zähler hatten wir bereits in Algorithmus 3.5:

```

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

```

$\square$

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 4

# Einführung in MATLAB

### 4.1 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 ist eine Classroom-Lizenz (25 Stück) von MATLAB 7.3 (R2006b) vorhanden. Im Moment kann das Programm mit dem Skript

```
matlab.sh
```

gestartet werden, welches von der gleichen Homepage geladen werden kann, auf der die Übungsblätter zu finden sind (Aufruf `sh matlab.sh`).

Zur Einarbeitung in MATLAB gibt es viele Bücher, siehe [www.amazon.de](http://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. 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*.

## 4.2 Bemerkungen zu Vektoren und Matrizen

Vektoren sind aus der Schule bekannt. Man unterscheidet

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

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 Vektoren 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.

## 4.3 Matrizen

MATLAB rechnet mit Matrizen. Dabei ist zum Beispiel eine Zahl eine  $1 \times 1$ -Matrix und ein Spaltenvektor eine  $n \times 1$ -Matrix. 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 gegebenen Funktionen,
- Erzeugen mit eigenen Funktionen.

**Beispiel 4.1** 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:

$$\begin{aligned} A(1,1) &= 2 \\ A(1,2) &= 4 \\ A(1,3) &= 7 \\ A(2,1) &= -5 \\ A(2,2) &= 4 \\ A(2,3) &= 2 \end{aligned}$$

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

$$B = \text{eye}(n)$$

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

$$C = \text{zeros}(n)$$

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

$$D = \text{rand}(m,n)$$

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

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

$$[m,n] = \text{size}(A)$$

Hierbei ist  $m$  die Anzahl der Zeilen und  $n$  die Anzahl der Spalten.

- Die Teilmatrix mit den Zeilenindizes  $i1, \dots, i2$  und den Spaltenindizes  $j1, \dots, j2$  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

$$C = A+B \quad C = A-B \quad C = A*B$$

- Multiplikation, Division und Potenzierung einer Matrix A mit einem Skalar a ((1 × 1)-Matrix) werden mit den üblichen Symbolen bezeichnet

$$B = a*A \quad B = A/a \quad B=A^a$$

- Elementweise Multiplikation und Division werden wie folgt durchgeführt

$$C = A.*B \quad C = A./B$$

Beispiel:

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

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:

$$r = \text{rank}(A)$$

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

$$B = \text{inv}(A)$$

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

$$d = \text{det}(A)$$

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

$$d = \text{norm}(A)$$

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:

$$[u, v] = \text{eig}(A);$$

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

$$x = A \setminus b$$

## 4.4 Wichtige Funktionen in MATLAB

Befehl	Bedeutung
atan	Arcus-Tangens
cos	Kosinus
exp	Exponentialfunktion
log	Logarithmus naturalis ln !
log10	Logarithmus zur Basis 10

<code>sin</code>		Sinus
<code>sqrt</code>		Wurzel
<code>tan</code>		Tangens

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

## 4.5 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. Ein Beispiel gibt es in den Übungen.

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.

## 4.6 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

# Die Programmiersprache C

# Kapitel 5

## Einführung

Die 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 Compile-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 es in die vom Compiler erzeugte Objektdatei Funktionen (siehe auch Kapitel 9) 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.

Dieser Teil der Vorlesung basiert auf der Vorlesung mit dem gleichen Titel aus dem Wintersemester 2005/2006, gehalten von Erik Wallacher.

### 5.1 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;
}
```

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).
- Das *Schlüsselwort* `main` markiert den Beginn des Hauptprogramms, d.h. den Punkt, an dem die Ausführung der Anweisungen beginnt. Auf die Bedeutung des Schlüsselwortes `void` und 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 `hallo.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
```

## 5.2 Interne Details beim Compilieren (\*)

Der leicht geänderte Aufruf zum Compilieren

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

erzeugt eine längere BildschirmAusgabe, 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 Objektcode dateien 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
```

---

<sup>0</sup>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 6

## Variablen, Datentypen und Operationen

### 6.1 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, d.h. 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.

#### Variablennamen

Bei der Vergabe von Variablennamen ist folgendes zu beachten:

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

### 6.2 Elementare Datentypen

Die Tabelle 6.1 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 6.1: Elementare Datentypen. Die Bytelänge ist von Architektur zu Architektur unterschiedlich (hier: GNU-C-Compiler unter LINUX für die x86-Architektur. Siehe auch sizeof()).

### Beispiel 6.1 Deklaration, Initialisierung, Definition.

```

#include <stdio.h>

int main()
{
    int a=4;
    /* Deklaration von a als ganze Zahl */
    /* + Initialisierung von a, d.h. 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 6.2 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 6.4) 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, z.B.

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

*Nicht zulässig:*

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

□

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

**Beispiel 6.3 C-Anweisung : `sizeof()`.**

```
/* Beispiel: sizeof() */  
# include <stdio.h>  
  
int main()  
{  
    printf("Eine int-Variablen benötigt %i Bytes\n", sizeof(int));  
    return 0;  
}
```

□

## 6.3 Felder und Strings

### 6.3.1 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,  $i = 0, \dots, n-1$ .
- Felder können bei der Deklaration initialisiert werden. Dies geschieht unter Verwendung des Zuweisungsoperators und der geschweiften Klammer.

**Beispiel 6.4 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 bzw.  
    double-Variable ausgegeben wird */  
    return 0;  
}
```

□

### 6.3.2 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:

$$\text{Datentyp Feldname}[n_1][n_2] \dots [n_d];$$

**Beispiel 6.5 Deklaration und Initialisierung einer ganzzahligen 2 x 3-Matrix.**

```
#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;
}
```

□

### 6.3.3 Zeichenketten (Strings)

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

$$\text{char Stringname}[\text{Länge}];$$

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

$$\text{char text}[]=\{\text{'H'},\text{'a'},\text{'\0'},\text{'\0'},\text{'\0'},\text{'\0'}\};$$

initialisiert.

Ein String kann auch durch

$$\text{char text}[]=\text{"Hallo"};$$

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 6.6 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;
}
```

□

## 6.4 Ausdrücke, Operatoren und mathematische Funktionen

Der Zuweisungsoperator

$$\text{operand1} = \text{operand2}$$

weist dem linken Operanden den Wert des rechten Operanden zu.

**Beispiel 6.7 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
     * ausgegeben 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 6.8 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;
}
```

□

## 6.4.1 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, z.B. liefert 8/3 das Ergebnis 2. Wird jedoch einer der beiden Operanden in eine Gleitkommazahl umgewandelt, so erhält man das numerisch exakte Ergebnis. z.B. 8.0/3 liefert 2.66666 als Ergebnis (siehe auch Kapitel 6.8).

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

### Beispiel 6.9 Arithmetische Operatoren.

```
% ermöglicht 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 6.8 */
    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;
}
```

□

## 6.4.2 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 6.10 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;
}
```

□

## 6.4.3 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.

### 6.4.4 Bitorientierte Operatoren (\*)

Bitorientierte Operatoren sind nur auf int-Variablen (bzw. 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:

$$\begin{cases} \text{bit ungesetzt} \\ \text{bit gesetzt} \end{cases} \equiv \begin{cases} 0 \\ 1 \end{cases} \equiv \begin{cases} \text{falsch} \\ \text{wahr} \end{cases}$$

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

#### Beispiel 6.11 (Short)-Integerdarstellung im Rechner.

Darstellung im Rechner (binär)	Dezimal
$\begin{array}{r} 0 \quad 0000000 \quad 00001010 \\ + \\ \hline \end{array}$ <p style="text-align: center;">1. Byte                      2. Byte</p>	$2^3 + 2^1 = 10$
$\begin{array}{r} 1 \quad 1111111 \quad 11011011 \\ - \\ \hline \end{array}$ <p style="text-align: center;">1. Byte                      2. Byte</p>	$-(2^5 + 2^2) - 1 = -37$

□

#### 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 6.12 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;
}

```

□

### 6.4.5 Inkrement- und Dekrementoperatoren

#### Präfixnotation

Notation	Beschreibung
++ operand	operand=operand+1
-- operand	operand=operand-1

Inkrement- und Dekrementoperatoren in Präfixnotation liefern den inkrementierten bzw. dekrementierten Wert als Ergebnis zurück.

#### Beispiel 6.13 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 bzw. Dekrementieren zurück.

#### Beispiel 6.14 Postfixnotation.

```

#include <stdio.h>

int main()
{
    int i,j;

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

    /* oben angegebene Notation ist aequivalent zu */

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

    return 0;
}

```

□

### 6.4.6 Adressoperator

Der Vollständigkeit halber wird der Adressoperator “ & “ schon in diesem Kapitel eingeführt, obwohl die Bedeutung erst in Kapitel 8 klar wird.

& datenobjekt

### 6.4.7 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

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

Im Zweifelsfall kann die Priorität durch Klammerung erzwungen werden.

### Beispiel 6.15 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.  $\square$

## 6.5 Operationen mit vordefinierten Funktionen

### 6.5.1 Mathematische Funktionen

Im Headerfile `math.h` werden u.a. Deklarationen der in Tabelle 6.2 zusammengefassten mathematischen Funktionen und Konstanten bereitgestellt:

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

Tabelle 6.2: Mathematische Funktionen

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

### Beispiel 6.16 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;
}

```

□

## 6.5.2 Funktionen für Zeichenketten (Strings)

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

Funktion	Beschreibung
<code>strcat(s1,s2)</code>	Anhängen von <code>s2</code> an <code>s1</code>
<code>strcmp(s1,s2)</code>	Lexikographischer Vergleich der Strings <code>s1</code> und <code>s2</code>
<code>strcpy(s1,s2)</code>	Kopiert <code>s2</code> auf <code>s1</code>
<code>strlen(s)</code>	Anzahl der Zeichen in String <code>s</code> ( = <code>sizeof(s)-1</code> )
<code>strchr(s,c)</code>	Sucht Zeichenkonstante (Character) <code>c</code> in String <code>s</code>

Tabelle 6.3: Funktionen für Strings

### Beispiel 6.17 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, z.B. 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 Beispiel 6.17).

## 6.6 Zusammengesetzte Anweisungen

Wertzuweisungen der Form

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

können zu

```
op1 operator = op2;
```

verkürzt werden.

Hierbei ist  $operator \in \{+, -, *, /, \%, |, ^, \ll, \gg\}$ .

**Beispiel 6.18 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;
}
```

□

## 6.7 Nützliche Konstanten

Für systemabhängige Zahlenbereiche, Genauigkeiten usw. ist die Auswahl der Konstanten aus Tabelle 6.4 und Tabelle 6.5 recht hilfreich. Sie stehen dem Programmierer durch Einbinden der Headerdateien `float.h` bzw. `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 6.4: 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 Zahl mit $1.0 + eps \neq 1.0$
DBL_	wie oben für double
LDBL_	wie oben für long double

Tabelle 6.5: Konstanten aus limits.h

Konstante	Beschreibung
INT_MIN	Kleinste, darstellbare int Zahl
INT_MAX	Größte, darstellbare int Zahl
SHRT_	wie oben für short int

## 6.8 Typkonversion (cast)

### Beispiel 6.19 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 standardmäßig 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 6.20** Die im ersten Beispiel gezeigte abgeschnittene Division erlaubt in Verbindung mit dem Modulooperator `%` eine einfache Programmierung der Division mit Rest. *Übungsaufgabe* □

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.

## 6.9 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.

### 6.9.1 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 (Variablenamen, arithmetische Ausdrücke etc.). Die Formatstringkonstante enthält neben Text zusätzliche spezielle Zeichen: spezielle Zeichenkonstanten (Escapesequenzen) und Formatangaben.

Zeichenkonstante	erzeugt
<code>\n</code>	neue Zeile
<code>\t</code>	Tabulator
<code>\v</code>	vertikaler Tabulator
<code>\b</code>	Backspace
<code>\\</code>	Backslash <code>\</code>
<code>\?</code>	Fragezeichen <code>?</code>
<code>\'</code>	Hochkomma
<code>\"</code>	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
<code>%f</code>	float, double
<code>%i, %d</code>	int, short
<code>%u</code>	unsigned int
<code>%o</code>	int, short oktal
<code>%x</code>	int, short hexadezimal
<code>%c</code>	char
<code>%s</code>	Zeichenkette (String)
<code>%li, %ld</code>	long
<code>%Lf</code>	long double
<code>%e</code>	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 6.21 entnehmen.

### Beispiel 6.21 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
```

□

## 6.9.2 Eingabe

Für das Einlesen von Tastatureingaben des Benutzers steht u.a. 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 `&`.

### Beispiel 6.22 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 8), 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 sog. 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 7

## Programmflusskontrolle

### 7.1 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.

#### 7.1.1 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 7.1 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;
}

```

□

### 7.1.2 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 f\"ur Fall 1
        break;
    }
    ...
    case Fall n:
    {
        Anweisungen f\"ur Fall n
        break;
    }
    default:
    {
        Anweisungen f\"ur alle anderen F\"alle
        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 7.2 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;
}

```

□

## 7.2 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 6.4 benutzt.

### 7.2.1 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 7.3** Summe der natürlichen Zahlen von 1 bis n. Vergleich dazu auch Algorithmus 3.5.

```

#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  $i \leq n$  (ausdruck2) bzgl. 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.  $\square$

**Beispiel 7.4 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 dem Abschluss einer Anweisung “;“ und dem Trennzeichen “,” in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet.  $\square$

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 Beispiele 7.5, 7.6 verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tipps zu deren Umgehung.

**Beispiel 7.5** Ausgabe der diskreten Knoten  $x_i$  des Intervalls  $[a,b]$ , welches in  $n$  gleichgroße Teilintervalle zerlegt wird, d.h.

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

```

#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 ausgegeben wird. Auswege sind:

- 1.) Änderung des Abbruchtests in  $xi \leq b + h/2.0$  (jedoch ist  $x_n$  immer noch fehlerbehaftet).
- 2.) Keine forlaufende 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 3.2 zur Lösung der quadratischen Gleichung.

**Beispiel 7.6** Im diesem Beispiel wird die Summe  $\sum_{i=1}^n 1/i^2$  auf zwei verschiedene Arten berechnet:

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

Der Reihenwert ist  $\pi^2/6 = 1.644934068\dots$

```

#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 */
    /* nicht 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  $i*i > \text{INT\_MAX}$ . Dagegen erfolgt die Berechnung  $1.0/i/i$  vollständig im Bereich von Gleitkommazahlen.  $\square$

## 7.2.2 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 7.7 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;
}

```

□

### 7.2.3 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 7.8 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 z.B. die Zeichenkonstanten '0',..., '9' den Werten 48,...,57. □

### 7.3 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 8

## 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.

### 8.1 Adressen

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

#### Beispiel 8.1 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 8.2 Zu Beispiel 8.1.

- 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 der letzten Zeile 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.

□

## 8.2 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 8.3

```
#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\"o\3e %i \n"
, a, (unsigned int)&a, sizeof(a));
    printf("Variable pa : Inhalt = %u\t Adresse = %u\t Gr\"o\3e %i \n"
, (unsigned int)pa, (unsigned int)&pa, sizeof(pa));
    printf("Variable f : Inhalt = %f\t Adresse = %u\t Gr\"o\3e %i \n"
, f, (unsigned int)&f, sizeof(f));
    printf("Variable pf : Inhalt = %u\t Adresse = %u\t Gr\"o\3e %i \n"
, (unsigned int)pf, (unsigned int)&pf, sizeof(pf));
    return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Variable a : Inhalt = 16      Adresse = 2289604      Gr\"o\3e 4
Variable pa : Inhalt = 2289604 Adresse = 2289600      Gr\"o\3e 4
Variable f : Inhalt = 1.230000 Adresse = 2289592      Gr\"o\3e 8
Variable pf : Inhalt = 2289592 Adresse = 2289588      Gr\"o\3e 4
```

□

**Bemerkung 8.4 Zu Beispiel 8.3.**

- 1.) Da Pointervariablen wieder eine Speicheradresse besitzen, ist die Definition eines Pointers auf einen Pointer nicht nur sinnvoll, sondern auch nützlich (siehe Beispiel 8.9).
- 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).

□

## 8.3 Adressoperator und Zugriffoperator

Der unäre Adressoperator & (Referenzoperator)

& Variablenname

bestimmt die Adresse der Variable. Der unäre Zugriffoperator \* (Dereferenzoperator)

\* pointer

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

**Beispiel 8.5**

```
#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 */
            /* d.h. 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
```

□

## 8.4 Zusammenhang zwischen Zeigern und Feldern

Felder nutzen das Modell des linearen Speichers, d.h. 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 8.6 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 8.7 Zu Beispiel 8.6.

- 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 x sizeof(Typ)  
(und nicht (Adresse in pf) + 3 !!!)

- 3.) Der Zuweisungsoperator = ist für Felder nicht anwendbar, d.h. f=ausdruck ist nicht zulässig. Einzelne Feldelemente können jedoch wie gewohnt manipuliert werden (z.B. 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 Beispiel liefert sizeof(pf) den Wert 4 und sizeof(f) den Wert 16 (=4 x sizeof(float)).

Die folgenden Operatoren sind auf Zeiger anwendbar :

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

## 8.5 Dynamische Felder mittels Zeiger

Bisher wurde die Länge von Feldern bereits bei der Deklaration bzw. Definition angegeben. Da viele Aufgaben und Probleme stets nach dem selben 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:

<code>malloc()</code>	reserviert Speicher einer bestimmten Größe
<code>calloc()</code>	reserviert Speicher einer bestimmten Größe und initialisiert die Feldelemente mit 0
<code>realloc()</code>	erweitert einen reservierten Speicherbereich
<code>free()</code>	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, d.h. 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 8.8 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 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 8.9 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.

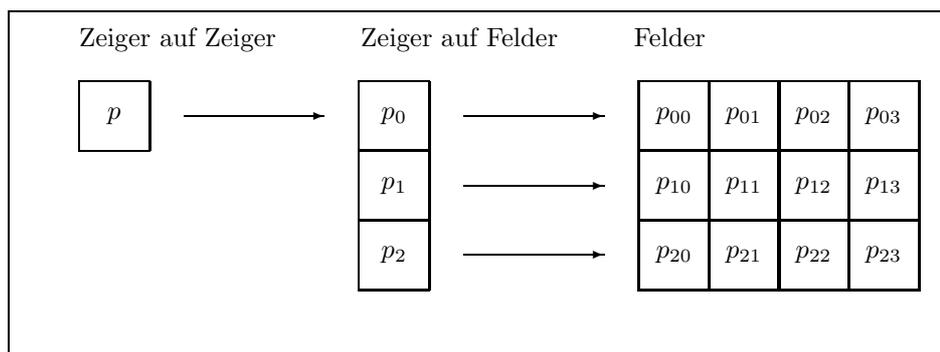


Abbildung 8.1: Dynamisches 2D Feld mit Zeiger auf Zeiger

**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 6.3.2), 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. (*Übungsaufgabe*)

# Kapitel 9

## 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 (z.B. `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.

### 9.1 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 z.B. 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 ggf. aus Deklarationen von weiteren Variablen (z.B. für Hilfsgrößen oder den Rückgabewert) sowie Anweisungen.

Folgendes ist bei Funktionen zu beachten:

- Die Deklaration bzw. 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, d.h. die Datentypen für Rückgabewert und Funktionsparameter müssen übereinstimmen.

### Beispiel 9.1 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 bzw. y */
    float maxi;

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

    return maxi;      /* Rückgabewert der Funktion */
}
```

□

## 9.2 Lokale und globale Variablen

Variablen lassen sich nach ihrem Gültigkeitsbereich unterteilen:

- *lokale Variablen*: Sie gelten in dem Anweisungsblock (z.B. Funktionsrumpf 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 (z.B. 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 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 bzw. Sichtbarkeit von Variablen.

### Beispiel 9.2 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 bzw. 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 verwendet. Auch wegen der Übersichtlichkeit empfiehlt es sich, für jede Variable einen anderen Namen zu verwenden. □

### 9.3 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 9.3 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 behält ihren Wert. □

#### Beispiel 9.4 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. □

## 9.4 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 9.5 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. □

## 9.5 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 jedesmal nur eine Teilaufgabe löst. Das führt zu rekursiven Aufrufen.

**Beispiel 9.6 Rekursive Programmierung.** Das folgende Programm berechnet  $x^k$ ,  $x \in \mathbb{R}$ ,  $k \in \mathbb{N}$ , 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

$$x^k = \underbrace{x(x(x(\dots 1)))}_k$$

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.

## 9.6 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 9.7 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 werden.

*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 *argument count* und *argument values* 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" usw.

*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.  $\square$

## 9.7 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 usw. bis zu einer einzelnen rechten Klammer. Dann fährt man links vom Bezeichner mit evtl. 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 9.8

$\underbrace{\text{char}}_7 \quad \underbrace{*}_6 \quad \underbrace{(\quad *)}_4 \quad \underbrace{(\quad *)}_2 \quad \underbrace{\text{Bezeichner}}_1 \quad \underbrace{(\quad )}_3 \quad \underbrace{[20]}_5$

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!  $\square$

**Beispiel 9.9** Die folgenden vier Beispiele verdeutlichen den Einsatz von Klammern:

(i)	<code>char * a[10]</code>	a ist ein Feld der Größe 10 mit Zeigern auf <code>char</code> -Werte
(ii)	<code>char (* a)[10]</code>	a ist Zeiger auf ein Feld der Größe 10 mit <code>char</code> -Werte
(iii)	<code>char *a(int)</code>	a ist Funktion die als Eingabeparameter einen <code>int</code> -Wert verlangt und einen Zeiger auf <code>char</code> -Wert zurückgibt
(iv)	<code>char (*a)(int)</code>	a ist Zeiger, auf eine Funktion die als Eingabeparameter einen <code>int</code> -Wert verlangt und einen <code>char</code> -Wert zurückgibt

## 9.8 Zeiger auf Funktionen

Manchmal ist es nützlich, Funktion an Funktionen zu übergeben. Dies kann mit Hilfe von Zeigern auf Funktionen realisiert werden.

**Beispiel 9.10 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 10

## 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 6.3 vorgestellt.

### 10.1 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.

#### 10.1.1 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 Struktur können nicht initialisiert werden:

```
struct Strukturname
{
    Datendeklaration
};
```

Im folgenden Beispiel wird eine Struktur mit Namen Student deklariert, die aus einer `int`-Variablen und einem String besteht.

**Beispiel 10.1 Deklaration einer Struktur.**

```
struct Student
{
    int matrikel;
    char name[16];
};
```

□

### 10.1.2 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 10.2 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. □

### 10.1.3 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 [].

### 10.1.4 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 10.3 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;
}

```

□

### 10.1.5 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 10.4 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;
}

```

□

### 10.1.6 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 10.5 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;
}

```

□

### 10.1.7 Listen

Wie in Abschnitt 10.1.6 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 10.6 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 Eintrag */

    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;
}

```

□

## 10.2 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 10.7 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 10.7: `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 werden, 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.

### 10.3 Aufzählungstyp

Der Aufzählungstyp ist ein Grundtyp mit frei wählbarem Wertebereich. Veranschaulicht wird dies am Beispiel der Wochentage.

#### Beispiel 10.8 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;
}
```

□

### 10.4 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 10.9 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

$$x = \operatorname{sgn}(x)B^N \sum_{n=1}^{\infty} x_{-n}B^{-n}, \quad (\text{A.1})$$

wobei  $2 \leq B \in \mathbb{N}$ ,  $N \in \mathbb{Z}$  und  $x_n \in \{0, 1, \dots, B-1\}$  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

$$m(x) = \sum_{n=1}^l x_{-n}B^{-n}. \quad (\text{A.2})$$

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

$$N_- \leq N \leq N_+, \quad N_-, N_+ \in \mathbb{Z}.$$

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

$$\frac{|\tilde{x} - x|}{|x|} \leq \frac{1}{2}B^{-l+1} =: \textit{eps}.$$

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.

$$\begin{aligned} & \overbrace{10110110100000000000000}^{23 \text{ Bit}} \\ & = 0.101101101_2 + 1 \\ & = 1.101101101_2 \end{aligned}$$

□

**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 precision.** Welche Dezimalzahl wird durch den Binärcode  $\underbrace{01000001}_{\text{Byte 3}} \underbrace{11011011}_{\text{Byte 2}} \underbrace{01000000}_{\text{Byte 1}} \underbrace{00000000}_{\text{Byte 0}}$  dargestellt?

$$\begin{aligned} & \begin{array}{ccc} \text{Bit 32} & \text{Bit 31-24} & \text{Bit 23-1} \\ \underbrace{0} & \underbrace{10000011} & \underbrace{10110110100000000000000} \\ \text{Zahl ist positiv} & N=131-127=4 & \text{Mantisse} \end{array} \\ & = 1.101101101_2 * 2^N \\ & = 11011.01101_2 \\ & = 2^4 + 2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3} + 2^{-5} \\ & = 27.406250 \end{aligned}$$

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 Auslöschung.** 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.  $\square$

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.
- [Car05] P.A. Carter. *PC Assembly Language*. online unter <http://www.drpaulcarter.com/pcasm/>, 2005.
- [DS04] Timothy A. Davis and Kermit Sigmon. *MATLAB Primer*. Chapman & Hall/CRC, 7th edition, 2004.

# Index

- ' , 21
- .\* , 22
- ./ , 22
- : , 21
- <= , 23
- < , 23
- == , 23
- >= , 23
- > , 23
- && , 23
- ~= , 23
- ~ , 23
- \ , 22
- atan , 22
- break , 24
- case , 23
- cd , 19
- clear , 19
- clf , 19
- cos , 22
- det , 22
- disp , 20
- edit , 19
- eig , 22
- elseif , 23
- else , 23
- exp , 22
- eye , 21
- format , 20
- for , 24
- help , 19
- hold , 24
- if , 23
- inv , 22
- i , 21
- j , 21
- log10 , 22
- log , 22
- ls , 19
- mesh , 24
- norm , 22
- otherwise , 24
- pi , 21
- plot , 24
- pwd , 19
- rand , 21
- rank , 22
- sin , 23
- size , 21
- sqrt , 23
- switch , 23
- tan , 23
- what , 19
- while , 24
- who , 20
- xor , 23
- zeros , 21
  
- Abrundung , 83
- Abschneiden , 83
- Adressoperator , 57, 58
- Allokieren , 60
- Alternative , 23
- Anfangsbedingung , 6
- Anweisung , 17
- array , 31
- ASCII-Tabelle , 55
- Aufrundung , 83
- Aufzählungstyp , 74
- Auslöschung , 86
  
- B-adische Entwicklung , 82
- bdf , 15
- Bezeichner , 71
- Bibliotheksfunktionen , 63
  
- C-Anweisung
  - break , 55
  - case , 49
  - continue , 55
  - do-while() , 54
  - for() , 50
  - goto , 55
  - if() , 48
  - switch() , 49
  - while() , 53
- C-Anweisung
  - calloc() , 60
  - free() , 60
  - malloc() , 60

- printf(), 27, 44
- realloc(), 60
- scanf(), 45
- sizeof(), 31
- call by reference, 68
- call by value, 66
- cast, 43
- cd, 15
- chmod, 15
- cp, 15
- Datentyp
  - char, 30
    - signed, 30
    - unsigned, 30
  - const, 30
  - double, 30
    - long, 30
  - enum, 74, 80
  - float, 30
  - int, 30
    - short, 30
    - signed, 30
    - unsigned, 30
  - pointer, 56
  - struct, 74
  - strukturierte, 74
  - typedef, 80
  - union, 74, 79
  - void, 30
  - zeiger, 56
- Datentypen, 29
- definiert, 29
- Deklaration, 29
- deklarieren, 29
- Dereferenzoperator, 58
- Differentialgleichung
  - gewöhnliche, 7
- Dirichlet–Randbedingung, 6
- enum, 80
- env, 15
- eps, 83
- Euler–Verfahren
  - explizites, 9
  - implizit, 10
- Exponentenüberlauf, 83
- Felder, 31
  - dynamisch, 60
- Festkommadarstellung, 82
- find, 15
- Fouriersches Gesetz, 5
- Funktion
  - call by reference, 68
  - call by value, 66
  - Funktionsparameter, 63
  - Funktionsrumpf, 64
  - Rückgabewert, 63
- Funktionen, 63
- Funktionsparameter, 63
- Funktionsrumpf, 64
- Gleitkommadarstellung, 83
- grep, 15
- Grundgesetz der Dynamik, 4
- gunzip, 15
- gzip, 15
- Header
  - float.h, 43
  - limits.h, 43
  - malloc.h, 60
  - math.h, 40
  - stdio.h, 27
  - string.h, 41
- Headerdatei, 27
- IEEE, 84
- Initialisierung, 29
- kill, 15
- Kommandozeilen–Parameter, 69
- Kommentare, 27
- Laplace-Gleichung, 7
- libraries, 26
- Listen, 78
- ll, 15
- ls, 15
- M–File, 19
- man, 15
- Mantisse, 82
- Mantissenlänge, 82
- Maschinengenauigkeit, 83
- Maschinenzahl, 82
- mkdir, 15
- Modellfehler, 7
- Monitor, 28
- more , 15
- mv, 15
- Neumann–Randbedingung, 7
- Operationen, 29
- Operator
  - Adressoperator, 57
  - Assoziativität, 38

- Auswahloperator, 76
- bitorientiert, 36
- dekrement, 37
- Dereferenzoperator, 58
- inkrement, 37
- logisch, 35
- Priorität, 38
- Referenzoperator, 58
- Strukturoperator, 75
- Vergleichsoperator, 35
- Zugriffsoperator, 58

Pointer, 56

Pointervariable, 57

Poisson-Gleichung, 7

Postfixnotation, 38

Präfixnotation, 37

Präprozessor, 27

Präprozessordirektiven, 27

Programmflusskontrolle, 48

ps, 15

pwd, 15

Rückgabewert, 63

Randbedingungen, 6

Referenzoperator, 58

rekursiv, 68

rm, 15

rmdir, 15

Rundung, 83

Rundungsfehler, 83

- absolut, 83
- relativ, 83

Schleife, 17

Schleifen, 50

- do-while, 54
- for, 50
- while, 53

Shell, 13

Signum-Funktion, 48

String, 31

Strings, 27

Struktur, 74

Suffix, 27

tail, 15

tar, 15

typedef, 80

typeset, 15

Typkonversion, 43

underscore, 29

Union, 74

union, 79

Variablen, 29

- Gültigkeit, 65
- global, 64
- lokal, 64
- Sichtbarkeit, 65

Verzweigung, 17, 48

w, 15

Wärmeleitungsgleichung, 6

which, 15

who, 15

Wiederholung, 17

Wurf, 4

yppasswd, 15

Zeichenketten, 27

Zeichenkettten, 32

Zeiger, 56

Zeiger auf Funktionen, 71

Zeigervariable, 57

Zugriffsoperator, 58

Zyklus

- abweisend, 18, 53
- nichtabweisend, 18, 54
- Zählzyklus, 50