Modellierung/Programmierung

Vorlesungsskript

Dipl.-Math. Erik Wallacher
FR 6.1 - Mathematik
Universität des Saarlandes

Wintersemester 2005/2006

Inhaltsverzeichnis

1 Das erste Programm

1.1 Allgemeine Grundlagen und Organisation

Die Programmentwicklung erfolgt grundsätzlich in vier Schritten.

1.) Problemstellung : Welche Aufgabe soll der Computer bewältigen.

2.) Erarbeiten eines Algorithmus zur Lösung der Aufgabe.

3.) Erstellen (editieren) eines Quelltextes zur Umsetzung des Algorithmus.

4.) Übersetzen (compilieren bzw. interpretieren) des Quelltextes in einen Code, der vom Computer verstanden werden kann.

Ein Computerprogramm ist also die Umsetzung eines Algorithmus in eine Form, die von einem Computer verarbeitet werden kann.
Der Begriff Programm wird sowohl für den in einer Programmiersprache verfassten Quelltext, als auch für den von einem Computer ausführbaren Maschinencode verwendet. Um aus dem Quelltext den Maschinencode zu generieren, wird ein Compiler oder Interpreter benötigt. Diese übersetzen die Befehle der Programmiersprache, die für menschliche Benutzer verständlich und bearbeitbar sein sollen, in die semantisch entsprechenden Befehle der Maschinensprache des verwendeten Computers.

1.2 Programmieren in C

Wie aus Kapitel 1.1 schon hervorgeht wird zum Programmieren (neben einem Computer) ein Editor und ein Übersetzer benötigt.

Editor

Ein Editor ist ein Programm zum Erstellen und Verändern von Textdateien. Speziell wird es bei der Programmierung zum Erstellen des Quellcodes verwandt.

Übersetzer (Compiler, Linker)

Wie bei allen Compiler-Sprachen besteht der Übersetzer aus zwei Hauptkomponenten: dem eigentlichen Compiler und dem Linker. 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 5) aus Bibliotheken (Libraries) einbindet.
Der Begriff Compiler wird häufig auch als Synonym für das gesamte Entwicklungssystem (Compiler, Linker, Bibliotheken) verwendet.

Bemerkung:
Für die Bearbeitung der Übungsaufgaben werden Editor und C-Compiler bereitgestellt. Es handelt sich dabei um frei erhältliche (kostenlose) Programme, die auf LINUX-Betriebssystemen (siehe Kapitel 1.3) arbeiten. Grundlegende Kenntnisse über LINUX werden in der ersten Übung vermittelt. Die Aufgaben können an den an der Universität verfügbaren Rechnern oder in Heimarbeit erledigt werden. Für die Bearbeitung unter Windows siehe Kapitel 1.4.

Warnung:

Es handelt sich bei C um eine weitgehend standardisierte Programmiersprache. Dennoch 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 den von uns bereitgestellten Compilern arbeitet, hat unter Umständen mit Schwierigkeiten bei der Kompatibilität zu rechnen. In den Übungen kann hierauf keine Rücksicht genommen werden.

1.3 Das Betriebssystem LINUX

LINUX wurde von dem finnischen Informatikstudenten Linus Torvals entwickelt, um PCs kompatibel zu den u.a. im universitären Umfeld weit verbreiteten UNIX-Systemen zu betreiben. Die rasch wachsende Beliebheit führte dazu, dass heute eine großen Programmierergemeinde zum größten Teil unentgeltlich die Entwicklung weiterführt. LINUX ist frei erhältlich und kann kostenlos weitergeben werden. Das System wird in unterschiedlichen Varianten (Distributionen) angeboten.

Dateien und Verzeichnisse

Daten werden auf dem Medium in Dateien zusammengefasst abgespeichert. Die Datei wird mit einem bestimmten Dateinamen bezeichnet.
Folgendes ist für Dateinamen zu beachten:

- LINUX unterscheidet zwischen Groß- und Kleinschreibung.

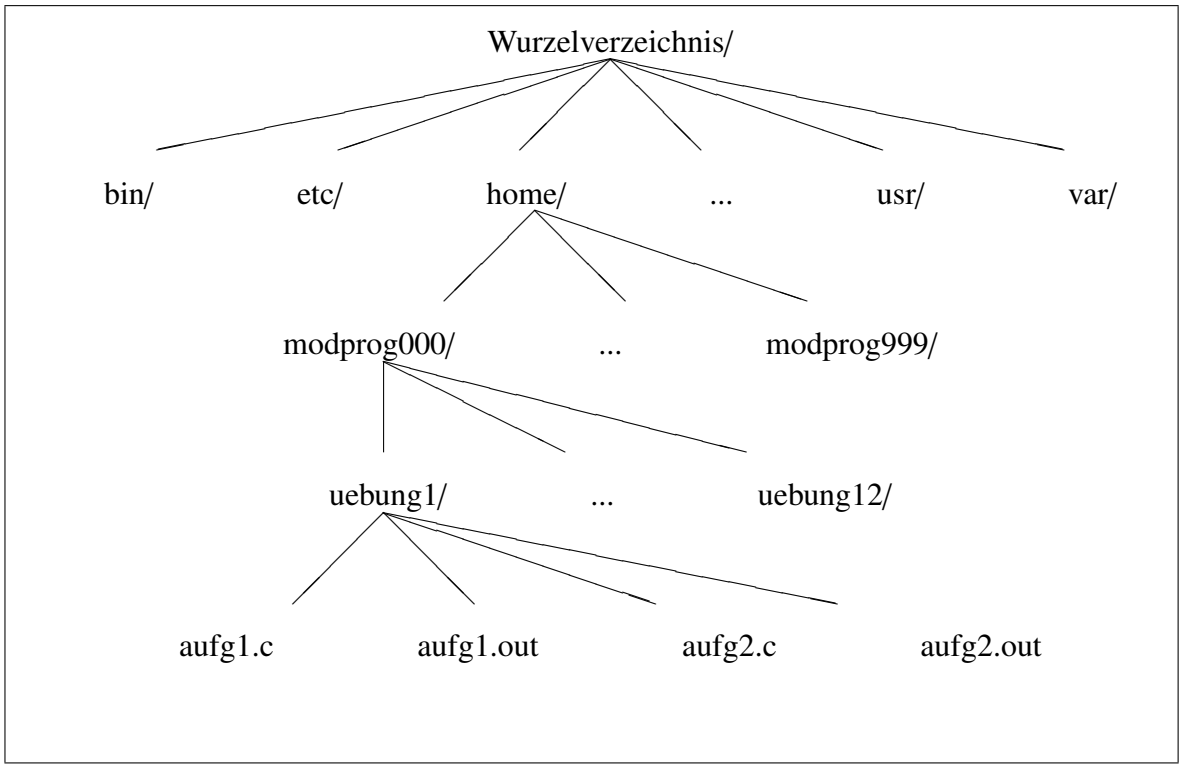

Abbildung 1.1: Der Verzeichnisbaum am Beispiel von LINUX

- Der Schrägstrich / darf nicht verwendet werden, da er zur Trennung von Verzeichnisnamen verwendet wird.
- Sonderzeichen sollte man nicht verwenden, da einige eine spezielle Bedeutung haben.

Mehrere Dateien können in einem Verzeichnis gesammelt werden. LINUX besitzt ein hierarchisches, baumstrukturiertes Dateisystem: Ausgehend von dem Wurzelverzeichnis / besitzt der Verzeichnisbaum die Struktur in Abbildung 1.1.

Die Position eines beliebigen (Unter-)Verzeichnisses innerhalb des Dateisystems ist durch den Pfad gegeben. Der Pfad gibt an, wie man vom Wurzelverzeichnis zu der gewünschten Datei bzw. dem gewünschten Verzeichnis gelangt. Die Unterverzeichnisnamen im Pfad werden durch den Schrägstrich getrennt.
Pfade, die mit / beginnen, heißen **absolute Pfade**. Daneben gibt es die so genannten **relativen Pfade**, die relativ zur aktuellen Position im Verzeichnisbaum aufzufassen sind. Die folgenden relativen Pfadbezeichner haben eine besondere Bedeutung:

.	aktuelle Position im Verzeichnisbaum (Arbeitsverzeichnis)
..	das übergeordnee Verzeichnis

Der absolute Pfad der Datei aufg1.c lautet beispielsweise */home/modprog000/uebung1/aufg1.c*.

Es ist möglich, mehrere Dateien oder Verzeichnisse gleichzeitig anzusprechen. Dazu dienen die so genannten Wildcards:

*	wird durch beliebig viele Zeichen (auch keines) ersetzt
?	wird durch ein Einzelzeichen ersetzt
[zeichen1 - zeichen2]	Auswahlsequenz; wird ersetzt durch alle Zeichen zwischen zeichen1 und zeichen2

Die Shell

Ein elementares Programm ist die Shell. Sie ist ein Kommandointerpreter, der von der Kommandozeile Anweisungen einliest, auf ihre Korrektheit überprüft und ausführt. Hier ist eine kleine Übersicht über die allerwichtigsten Befehle (Kommandos).

cd Verzeichnis	wechsele in das angegebene Verzeichnis
pwd	zeige die aktuelle Position im Verzeichnisbaum an
cp Datei Pfad cp -R Verzeichnis1 Verzeichnis2	kopiere Datei an die angegebene Position kopiere rekursiv, d.h. mit allen Unterverzeichnissen
mv Pfad1 Pfad2	verschiebe Pfad1 nach Pfad2
ls ls Verzeichnis ls -l Pfad	zeige Inhalt des aktuellen Verzeichnisses an zeige Inhalt von Verzeichnis an ausführliche Anzeige von Informationen
rm Datei rm -i Datei rm -f Datei rm -Rf Verzeichnis	lösche Datei lösche Datei mit Bestätigung lösche Datei ohne Bestätigung lösche Verzeichnis rekursiv ohne Bestätigung
mkdir Verzeichnis rmdir Verzeichnis	erzeuge neues Verzeichnis lösche (leeres) Verzeichnis
cat Datei	zeige Inhalt von Datei
more Datei	zeige Inhalt von Datei seitenweise

Hilfe zu Kommandos

Die man-Anweisung zeigt eine Dokumentation (Manpage) zu einem Befehl an. Man verwendet sie einfach durch

man Befehlsname

Sucht man Befehle, die mit einem bestimmten Schlüsselbegriff zusammenhängen, so leistet der Aufruf

man -k Schlüsselbegriff

das Gewünschte.

1.4 Arbeiten unter Windows

Wie schon in Kapitel 1.2 angemerkt, kann es bei Verwendung von unterschiedlichen C-Compilern zu Schwierigkeiten kommen. In dieser Vorlesung beziehen wir uns stets auf den unter LINUX verwendeten GNU-C-Compiler. Für die Nutzung dieses Programms unter Windows muss eine LINUX ähnliche Plattform unter Windows simuliert werden. Diese Aufgabe übernimmt das sogenannte Programmpaket **Cygwin**. Wir stellen hierfür eine Basisausstattung von Cygwin bereit. Auf Anfrage erhält jeder Teilnehmer leihweise eine CD mit Installationsanleitung.

1.5 Ein erstes C-Programm

Aufgabe: Der Computer soll nur eine Meldung auf dem Bildschirm schreiben.

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 hallo.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).

1.6 Interne Details beim Compilieren

Der leicht geänderte Aufruf zum Compilieren

gcc -v hallo.c

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

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

gcc -E hallo.c > hallo.E

Der Zusatz > hallo.E lenkt die Bildschirmausgabe in die Datei hallo.E. Diese Datei hallo.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 hallo.c

Die entstandene Datei hallo.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 hallo.c

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

hexedit hallo.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* hallo.c

2 Variablen, Datentypen und Operationen

2.1 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 findet vor der ersten Ausführungsanweisung statt.
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.

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	

Abbildung 2.1: Elementare Datentypen. Die Bytelänge ist von Architektur zu Architektur unterschiedlich (hier: GNU-C-Compiler unter LINUX für die x86-Architektur. Siehe auch sizeof()).

2.2 Elementare Datentypen

Die folgende Tabelle 2.1 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.

Beispiel 2.1 (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 2.2 (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 2.4) 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 1zeichen; (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 2.3 (C-Anweisung : sizeof())

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

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

2.3 Felder und Strings

2.3.1 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.
- Felder können bei der Deklaration initialisiert werden. Dies geschieht unter Verwendung des Zuweisungsoperators und der geschweiften Klammer.

Beispiel 2.4 (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" ,a[0] );
    /* Die Formatangabe %f zeigt an, dass eine float bzw.
    double-Variable ausgegeben wird */
    return 0;
 }
```

2.3.2 Mehrdimensionale Felder

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

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

Beispiel 2.5 (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",a[0][1]);
    return 0;
}
```

2.3.3 Zeichenketten (Strings)

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

char *Stringname* [Länge]

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

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

initialisiert.

Ein String kann auch durch

char text[]="Hallo";

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

Beispiel 2.6 (Deklaration und Initialisierung eines Strings)

```
#include <stdio.h>

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

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

2.4 Ausdrücke, Operatoren und mathematische Funktionen

Der Zuweisungsoperator

$$operand1 = operand2$$

weist dem linken Operanden den Wert des rechten Operanden zu.

Zum Beispiel ist im Ergebnis der Anweisungsfolge

Beispiel 2.7 (Zuweisungsoperator)

```
#include <stdio.h>

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

        return 0;
}
```

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

Es können auch Mehrfachzuweisungen auftreten. Die folgenden drei Zuweisungen sind äquivalent.

Beispiel 2.8 (Mehrfachzuweisung)

```
#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",a,b,c);
        return 0;
}
```

2.4.1 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 exakte Ergebnis. z.B. 8.0/3 liefert 2.66666 als Ergebnis (siehe auch Kapitel 2.8).

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

Beispiel 2.9 (Arithmetische Operatoren)

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

int main()
```

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

        b=6;            /* b=6 */
        c=10;           /* c=10 */
        x=b/c;          /* x=0 */
        x=(double) b/c; /* x=0.6 siehe Kapitel 2.8 */
        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;
}
```

2.4.2 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 2.10 (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;
}
```

2.4.3 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

Operator	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.

2.4.4 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:

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

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

Beispiel 2.11 ((Short)-Integerdarstellung im Rechner)

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

Unäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
~	Binärkomplement	~ a

Binäre bitorientierte Operatoren

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

Wahrheitstafel

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Beispiel 2.12 (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;
}
```

Anwendungen zu bitorientierten Operatoren werden in den Übungen besprochen.

2.4.5 Inkrement- und Dekrementoperatoren

Präfixnotation

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

Beispiel 2.13 (Präfixnotation)

```
#include <stdio.h>

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

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

        /* oben angegebene Notation ist aequivalent zu */

        i=i+1;
        j=i;

        return 0;
}
```

Postfixnotation

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

Beispiel 2.14 (Postfixnotation)

```
#include <stdio.h>

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

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

        /* oben angegebene Notatation ist aequivalent zu */

        j=i;
        i=i+1;

        return 0;
}
```

2.4.6 Adressoperator

Der Vollständigkeit halber wird der Adressoperator " & " schon in diesem Kapitel eingeführt, obwohl die Bedeutung erst in Kapitel 4 klar wird.

& datenobjekt

2.4.7 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 exklusivers 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 2.15 (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;
}
```

2.5 Operationen mit vordefinierten Funktionen

2.5.1 Mathematische Funktionen

Im Headerfile math.h werden u.a. Deklarationen der in Tabelle 2.1 zusammengefassten mathematischen Funktionen und Konstanten bereitgestellt:

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

Tabelle 2.1: Mathematische Funktionen

Für die Zulässigkeit der Operation, d.h. den Definitionsbereich der Argumente, ist der Programmierer verantwortlich. Ansonsten werden Programmabbrüche oder unsinnige Ergebnisse produziert.

Beispiel 2.16 (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;
}
```

2.5.2 Funktionen für Zeichenketten (Strings)

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

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

Tabelle 2.2: Funktionen für Strings

Beispiel 2.17 (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 für s2 */
        char s3[100]="Hallo"; /* reserviert 100 Byte im Speicher für 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 – die Alternative sind seltsame und schwer zu findende Fehler des Programms (siehe auch Beipiel 2.17).

2.6 Zusammengesetzte Anweisungen

Wertzuweisungen der Form

$$\text{op1}=\text{op1 operator op2}$$

können zu

$$\text{op1 operator} = \text{op2}$$

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

Beispiel 2.18 (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) */
        j *= i; /* j=j*i */

        return 0;
}
```

2.7 Nützliche Konstanten

Für systemabhängige Zahlenbereiche, Genauigkeiten usw. ist die Auswahl der Konstanten aus Tabelle 2.3 und Tabelle 2.4 recht hilfreich. Sie stehen dem Programmierer durch Einbinden der Headerdateien *float.h* bzw. *limits.h* zur Verfügung.

Tabelle 2.3: Konstanten aus float.h

Konstante	Beschreibung
FLT_DIG	Anzahl gültiger Dezimalstellen für float
FLT_MIN	Kleinste, darstellbare echt 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 2.4: Konstanten aus limits.h

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

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

2.8 Typkonversion (cast)

Beispiel 2.19 (Abgeschnittene Division)
Nach der Zuweisung hat die Variable quotient den Wert 3.0, obwohl sie als Gleitkommazahl deklariert wurde!
Ursache: Resultat der Division zweier int-Variablen ist standardmäsig wieder ein int-Datenobjekt.

```
#include <stdio.h>

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

        return 0;
}
```

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.

Vorsicht 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 2.20
Die im ersten Beispiel gezeigte abgeschnittene Division erlaubt in Verbindung mit dem Modulooperator % eine einfache Programmierung der Division mit Rest.

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

quotient = (float) 10/b;

kann man die Anweisung

quotient = 10.0/b;

verwenden.

2.9 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.

2.9.1 Ausgabe

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

printf(*Formatstringkonstante, Argumentliste*);

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

Zeichenkonstante	**erzeugt**
\n	neue Zeile
\t	Tabulator
\v	vertikaler Tabulator
\b	Backspace
\\	Backslash \
\?	Fragezeichen ?
\’	Hochkomma
\”	Anführungsstriche

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

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

- Durch Einfügen eines Leerzeichens nach % wird Platz für das Vorzeichen ausgespart. Nur negative Vorzeichen werden angezeigt. Fügt man stattdessen ein + ein, so wird das Vorzeichen immer angezeigt.
- Weitere Optionen kann man aus Beispiel 2.21 entnehmen.

Beispiel 2.21 (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

2.9.2 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 2.22 (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",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 4), so entfällt der Adressoperator & im scanf()-Befehl.
Bsp.:
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];

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

3 Programmflusskontrolle

3.1 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:

3.1.1 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. Dies sollte dann aber durch Schreiben der einzigen Anweisung hinter den *if*-Befehl deutlich gemacht werden!

Beispiel 3.1 (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;
}
```

3.1.2 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 für Fall 1
              break;
      }
      ...
      case Fall n:
      {
              Anweisungen für Fall n
              break;
      }
      default:
      {
```

```
            Anweisungen für alle anderen Fälle
            break;
        }
  }
```

Beispiel 3.2 (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;
}
```

3.2 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 2.4 benutzt.

3.2.1 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
}
```

Am besten versteht man den Zählzyklus an einem Beispiel.

Beispiel 3.3 (Summe der natürlichen Zahlen von 1 bis n)

```
#include <stdio.h>

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

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

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

Eine kompakte Version dieser Summationsschleife (korrekt, aber sehr schlecht lesbar) wäre:

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

Man unterscheidet dabei zwischen den Abschluss einer Anweisung ";" und dem Trennzeichen "," in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet. Der ausdruck2 ist stets ein logischer Ausdruck und ausdruck3 ist ein arithmetischer Ausdruck zur Manipulation der Laufvariablen. Die Laufvariable kann eine einfache Variable vom Typ int, float oder double sein.

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

Die folgenden Beipiele 3.4, 3.5 verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tipps zu deren Umgehung.

Beispiel 3.4
Ausgabe der diskreten Knoten x_i des Intervalls [a,b], welches in n gleichgroße Teilintervalle zerlegt wird, d.h.

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

```
#include <stdio.h>

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

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

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

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

Da Gleitkommazahlen nur eine limitierte Anzahl gültiger Ziffern besitzt, kann es (meistens) passieren, dass der letzte Knoten $x_n = b$ nicht ausgegeben wird. Auswege sind:

1.) Änderung des Abbruchtests in $xi <= b + h/2.0$ (jedoch ist x_n immer noch fehlerbehaftet).

2.) Zyklus mit int-Variable

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

Die gemeinsame Summation kleinerer und größerer Zahlen kann ebenfalls zu Ungenauigkeiten führen. Im Beispiel 3.5 wird die Summe $\sum_{i=1}^{n} 1/i^2$ auf zwei verschiedene Arten berechnet.

Beispiel 3.5

```
#include <stdio.h>
#include <math.h>
#include <limits.h> /* enthält 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 für n bei ca. %f \n",
              ceil(sqrt(1.0/1.0e-6)) );
       /* siehe Kommentar 1.Schranke */

       printf("Weitere Fehler ergeben sich für 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 für i */
              /* Der Summand 1.0/(i*i) wird bei der Addition */
              /* nich mehr berücksichtigt, falls 1.0/(i*i) < 1.0e-6 */

              /* 2. Schranke für i */
              /* Das Produkt i*i ist als int-Variable nicht */
```

```
                /* mehr darstellbar, falls i*i > INT_MAX */

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

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

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

*Das numerische Resultat in summe2 ist genauer, da dort zuerst alle kleinen Zahlen addiert werden, welche bei summe1 wegen der beschränkten Anzahl gültiger Ziffern keinen Beitrag zur Summation mehr liefern können. Gleichzeitig ist zu beachten, dass die Berechnung von i*i nicht mehr in int-Zahlen darstellbar ist für i*i > INT_MAX. Dagegen erfolgt die Berechnung 1.0/i/i vollständig im Bereich von Gleitkommazahlen.*

3.2.2 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 3.6 (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'␣für␣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' für Ende) : ");
                fgets(tempstring, sizeof(tempstring), stdin);
                sscanf(tempstring, "%f", &zahl);
        }
        return 0;
}
```

3.2.3 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 Zykleninneren.

Die allgemeine Form ist

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

Beispiel 3.7 (do-while-Schleife)
Es wird solange eine Zeichenkette von der Tastatur eingelesen, bis die Eingabe eine ganze Zahl ist.

```
#include <stdio.h>
#include <math.h>  /* Für pow */

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

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

                /* Es wird nacheinander geprüft 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 Länge des Strings == i */
        for (i=0;i<=n-1;i++)
        {
                zahl=zahl+ (text[i]-'0')*pow(10,n-1-i);
        }
        printf("Die Eingabe %s entspricht der ganzen 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.

3.3 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**.

4 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.

4.1 Adressen

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

Beispiel 4.1 (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 für 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 4.2 (Bemerkung zu Beispiel 4.1)

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.*

4.2 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 4.3

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

Das Programm erzeugt folgende Ausgabe:

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

Bemerkung 4.4 (Bemerkung zu Beispiel 4.3)

1.) *Da Pointervariablen wieder eine Speicheradresse besitzen, ist die Definition eines Pointers auf einen Pointer nicht nur sinnvoll, sondern auch nützlich (siehe Beispiel 4.9).*

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).*

4.3 Adressoperator und Zugriffsoperator

Der unäre Adressoperator & (Referenzoperator)

& Variablenname

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

* pointer

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

Beispiel 4.5

```
#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 erhöht */
                /* 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
```

4.4 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 4.6 (Zeiger und Felder)

```
#include <stdio.h>

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


        pf=f;  /* Äquivalent wäre die Zuweisung pf=&f[0] */
```

```
        /* Nicht Zulässige Operationen mit Feldern */
        /* f=g;
         * f=f+1; */

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

Bemerkung 4.7 (zu Beispiel 4.6)

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 Beipiel 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 +=, -=

4.5 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:

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

Die Funktionen *malloc()*, *calloc()* und *realloc()* versuchen, den angeforderten Speicher bereitzustellen (allokieren), und liefern einen Pointer auf diesen Bereich zurück. Konnte die Speicheranforderung nicht erfüllt werden, wird ein Null-Pointer (NULL in C, 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.
Das folgende Programm demonstriert die Reservierung von Speicher durch die Funktion *malloc()* und die Freigabe durch *free()*.

Beispiel 4.8 (Norm des Vektors (1,...,n))

```
#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) /* Genügend Speicher vorhanden? */
       {
              printf("Nicht␣genügend␣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 4.9 (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);

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

        for (i=0;i<m;i++)
        {
                /* Allokalisiert Speicher für die Spalten 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 Spalten */
        }
        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.

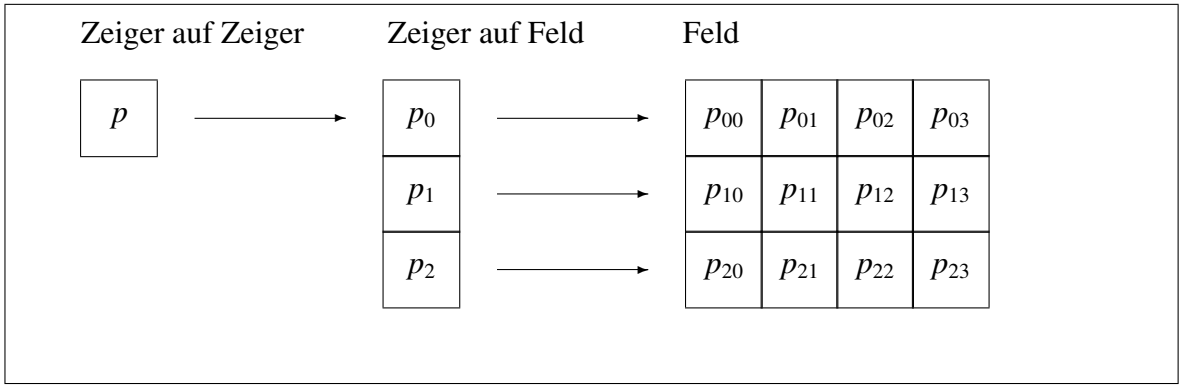

Abbildung 4.1: Dynamisches 2D Feld mit Zeiger auf Zeiger

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 2.3.2), 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.

5 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.

5.1 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 5.1 (Funktion)

```
#include <stdio.h>

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

/* Deklaration globaler Variablen (nur für den Debugger) */
/*********************************************************/
float *pa=(float *)&pa;
float *pb=(float *)&pb;
float *px=(float *)&px;
float *py=(float *)&py;
/*********************************************************/


/* Hauptprogramm                                         */
/*********************************************************************/
int main()
{
       float a=3.0,b=2.0;
       pa=&a;pb=&b; /* pa, pb Zeiger auf a bzw. b */
       printf("Das␣Maximumvon␣%f␣und␣%f␣ist␣%f",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;
       px=&x;py=&y; /* px, py Zeiger auf x bzw. y */

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

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

5.2 Lokale und globale Variablen

Variablen lassen sich nach ihrem Gültigkeitsbereich unterteilen:

lokale Variablen: Sie gelten in dem Anweisungsblock (z.B. Funktionenrumpf oder Schleifenrumpf), in dem sie deklariert wurden. Für diesen Block gelten sie als lokal. Bei der Ausführung des Programms existieren die Variablen bis zum Verlassen des Anweisungsblocks.

globale Variablen: Sie werden außerhalb aller Funktionen deklariert/definiert (z.B. direkt nach den Präprozessordirektiven) und sind zunächst im gesamten Programm einschließlich aller Subroutinen 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 5.2 (Gültigkeitsbereich von Variablen)

```
/*************************************************/
/* Im Beispiel wird 4 x die Variable a deklariert
 * Mit Hilfe eines Debuggers und der Zeiger
 * pa_global, pa_main , pa_sub_main und pa_summe
 * kann jeweils die Adresse und deren zugehöriger
 * Inhalt angezeigt werden */
/*************************************************/
```

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

/* Deklaration globaler Variablen         */
/******************************************/
int a=1;              /* globales a =1     */
int *pa_global=(int *) &pa_global;
int *pa_main=(int *) &pa_main;
int *pa_sub_main=(int *) &pa_sub_main;
int *pa_summe=(int *) &pa_summe;
/******************************************/

int main()
{
       pa_global=&a; /* Zeiger auf globales a */
       int a=2;      /* a in main() = 2 und
                      * überdeckt globales a*/

       pa_main=&a;   /* Zeiger auf a in main() */
       {
              int a=2;      /* lokales a in main() = 2 und
                             * überdeckt a in main() */

              pa_sub_main=&a; /* Zeiger auf lokales a in main() */

              a=a+1;        /* lokales a in main() wird um 1 erhöht */
              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 erhöht */
              /* lokales a in main() wird gelöscht */
       }

       a=summe(a,a);  /* a in main = 2 x lokales a in main()
                       * Gleichzeitig wird das globale a in
                       * der Funktion summe um 1 erhöht */

       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 */

        a=a+1; /* globales a wird um 1 erhöht */

        int a=0;      /* a in der Funktion summe() */
        pa_summe=&a;  /* Zeiger auf a in der Funkton summe */

        a=x+y;        /* a in der Funktion summe = x+y */

        return a;     /* a in der Funktion wird zurückgegeben */

        /* a,x,y in der Funktion summe werden gelöscht */
}
/*****************************************************************/
```

5.3 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 5.3 (Call by value I)

```
#include <stdio.h>

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

/* Deklaration globaler Variablen (nur für den Debugger) */
/*********************************************************/
int *pb=(int *)&pb;
int *pb_setze=(int *)&pb_setze;
/*********************************************************/


/* Hauptprogramm         */
/*****************************/
int main()
{
```

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

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

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 behielt ihren Wert.

Beispiel 5.4 (Call by value II)

```
#include <stdio.h>

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

/* Deklaration globaler Variablen (nur für den Debugger) */
/*********************************************************/
int *pb=(int *)&pb;
int *pb_setze=(int *)&pb_setze;
/*********************************************************/


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

```
/*****************************/

/* Definition der Funktion setze () */
/************************************/
int setze (int b)
{
       b=3;pb_setze=&b;
       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.

5.4 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 5.5 (Call by reference)

```
#include <stdio.h>

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

/* Deklaration globaler Variablen (nur für den Debugger) */
/*********************************************************/
int *pb=(int *)&pb;
/*********************************************************/


/* Hauptprogramm          */
/*****************************/
int main()
```

```
{
        int b=0;pb=&b;
        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.

5.5 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.

Das folgende Programm berechnet x^k ($x \in \mathbb{R}$, $k \in \mathbb{N}$) rekursiv.

Beispiel 5.6 (Rekursive Programmierung)

```
#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));
}

/* 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 */
                }
        }
}
```

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 zur Unendlichkeit rechnet.

5.6 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 5.7

```
/* 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

```
beispiel5_7.out 1.4 3.2 4.5
```

erscheint folgende Ausgabe auf dem Bildschirm

```
Parameter 0 = beispiel5_7
Parameter 1 = 1.4
Parameter 2 = 3.2
Parameter 3 = 4.5

Die Summe der Kommandozeilen-Parameter : 9.100000
```

Die Angaben in der Kommandozeile sind an das Programm übergeben worden und konnten hier auch verarbeitet werde.

Zeile 3: ***main*** erhält vom Betriebssystem zwei Parameter. Die Variabel ***argc*** enthält die Anzahl der übergebenen Parameter und ***argv[]*** die Parameter selbst. Die Namen der Variablen ***argc*** und ***argv*** sind natürlich frei wählbar, es hat sich jedoch eingebürgert, die hier verwendeten Bezeichnungen zu benutzen. Sie leiten sich von **arg**ument **c**ount und **arg**ument **v**alues ab. Bei ***argc*** ist eine Besonderheit zu beachten. Hat diese Variable zum Beispiel den Wert 1, 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, 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*** ange-

sprochen 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.

5.7 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 5.8

$$\underbrace{char}_{7}\ \underbrace{*}_{6}\ \underbrace{(*}_{4}\ \underbrace{(*}_{2}\ \underbrace{Bezeichner)}_{1}\ \underbrace{()}_{3}\ )\ \underbrace{[20]}_{5}$$

Bezeichner (1) ist hier ein Zeiger (2) auf eine Funktion (3), die einen Zeiger (4) auf ein Feld mit 20 Elementen(5) zurückgibt, die Zeiger (6) auf char-Werte(7) sind!

Die folgenden vier Beispiele sollen noch einmal den Einsatz von Klammern verdeutlichen:

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

5.8 Zeiger auf Funktionen

Manchmal ist es nützlich Funktion an Funktionen zu übergeben. Dies kann mit Hilfe von Zeigern auf Funktionen realisiert werden. Im folgenden Beispiel (5.9) 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.

Beispiel 5.9

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

/* Deklaration der Funktion trapez_regel()          */
/*****************************************************/
double trapez_regel(double (*f)(double ),double ,double ,int );
/* Eingabeparameter : 1.) Zeiger auf Funktion mit
 *                       Eingabeparameter double-Wert
 *                       und double-Rückgabewert
 *                  2.) double für linke Intervallgrenze
 *                  3.) double für rechte Intervallgrenze
 *                  4.) int für Anzahl der Stützstellen
 * Rückgabewert : double für das Integral
*****************************************************/

/* Hauptprogramm                                    */
/****************************************************/
int main(int argc,char** argv)
{
       int n;
       double a,b,integral;

       /* Zeiger auf eine Funktion die als Rückgabewert
        * eine double-Variable besitzt und als
        * Eingabe eine double-Variable verlangt  */
       double (*fptr)(double);

       if (argc<4)
       {
              printf("Programm benötigt 3 Kommandozeilenparameter :\n");
              printf("1.) Linker Intervallrand (double)\n");
              printf("2.) Rechter Intervallrand (double)\n");
              printf("3.) Anzahl der Teilintervalle für");
              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␣über␣das␣Intervall");
            printf("␣[%f␣,␣%f]\n",a,b);
            printf("beträgt␣:␣\t␣%f␣(numerisch␣mit␣%i",integral,n+1);
            printf("␣Stützstellen)\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␣über␣das␣Intervall");
            printf("␣[%f␣,␣%f]\n",a,b);
            printf("beträgt␣:␣\t␣%f␣(numerisch␣mit␣%i",integral,n+1);
            printf("␣Stützstellen)\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;
}
/*************************************************************************/
```

6 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 2.3 vorgestellt.

6.1 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.

6.1.1 Deklaration von Strukturen

Zur Deklaration einer Struktur benutzt man das Schlüsselwort *struct*. Ihm folgt der Name der Struktur. In geschweiften Klammern werden dann die einzelnen Variablen aufgeführt, aus denen die Struktur bestehen soll.
Achtung: Die Variablen innerhalb einer Stuktur können nicht initialisiert werden.

```
struct Strukturname  Datendeklaration ;
```

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

Beispiel 6.1 (Deklaration einer Struktur)

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

6.1.2 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 6.2 (Deklaration einer Struktur + 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.

6.1.3 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 Studenten2005[1000];

Der Zugriff auf einzelne Feldelemente erfolgt wie gewohnt mit eckigen Klammern [].

6.1.4 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 6.3 (Deklaration + Definition + Zuweisung)

```
#include <stdio.h>
#include <string.h> /* Für 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",Mathe[2].Vorname);
}
        return 0;
```

6.1.5 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 6.4 (Zugriff mit Zeigern)

```
#include <stdio.h>
#include <string.h> /* Für 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␣"
        ,paul.matrikel,paul.Strasse);
        return 0;
}
```

6.1.6 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.
Das Programm 6.5 zeigt ein Beispiel für eine geschachtelte Struktur.

Beispiel 6.5 (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;
}
```

6.1.7 Listen

Wie in Abschnitt 6.1.6 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 6.6 (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 nächsten Eintag */

        strcpy(Mathe[1].name,"Claudia");
        strcpy(Mathe[1].familienstand,"verheiratet");
        strcpy(Mathe[1].geschlecht,"weiblich");
        Mathe[1].next=NULL; /* next zeigt auf NULL, d.h Listenende */

        /* Initialisierung der Bio-Liste */
        strcpy(Bio[0].name,"Peter");
        strcpy(Bio[0].familienstand,"geschieden");
        strcpy(Bio[0].geschlecht,"männlich");
        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);
        }

        /* Anhängen 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;
}
```

6.2 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 6.7 (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;
        strcpy(ergebnispointer->nichtbestanden,"nicht␣bestanden");
        printf("%s\n",ergebnispointer->nichtbestanden);
```

```
        /* Zugriff ohne pointer */

        ergebnis.note=3.3;
        strcpy(ergebnis.nichtbestanden,"nicht bestanden");
        printf("%s\n",ergebnis.nichtbestanden);
        return 0;
}
```

Der Speicherplatzbedarf einer Union richtet sich nach der größten Komponente (im Beispiel: 16 x sizeof(char) = 16 Byte).
Gründe für das Benutzen von Unions fallen nicht so stark ins Auge wie bei Strukturen. Im wesentlichen sind es zwei Anwendungsfälle, in denen man sie einsetzt:

- Man möchte auf einen Speicherbereich auf unterschiedliche Weise zugreifen. Dies könnte bei der obigen Union doppelt der Fall sein. Hier kann man mit Hilfe der Komponente *nichtbestanden* auf die einzelnen Bytes der Komponente *note* zugreifen.
- Man benutzt in einer Struktur einen Bereich für verschiedene Aufgaben. Sollen beispielsweise in einer Struktur Mitarbeiterdaten gespeichert werde, so kann es sein, dass für den einen Angaben zum Stundenlohn in der Struktur vorhanden sein sollen, während der andere Speicherplatz für ein Monatsgehalt und der Dritte noch zusätzlich Angaben über Provisionen benötigt. Damit man nun nicht alle Varianten in die Struktur einbauen muss, wobei jeweils zwei unbenutzt blieben, definiert man einen Speicherbereich, in dem die jeweils benötigten Informationen abgelegt werden, als Union.

6.3 Aufzählungstyp

Der Aufzählungstyp ist ein Grundtyp mit frei wählbarem Wertebereich. Veranschaulicht wird dies am Beipiel der Wochtage.

Beispiel 6.8 (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("Schlechte Laune");
        return 0;
}
```

6.4 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 6.9 (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.

7 Arbeiten mit Dateien

Viele der Programme, die bisher vorgestellt wurden, erlaubten es, Daten einzugeben, zu bearbeiten und wieder auszugeben. Nach dem Programmende waren diese Zahlen und Texte dann allerdings wieder verloren. Was also noch fehlt, ist die Möglichkeit, die Daten auf Diskette oder Platte dauerhaft zu sichern.

7.1 Dateien öffnen und schließen

Der erste Schritt beim Arbeiten mit einer Datei besteht darin, diese Datei zu öffnen. Diese Aufgabe wird mit der Bibliotheksfunktion *fopen()* erledigt. Nach dem Öffnen kann der Inhalt bearbeitet werden. Ist dies beendet, wird die Datei mit der Funktion *fclose()* wieder geschlossen. Die Funktionen *fclose()* und *fopen()* werden in der Headerdatei *stdio.h* deklariert.
Für die Arbeit mit Dateien, allgemeiner gesagt mit Datenströmen, gibt es in C den Datentyp FILE. Dieser Typ wird in der Headerdatei *stdio.h* definiert.

fopen() Die Funktionsbibliothek deklariert *fopen()* folgendermaßen:

```
FILE * fopen (char *Dateiname, char *Modus);
```

Diese Beschreibung wird folgendermaßen interpretiert:

- Die Funktion heißt *fopen*.
- Sie hat als Rückgabewert einen Pointer vom Typ FILE.
- Der erste Eingabeparameter erhält einen Zeiger auf den String des Dateinamens.
- Der zweite Eingabeparameter erhält einen Zeiger auf den String des Bearbeitungsmodus. Er legt fest, wie auf die Datei zugegriffen werden kann.

Als Bearbeitungsmodi sind gültig:

- "r" öffnet die Datei zum Lesen.
- "w" öffnet die Datei zum Schreiben. Wenn die Datei existiert, wird sie überschrieben, andernfalls neu angelegt.
- "a" öffnet die Datei zum Schreiben am Ende der Datei (Anhängen). Sie wird neu angelegt, falls sie noch nicht existiert.

- “rb“, “wb“, “ab“ öffnet die Datei wie oben für binäre Ein- und Ausgabe im Gegensatz zur zeichenorientierten Ein- und Ausgabe.
- Das zusätzliche Anhängen von „+“ gibt die Erlaubnis, eine Datei gleichzeitig zu manipulieren und zu lesen.

Schlägt das Öffnen einer Datei fehl, so liefert die Funktion *fopen()* den NULL-Pointer als Rückgabewert. Die Funktion *fclose()* liefert den Wert 0 zurück, falls eine Datei erfolgreich geschlossen werden konnte. Um sicher zu sein, dass eine Datei wirklich geschlossen ist, sollte man stets vor Beendigung des Programms alle mit *fopen()* geöffneten Datenströme auch wieder schließen. Die folgenden speziellen Datenströme sind vordefiniert und müssen nicht eigens geöffnet werden:

- *stdin*: Standardeingabe
- *stdout*: Standardausgabe
- *stderr*: Standardfehlerausgabe

Das Beispiel 7.1 öffnet die Dateien test1.txt und test2.txt und schließt sie anschließend wieder. Falls die Dateien bis dahin noch nicht im Verzeichnis waren, so existieren sie nach dem Programmende. Es sind allerdings Dateien ohne Inhalt.

Beispiel 7.1 (Öffnen und schließen von Dateien)

```
#include <stdio.h>

int main()
{
       FILE * datei_ptr;    /* Zeiger auf Datei (Datenstrom) */
       char dateiname[]="test1.txt";

        /* Öffnen der Datei "test1.txt" zum schreiben*/
       datei_ptr=fopen(dateiname,"w");
       fclose(datei_ptr);   /* Schließen der Datei "test1.txt" */

       dateiname[4]='2';

       /* Öffnen der Datei "test2.txt" zum schreiben*/
       datei_ptr=fopen(dateiname,"w");
       fclose(datei_ptr);   /* Schließen der Datei "test2.txt" */

       return 0;
}
```

7.2 Existenz einer Datei prüfen

Manchmal ist es für ein Programm nur wichtig, ob eine bestimmte Datei existiert. Da es für diese Aufgabe in der Standardbibliothek keine Funktion gibt, muss man sich mit einem Trick behelfen, den auch das folgende Programm benutzt.

Beispiel 7.2 (Existenz von Dateien prüfen)

```
#include <stdio.h>

int main()
{
       FILE * datei_ptr;    /* Zeiger auf Datei (Datenstrom) */
       char dateiname[]="test1.txt";

        /* Öffnen der Datei "test1.txt" zum lesen*/
       datei_ptr=fopen(dateiname,"r");

       if (datei_ptr==NULL) /* Fehler beim Öffnen ? */
       {
              printf("Die␣Datei␣%s␣existiert␣nicht\n",dateiname);
       }
       else
       {
              printf("Die␣Datei␣%s␣wurde␣erfolgreich␣geöffnet\n",dateiname);
              fclose(datei_ptr);   /* Schließen der Datei "test1.txt" */
       }
       return 0;
}
```

Die zu überprüfende Datei „test1.txt“ wird zum Lesen geöffnet. Schlägt der Versuch fehl, so liefert die Funktion *fopen()* den NULL-Zeiger zurück.

7.3 Zeichenorientierte Ein- und Ausgabe

Die folgenden in *stdio.h* deklarierten Funktionen dienen dem zeichen- bzw. string-orientierten Austausch von Daten eines Programms mit Dateien.

fprintf

Deklaration: int fprintf (FILE * Dateiname, const char* Formatstring,...);

Beschreibung: Verhält sich wie *printf()*, mit dem Unterschied, dass die einzelnen Zeichen des Strings nicht auf den Bildschirm, sondern in den Datenstrom (Datei) geschrieben werden. Es gilt : *printf(. . .) = fprintf(stdout,. . .)*;

fscanf

Deklaration: int fscanf (FILE * Dateiname, const char* Formatstring,...);

Beschreibung: Verhält sich wie *scanf()*, mit dem Unterschied, dass die einzelnen Zeichen des Strings nicht vom Bildschirm, sondern aus dem Datenstrom (Datei) eingelesen werden. Es gilt : *scanf(. . .)=fscanf(stdin,. . .)*;

fputc()

Deklaration: int fputc (int zeichen, FILE * Dateiname);

Beschreibung: Schreibt ein Zeichen in den Datenstrom (Datei) und liefert bei Erfolg das Zeichen zurück.

fgetc()

Deklaration: int fgetc (FILE * Dateiname);

Beschreibung: Liest ein Zeichen aus dem Datenstrom (Datei) und liefert bei Erfolg das Zeichen zurück. Bei erneutem Aufruf wird das nächste Zeichen von dort eingelesen.

Beispiel 7.3 (Lesen und Speichern von Matrizen)

Programmbeschreibung
Das Beispiel zeigt, wie man Matrizen in Dateien schreibt und sie wieder ausliest. Die Dimensionen der Matrizen werden durch Präprozessorvariablen realisiert. Der Name der zu betrachtenden Datei (hier: matrix.txt) wird ebenfalls durch eine Präprozessordirektive definiert.

(i) ***Hauptprogramm***
Das Hauptprogramm bietet dem Benutzer ein Menu mit fünf Auswahlmöglichkeiten an:

1 - Der Benutzer soll eine Matrix über die Tastatur initialisieren.

2 - Die aktuell im Speicher vorhandene Matrix wird auf dem Bildschirm ausgegeben.

3 - Die Datei matrix.txt wird ausgelesen, in den Speicher der aktuellen Matrix geschrieben und auf dem Bildschirm ausgegeben. Existiert die Datei matrix.txt nicht, so erscheint auf dem Bildschirm die Meldung: Datei existiert nicht.

4 - Die aktuell im Speicher vorhandene Matrix wird in die Datei matrix.txt geschrieben. Zeilen werden dabei durch das Zeichen "\n" markiert.

sonst - Abbruch.

(ii) ***newline (int N)***
Die Funktion newline erzeugt rekursiv Zeilenumbrüche. Die Anzahl der Zeilenumbrüche wird der Funktion als Parameter übergeben.

(iii) ***matrix_eingeben (float **A, int zeilen, int spalten)***
Initialisierung der Matrix durch den Benutzer. Anstatt des scanf()-Befehls wird die Kombination fgets() und sscanf() benutzt (siehe Kapitel **??***).*

(iv) ***matrix_ausgeben (float **A, int zeilen, int spalten)***
Ausgabe der Matrix.

(v) ***matrix_speichern (float **A, char *dateiname, int zeilen, int spalten)***
Funktion öffnet Datei namens des Strings, auf den der Pointer dateiname zeigt, zum Schreiben. Schlägt der Versuch fehl, so erscheint die Ausgabe: Fehler beim Öffnen. Andernfalls werden die einzelnen Matrixeinträge in die geöffnete Datei geschrieben. Die Einträge werden innerhalb der Zeile durch ein Leerzeichen getrennt. Zeilen werden durch das Zeichen „\n“ markiert.

(vi) ***matrix_laden(float **A, char *dateiname, int zeilen, int spalten)***
Funktion öffnet Datei namens des Strings, auf den der Pointer dateiname zeigt, zum Lesen. Schlägt der Versuch fehl, so erscheint die Ausgabe: Datei existiert nicht. Die Matrixeinträge werden nacheinander aus der Datei gelesen und im reservierten Speicher der Matrix abgelegt. (Auf die Parameterübergabe der Dimension der Matrix könnte in dieser Funktion verzichtet werden, da die Zeilen bzw. Matrixeinträge durch „\n“ bzw. durch ein Leerzeichen bei der Speicherung getrennt wurden. Zur Reduzierung des Quelltextes wurde dies jedoch nicht berücksichigt.)

```
#include <stdio.h>
#include <malloc.h>
#define M 3 /* Zeilen der Matrix */
#define N 3 /* Spalten der Matrix */
#define MATRIXDATEI "matrix.txt" /* Dateiname */

/***** Deklaration der Funktionen ************************/
void newline(int);
void matrix_eingeben(float **A, int zeilen, int spalten);
void matrix_ausgeben(float **A, int zeilen, int spalten);
void matrix_speichern(float **A, char *dateiname, int zeilen, int spalten);
void matrix_laden(float **A,char *, int, int);
      /* Variablennamen koennen, muessen aber nicht in der Deklaration
       * angegeben sein
       */

/***** Hauptprogramm *************************************/
int main()
{
      float ** A=NULL; /* Doppelpointer für Matrix */
      int zeilen = M; /* Anzahl Zeilen der Matrix */
      int spalten = N; /* Anzahl Spalten der Matrix */
      char menu;                  /* Menuauswahl */
      char dateiname[]=MATRIXDATEI; /* Dateiname */
      char tempstring[20]; /* Zum Einlesen vom Bildschirm */
      int i;
```

```
        /***** Allokalisiert Speicher für Matrix *****/
        A=(float **) malloc(M*sizeof(float));
        for (i=0;i<M;i++)
        {
                A[i]=(float *) malloc(N*sizeof(float));
        }

        /***** Eingabe Menu ************************/
        do
        {
                printf("1 - Matrix über Bildschirm eingeben\n");
                printf("2 - Matrix auf Bildschirm ausgeben\n");
                printf("3 - Matrix aus Datei laden \n");
                printf("4 - Matrix in Datei speichern \n");
                printf("sonst - Abbruch: ");
                fgets(tempstring,sizeof(tempstring),stdin);
                sscanf(tempstring,"%c",&menu);
                switch (menu)
                {
                        case '1': {matrix_eingeben(A,zeilen,spalten);break;}
                        case '2': {matrix_ausgeben(A,zeilen,spalten);break;}
                        case '3': {matrix_laden(A,dateiname,zeilen,spalten);
                                        break;}
                        case '4': {matrix_speichern(A,dateiname,zeilen,spalten);
                                        break;}
                        default: {printf("Wiedersehn\n");break;}
                }
        }
        while (menu >= '1' && menu <= '4');
        return 0;
}

/***** Definition der Funktionen ************************/


/***** Rekursive Definition einer neuen Zeile **/
void newline(int anzahl)
{
        if (anzahl > 0)
        {
                printf("\n");
                newline(anzahl-1);
        }
}

/***** Matrix vom Bildschirm einlesen **********/
```

```
void matrix_eingeben(float **A, int zeilen, int spalten)
{
	int i=0, j=0;
	char tempstring[20]="";
	newline(3);
	for (i=0;i<zeilen;i++)
	{
		for (j=0;j < spalten;j++)
		{
			printf("Wert für A[%i,%i] : ",i+1,j+1);
			fgets(tempstring,sizeof(tempstring),stdin);
			sscanf(tempstring,"%f",&A[i][j]);
		}
	}
	newline(3);
}

/***** Matrix auf dem Bildschirm ausgeben ******/
void matrix_ausgeben(float **A, int zeilen, int spalten)
{
	int i=0, j=0;
	newline(1);
	for (i=0;i<zeilen;i++)
	{
		for (j=0;j < spalten;j++)
		{
			printf("%f ",A[i][j]);
		}
		newline(1);
	}
	newline(1);
}

/***** Matrix in eine Datei schreiben **********/
void matrix_speichern(float **A, char *dateiname, int zeilen, int spalten)
{
	int i=0, j=0;
	FILE * datei_ptr;
	newline(3);

	datei_ptr=fopen(dateiname,"w");
		/* zum Schreiben öffnen */
	if (datei_ptr == NULL)
	{
		printf("Fehler beim Öffnen\n");
		fclose(datei_ptr);
```

```
        }
        else
        {
                for (i=0;i<zeilen;i++)
                {
                        for (j=0;j < spalten;j++)
                        {
                                fprintf(datei_ptr,"%f ",A[i][j]);
                        }
                        fprintf(datei_ptr,"\n");
                }
                fclose(datei_ptr);
                        /* sind mit Speichern fertig */
                printf("Matrix \n");
                matrix_ausgeben(A,zeilen,spalten);
                printf("in Datei %s geschrieben\n", dateiname);
                newline(2);
        }
}

/***** Matrix aus einer Datei lesen ************/
void matrix_laden(float **A ,char *dateiname,int zeilen,int spalten)
{
        int i=0, j=0;
        FILE * datei_ptr;
        char tempchar = ' ';

        newline(3);
        datei_ptr=fopen(dateiname,"r");
                /* zum Lesen öffnen */
        if (datei_ptr == NULL)
        {
                printf("Datei existiert nicht\n");
                newline(2);
        }
        else
        {
                for (i=0;i<zeilen;i++)
                {
                        for (j=0;j < spalten;j++)
                        {
                                fscanf(datei_ptr,"%f",&A[i][j]);
                        }
                        fscanf(datei_ptr,"%c",&tempchar);
                }
                fclose(datei_ptr);
```

```
                    /* sind mit Einlesen fertig */
                printf("Matrix \n");
                matrix_ausgeben(A,zeilen,spalten);
                printf("aus Datei %s eingelesen\n", dateiname);
                newline(2);
        }
}
```

Das Beispiel kann so modifiziert werden, dass die Dimension und der Dateiname durch den Benutzer eingegeben werden kann. Nach Initialisierung und Speichern kann die Datei matrix.txt mit dem Editor betrachtet werden. Ihr Inhalt hat bis auf die Werte die Gestalt:

```
4.000000 4.000000 3.000000
3.000000 2.000000 2.000000
5.000000 4.000000 1.000000
```

Die Zeilenumbrüche und Leerzeichen sieht man nur indirekt auf dem Bildschirm, sie stehen aber selbstverständlich auch in der Datei. Zur Speicherung eines Zeichens benötigt der Rechner ein Byte auf der Festplatte. Für obiges Beispiel benötigt der Computer 84 Byte (Jede Zahl wird durch 8 Zeichen dargestellt. Nach jeder Zahl folgt ein Leerzeichen. Jede Zeile wird mit „\n“ beendet).

7.4 Binäre Ein- und Ausgabe

Neben der zeichenorientierten Ein- und Ausgabe in Dateien gibt es noch die Möglichkeit, die Daten binär zu verarbeiten. Gründe, sich für diese Form der Datenausgabe zu entscheiden, sind:

- Bei der Ausgabe von Gleitkommazahlen als Zeichen hängt der benötigte Speicherplatz von der Anzahl der Stellen ab. Im Binärformat bestimmt der Datentyp die Größe.
- Die binäre Ein- und Ausgabe ist schneller.

Gründe dagegen sind:

- Binärformate unterscheiden sich von System zu System, so dass die entsprechenden Dateien evtl. in das jeweilige, systemabhängige Binärformat umzuwandeln sind. Die Daten sind nicht portabel.
- Manche System benötigen zur Kennung des Binärformats den Modusbezeichner *b*, andere Systeme wiederum nicht. Dies hat zur Folge, dass der Quelltext nicht portabel ist.
- Man kann Daten, die im Binärcode abgespeichert wurden, nicht mit einem Editor betrachten oder manipulieren.

Zur binären Ein- und Ausgabe verwendet man

fwrite()

Deklaration: size_t fwrite (const void *ptr, size_t groesse, size_t anzahl, FILE * Datei);

Beschreibung: Die Funktion schreibt von der Speicherposition, auf die *ptr* zeigt, *anzahl* Datenobjekte der Größe *groesse* in den Datenstrom *Datei*. Der Rückgabewert ist die Anzahl der erfolgreich geschriebenen Datenobjekte.

fread()

Deklaration: size_t fread (const void *ptr, size_t groesse, size_t anzahl, FILE * Datei);

Beschreibung: Die Funktion liest aus dem Datenstrom *Datei anzahl* Datenobjekte der Größe *groesse* und speichert sie (sequenziell) ab der Position, auf die *ptr* zeigt. Der Rückgabewert ist die Anzahl der erfolgreich gelesenen Datenobjekte.

Beispiel 7.4 (Binäre Ein- und Ausgabe)
Das Programm speichert die Matrix

$$A = \begin{pmatrix} 4.0 & 4.0 & 3.0 \\ 3.0 & 2.0 & 2.0 \\ 5.0 & 4.0 & 1.0 \end{pmatrix}$$

binär in die Datei matrix2.txt. Anschließend wird der Inhalt aus dieser Datei in die Matrix B geladen.

```
#include <stdio.h>
#define M 3
#define N 3
#define MATRIXDATEI "matrix2.txt"

int main()
{
        float A[M][N]={{4,4,3}, {3,2,2}, {5,4,1}};
        float B[M][N];

        void * ptr;
        FILE * datei_ptr;

        /***** Binäre Ausgabe ************************/

        /* Pointer auf die Adresse im Speicher setzen,
        *  ab der die Bytes in eine Datei geschrieben werden
        * sollen */
        ptr=&A[0][0];

        /* Datenstrom (Datei) öffnen */
        datei_ptr=fopen(MATRIXDATEI,"wb");
```

```
        /* Schreibt N*M*sizeof(float) Bytes ab
        * pointer-Adresse in den Datenstrom (Datei) */
        fwrite(ptr,sizeof(float),N*M,datei_ptr);

        /* Alternativ aber umständlich und langsam
        * for (i=0; i < M; i++)
        * {
        *      for (j=0; j < N; j++)
        *      {
        *              fwrite(ptr,sizeof(float),1,datei_ptr);
        *              ptr=ptr+sizeof(float);
        *      }
        * } */

        /* Datenstrom (Datei) schließen */
        fclose(datei_ptr);

        /***** Binäre Eingabe ************************/

        /* Datenstrom (Datei) öffnen */
        datei_ptr=fopen(MATRIXDATEI,"rb");

        /* Pointer auf die Adresse im Speicher setzen,
        *  ab der die Bytes aus der Datei in den Speicher
        *  geschrieben werden sollen */
        ptr=&B[0][0];

        /* Liest N*M*sizeof(float) Bytes aus der Datei
        *  und schreibt sie sequenziell an die pointer-Adresse */
        fread(ptr,sizeof(float),N*M,datei_ptr);

        /* Datenstrom (Datei) schließen */
        fclose(datei_ptr);

        return 0;
}
```

7.5 Dateien löschen/umbenennen

Die Anweisung, mit der Dateien gelöscht werden können, lautet *remove(pfadname)*. Diese Funktion liefert den Wert -1, wenn nicht gelöscht werden konnte, ansonsten den Wert 0.

Zum Umbennen von Verzeichnissen und Dateien und zum Verschieben von Dateien wird die Funktion *rename(pfadangabe)* benutzt. Liefert die Funktion einen Wert ungleich 0, dann ist ein Fehler aufgetreten. Als Parameter werden in der Klammer zuerst der alte, danach der neue

Name angegeben. Weichen die Verzeichnisnamen in den beiden Pfadangaben voneinander ab, dann wird die Datei verschoben. Verzeichnisse können nur umbenannt, nicht verschoben werden.

Beispiel 7.5 (Löschen und Umbenennen von Dateien)

```
#include <stdio.h>

int main()
{
        /***** Löschen der Datei matrix.txt ***********/

        /* Der Backslash bei der Angabe des Pfadnames * muss doppelt eingegeben
         * werden, da er sonst als Steuerzeichen interpretiert wird.
         * Der Backslash als Verzeichnistrenner ist Windows-spezifisch. Fuer
         * LINUX ist es ein normaler Slash '/', fuer Macintosh ein Doppelpunkt.
         */

        if (remove(".\\matrix.txt")==-1)
        {

        /* Ausgabe der Fehlermeldung; die Funktion perror() dient zur Ausgabe
         * von Fehlermeldungen auf dem Datenstrom stderr. Der Text in Klammern
         * wird um die interne Fehlermeldung ergänzt.
         * Die Ausgabe auf dem Bildschirm in diesem Fall lautet :
         * Fehler beim Löschen : No such file or directory
         */
                perror("Fehler␣beim␣Löschen␣");
        }
        else
        {
                printf("Datei␣matrix.txt␣gelöscht.\n");
        }
        /***** Umbenennen der Datei matrix2.txt **********/

        if (rename(".\\matrix2.txt","matrix.bin") != 0)
        {
                perror("Fehler␣beim␣Umbenennen␣");
        }
        else
        {
                printf("Datei␣matrix2.txt␣umbenannt␣zu␣matrix.bin.\n");
        }
        return 0;
}
```

7.6 Positionierung in einem Datenstrom

Nach dem Öffnen eines Datenstroms wird die aktuelle Position in dem *Datei-Pointer* abgelegt. Werden Zeichen oder Zahlen aus dem Datenstrom in den Speicher eingelesen, so erhöht sich die Positionsangabe entsprechend der eingelesen Bytes. Zur Ansteuerung einer bestimmten Stelle im Datenstrom enthält der Header *stdio.h* die Funktion *fseek()*.

fseek(datei-pointer,offset,ursprung)

Mit *offset* wird angegeben, um wieviele Bytes der *datei-pointer* sich vom *ursprung* entfernen soll. Für *ursprung* stehen drei Alternativen zur Auswahl:

1.) SEEK_SET: Anfang des Datenstroms

2.) SEEK_CUR: Aktuelle Position

3.) SEEK_END: Ende des Datenstroms (EOF)

Die aktuelle Position im Datebstrom kann mit der Funktion *ftell()* abgefragt werden. Das nächste Beispiel zeigt die Funktionsweise der beiden Funktionen.

Beispiel 7.6 (Positionierung im Datenstrom)

```
#include <stdio.h>
#define zeilen 3
#define spalten 3

int main()
{
       FILE * dateiptr;
       int i;
       double A[zeilen][spalten]={{1.0,2.0,3.0}, {4.0,5.0,6.0}, {7.0,8.0,9.0}};
       double ausgabe;
       void * ptr;

       /***** Speichern von A im Binärformat in die Datei matrix.bin ******/
       ptr=&A[0][0];
       dateiptr=fopen("matrix.bin","wb");
       fwrite(ptr,sizeof(A),1,dateiptr);
       fclose(dateiptr);

       /* Ausgabe der Diagonalen von A mit Angabe der aktuellen Position im
       * Datenstrom */
       dateiptr=fopen("matrix.bin","rb");
       ptr=&ausgabe;

       for (i=1;i<=spalten;i++)
       {
```

```
            printf("Position␣in␣Datei␣matrix.bin␣:␣%i␣\t",(int) ftell(dateiptr));
            /* Einlesen einer Zahl */
            fread(ptr,sizeof(double),1,dateiptr);
            printf("A[%i][%i]␣:␣%f\n",i,i,ausgabe);
            /* Sprung zum nächsten Diagonalelement */
            fseek(dateiptr,(spalten)*sizeof(double),SEEK_CUR);
      }
      fclose(dateiptr);
      return 0;
}
```

Das Programm erzeugt die Ausgabe:

```
Position in Datei matrix.bin : 0   A[1][1] : 1.000000
Position in Datei matrix.bin : 32  A[2][2] : 5.000000
Position in Datei matrix.bin : 64  A[3][3] : 9.000000
```

7.7 Dateiausgabe umlenken

Wie schon in Abschnitt 7.1 bemerkt, gibt es zwei stets geöffnete Standardausgaben, nämlich *stdout* und *stderr*. Normalerweise geben beide die Daten auf dem Bildschirm aus. Wie kann man nun beide Datenströme so trennen, dass sie auf separate Einheiten gelenkt werden, beispielsweise *stdout* weiterhin auf den Bildschirm und *stderr* in eine Protokolldatei?

Beispiel 7.7 (Umleiten des *stderr*-Stroms in die Datei error.log)

```
#include <stdio.h>

int main()
{
      /* Umleiten des Ausgabestroms stderr.
      * Anstatt der Ausgabe von stderr auf den Bildschirm, wird bei der
      * Ausgabe alles in die Datei error.log geschrieben (angehängt) */
      freopen("error.log","a",stderr);

      if (remove(".\\matrix.bin")==-1)
      {
            /* Tritt ein Fehler beim Löschen auf,
            * so wird der Text "Fehler ..." an die
            * Datei error.log angehängt */
            fprintf(stderr,"Fehler␣beim␣Löschen␣der␣Datei␣matrix.bin.\n");
      }
      else
      {
            /* Wird die Datei ordentlich gelöscht, so
```

```
            * folgt die Ausgabe "Datei ..." auf dem
            * Bildschirm */
            fprintf(stdout,"Datei matrix.bin erfolgreich gelöscht.\n");
            /* Aquivalent wäre printf("Datei ..."); */
        }
        return 0;
}
```

Die Funktion *freopen()* hat als Rückgabewert wieder einen Dateipointer. Die exakte Deklaration ist:

FILE * (const char * dateiname, const char * modus, FILE * datei-pointer);

Die Funktion verbindet einen gültigen Dateipointer mit einem neuen Datenstrom (Datei) und liefert als Rückgabewert einen Dateizeiger auf diesen Datenstrom. Die drei Parameter geben der Reihe nach den neuen Datenstrom, den neuen Bearbeitungsmodus und einen Dateipointer an, der zu einer geöfffneten Datei gehören muss. Hier wird also der stets geöffnete Datenstrom *stderr* umgelenkt auf die Datei *error.log* im Modus *append*. Die Rückgabe ist ebenfalls der Dateipointer und kann auch alternativ verwendet werden.

8 Mehrdateiprojekte

Die bisher gezeigten Programmbeispiele waren im Umfang noch recht überschaubar, was natürlich auf ihre eingeschränkte Funktionalität zurückzuführen ist. Größere Softwareprojekte werden im Allgemeinen von mehreren Entwicklern bearbeitet, was eine Aufteilung des Problems erfordert.
Wäre man in dieser Situation gezwungen, das Programm in einer einzigen Quelldatei zu halten, so wären chaotische Zustände im Quellcode praktisch unvermeidbar. Deshalb ist das Aufteilen der Aufgaben in C durch Mehrdateiprojekte ermöglicht.

1.) Präprozessordirektiven sowie die Deklaration eigener Strukturen, Unions, Typen und Subroutinen (Funktionen) werden in Headerdateien (Endung .h) zusammengefasst.

2.) Das Hauptprogramm wird in einer speziellen Datei, zumeist *main.c* gehalten.

3.) Die einzelnen Funktionen werden jeweils in entsprechenden Dateien gehalten (Endung .c). Inhaltlich zusammengehörige Subroutinen können gemeinsam in der jeweiligen Datei stehen.

4.) Die Übersetzung der einzelnen Dateien zu Modulen (Endung .o) ist zu koordinieren. Hier wird auf den in Abschnitt 1.6 vorgestellten mehrstufigen Kompiliervorgang zurückgegriffen.

5.) Optimierte und getestete Module können schließlich zu eigenen Programmbibliotheken (Endung .a) zusammengefaßt werden. Andere Projekte können dann auf die fertigen Module während des Linkvorgangs zugreifen.

In diesem Kapitel wird ein solches Projekt (immer noch in überschaubarem Umfang) illustriert. Dazu sollen Routinen zum Erzeugen, Laden und Speichern von beliebigen $m \times n$ Matrizen implementiert. Zusätzlich soll die Matrix-Multiplikation und das Anzeigen des Inhalts einer Matrix auf dem Bildschirm zur Verfügung stehen.
Begonnen wird mit der Strukturierung der Funktionalitäten:

1.) Matrizen werden in einer Struktur *matrix* gespeichert. Die Struktur enthält zwei int-Variablen für die Dimension der Matrix und einen *double* **-Zeiger für den Zugriff auf die einzelnen Elemente der Matrix:

```
struct matrix
{
  int anz_zeilen;
  int anz_spalten;
  double ** matrix;
};
```

2.) Laden einer Matrix von der Festplatte:
Es wird vorausgesetzt, dass die Matrizen im Zeichenformat abgespeichert wurden. Vor den eigentlichen Matrixeinträgen besteht die erste Zeile der Datei aus zwei Integerwerten, die die Anzahl der Zeilen bzw. Spalten angibt. Jede Zahl wird durch ein Leerzeichen und jede Zeile wird durch einen Zeilenumbruch \n getrennt. Die Funktion soll als Rückgabewert eine Struktur matrix haben:

struct matrix matrix_laden (char * dateiname);

3.) Speichern einer Matrix auf der Festplatte:
Eine Variable von Typ *struct matrix* soll in dem unter 2.) angegeben Format gespeichert werden. Formal soll die Subroutine folgende Eingabeparameter besitzen:

void matrix_speichern(char * dateiname, struct matrix A);

4.) Anzeigen einer Matrix auf dem Bildschirm:
Diese Funktionen gibt eine Variable der Struktur *matrix* auf dem Bildschirm aus:

void matrix_anzeigen(struct matrix A);

5.) Matrixmultiplikation:
Die Funktion hat als Rückgabewert das Matrixprodukt zweier Matrizen:

struct matrix matrix_multiplizieren(struct matrix A, struct matrix B);

6.) Erzeugen einer Matrix (Speicherreservierung):
Unter Verwendung der Angabe der Dimensionen der Matrix soll entsprechend Speicher für eine Struktur *matrix* reserviert werden.

struct matrix matrix_allokieren(int anz_zeilen,int anz_spalten);

Dieses Anforderungsprofil führt auf die folgende eigene Headerdatei, die *matrix.h* genannt wird.

```
/* Deklaration der Struktur matrix */

struct matrix
{
       int anz_zeilen;
       int anz_spalten;
       double ** matrix;
};

/* Deklaration der einzelnen Subroutinen */

struct matrix matrix_allokieren(int anz_zeilen,int anz_spalten);
struct matrix matrix_laden(char * dateiname);
```

```
struct matrix matrix_multiplizieren(struct matrix A,struct matrix B);
void matrix_speichern(char * dateiname,struct matrix A);
void matrix_anzeigen(struct matrix A);
```

8.1 Verteilung des Codes auf Dateien

Die Funktionalitäten sind nun auf die einzelnen Subroutinen verteilt und deren Deklarationen in der Headerdatei *matrix.h* zusammengefaßt. Inhaltlich zusammengehörige Subroutinen (wie zum Beispiel *matrix_laden()* und *matrix_speichern()*) können in einer Quelltextdatei zusammengefaßt werden. Da dieses Beispiel noch in einem überschaubarem Umfang vorliegt, wird für jede Funktion jedoch eine eigene Quelldatei erzeugt. Schließlich muss noch der Quellcode des Hauptprogramms in der Datei *main.c* abgelegt werden. Das Mehrdateiprojekt besteht mit dem Header aus den sieben Dateien:

main.c
matrix_speichern.c
matrix_laden.c
matrix_allokieren.c
matrix_anzeigen.c
matrix_multiplizieren.c
matrix.h

Achtung!!! Jede einzelne Datei ist beim späteren Übersetzen unabhängig von den anderen. Speziell wissen, die einzelnen Quelldateien nicht, welche Headerdateien von *main.c* eingebunden werden. Daraus folgt, dass jede einzelne Quelldatei diejenigen Headerdateien mit *#include* einfügen muss, die für die jeweils verwendeten Bibliotheksroutinen notwendig sind.

Der Quellcode der einzelnen Subroutinen könnte wie unten angegeben lauten. (Auf Sicherheitsabfragen bzgl. Speicheranforderungen und Dateioperationen wurde aus Platzgründen verzichtet, ebenso auf die Überprüfung der Grössen der Matrizen.)

1.) **matrix_speichern.c**

```
#include <stdio.h> /* fopen, fclose, fprintf */
#include "matrix.h" /* struct matrix + Deklaration der Funktion */

void matrix_speichern(char * dateiname,struct matrix A)
{
       FILE *file_ptr;
       int i,j;
       file_ptr=fopen(dateiname,"w");
       fprintf(file_ptr,"%i␣%i␣\n",A.anz_zeilen,A.anz_spalten);
       for (i=0;i < A.anz_zeilen; i++)
       {
```

```
                for (j=0;j < A.anz_spalten; j++)
                {
                        fprintf(file_ptr,"%f ",A.matrix[i][j]);
                }
                fprintf(file_ptr,"\n");
        }
        fclose(file_ptr);
}
```

2.) **matrix_laden.c**

```
#include <stdio.h> /* fopen, fclose, fscanf */
#include "matrix.h" /* struct matrix + Deklaration der Funktion */


struct matrix matrix_laden(char * dateiname)
{
        FILE *file_ptr;
        struct matrix A;
        int anz_zeilen, anz_spalten;
        int i,j;

        file_ptr=fopen(dateiname,"r");
        fscanf(file_ptr,"%i %i",&anz_zeilen,&anz_spalten);
        A=matrix_allokieren(anz_zeilen,anz_spalten);
        for (i=0;i < A.anz_zeilen ; i++)
        {
                for (j=0; j < A.anz_spalten; j ++)
                {
                        fscanf(file_ptr,"%lf",&A.matrix[i][j]);
                }
        }
        fclose(file_ptr);
        return A;
}
```

3.) **matrix_anzeigen.c**

```
#include <stdio.h> /* printf */
#include "matrix.h" /* struct matrix + Deklaration der Funktion */

void matrix_anzeigen(struct matrix A)
{
        int i,j;
        for (i=0;i< A.anz_zeilen;i++)
        {
                for (j=0;j < A.anz_spalten;j++)
```

```
                {
                        printf("%f␣",A.matrix[i][j]);
                }
                printf("\n");
        }
}
```

4.) **matrix_allokieren.c**

```
#include "matrix.h" /* struct matrix + Deklaration der Funktion */
#include <malloc.h> /* malloc */

struct matrix matrix_allokieren(int anz_zeilen,int anz_spalten)
{
        struct matrix a;
        int i;

        a.matrix=(double **) malloc(anz_zeilen*sizeof(double *));
        for (i=0;i<anz_zeilen;i++)
        {
                a.matrix[i]=(double *) malloc(anz_spalten*sizeof(double));
        }
        a.anz_zeilen=anz_zeilen;
        a.anz_spalten=anz_spalten;
        return a;
}
```

5.) **matrix_multiplizieren.c**

```
#include "matrix.h" /* struct matrix + Deklaration der Funktion */

struct matrix matrix_multiplizieren(struct matrix A,struct matrix B)
{
        struct matrix C;
        int i,j,k;
        C=matrix_allokieren(A.anz_zeilen,B.anz_spalten);

        for (i=0;i< A.anz_zeilen; i++)
        {
                for (j=0;j < B.anz_spalten;j++)
                {
                        C.matrix[i][j]=0;
                        for (k=0;k < A.anz_spalten;k++)
                        {
                                C.matrix[i][j]=C.matrix[i][j]
                                        +A.matrix[i][k]*B.matrix[k][j];
                        }
```

```
            }
        }
        return C;
}
```

6.) **main.c**

```
#include "matrix.h" /* structmatrix */

int main()
{
        struct matrix A,B,C;

        A=matrix_laden("matrixA.dat");
        B=matrix_laden("matrixB.dat");
        C=matrix_multiplizieren(A,B);
        matrix_anzeigen(C);
        matrix_speichern("matrixC.dat",C);

        return 0;
}
```

8.2 Manuelles Kompilieren eines Mehrdateiprojektes

Mehrdateiprojekte werden in mindestens zwei Stufen zu einem ausführbaren Programm übersetzt:

1.) Die einzelnen Quelldateien werden mit der Option -c kompiliert. Es entstehen die Objektdateien, in diesem Zusammenhang auch *Module* genannt:

    gcc -c “Liste der .c-Dateien“

    Aus jeder Quelltextdatei entsteht eine gleichnamige Datei mit der Endung .o.

2.) Die Objektdateien (Module) werden zu einer ausführbaren Datei gelinkt. An dieser Stelle werden eventuell benötigte Bibliotheken (z.B. die Mathebibliothek) eingebunden:

    gcc -o “Name des Programms“ “Liste der .o Dateien“ -lBibliothek

In dem einfachen Beispiel bedeutet dies:

gcc -c main.c matrix_speichern.c matrix_laden.c matrix_anzeigen.c
gcc -c matrix_allokieren.c matrix_multiplizieren.c

Mit diesen Anweisungen entstehen die Objektdateien

main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o
matrix_allokieren.o matrix_multiplizieren.o

Das Linken der Objektdateien zu einem ausführbarem Programm erfolgt durch:

gcc -o Matrix main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o matrix_allokieren.o matrix_multiplizieren.o

An dieser Stelle zeigt sich der Sinn des Aufteilens von Quelltexten auf mehrere Dateien sehr deutlich: Angenommen, man möchte bei der Funktion *matrix_anzeigen()* ein anderes Format für die Matrixeinträge. Dann editiert man die entsprechenden Datei *matrix_anzeigen.c*, übersetzt nur diese,

gcc -c matrix_anzeigen.c

und bindet die aktualisierte Objektdatei matrix_anzeigen.o mit den bereits vom letzten Compileraufruf vorhandenen anderen Objektdateien zum Programm zusammen:

gcc -o Matrix main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o matrix_allokieren.o matrix_multiplizieren.o

Man kann also Kompiliervorgänge durch sorgfältige Aufteilung auf das notwendige Maß reduzieren. Trotzdem ist es nicht immer ergonomisch, nur eine Funktion pro Datei zu halten.
Macht man manuell von dieser Möglichkeit Gebrauch, so ist bei hinreichend großen Projekten die Gefahr gegeben, dass man etwa ein Modul vergisst und sich die Fehlermeldungen nicht erklären kann.
Wichtiger ist, dass aus der Aktualisierung einer Quelldatei bzw. eines Moduls die Notwendigkeit folgen kann (und im Allgemeinen auch folgt !!), dass ein anderes Modul neu erzeugt werden muss, weil es auf das aktualisierte Modul zugreift.
In großen Projekten werden diese Abhängigkeiten so komplex, dass man nur äußerst schwer den Überblick bewahren kann (der Quellcode des Linuxkernels z. B. besteht aus mehr als 17.000 Dateien).

8.3 Automatisiertes Kompilieren mit make

Um die am Ende des letzten Abschnitts genannten Probleme meistern zu können, gibt es das Programm **make**, das unter Berücksichtigung der Abhängigkeiten der Programmmodule untereinander die notwendigen Übersetzungsvorgänge vornimmt. Die Abhängigkeiten werden dem Programm in einer Steuerdatei, die standardmäßig *Makefile* oder *makefile* heißt, mitgeteilt.

Das Konzept

- Der Kompiliervorgang setzt sich aus der Erzeugung von Zielen (Targets) zusammen. Das ausführbare Programm z.B. ist ein solches Target.
- Zu jedem Target wird im *Makefile* festgehalten, von welchen anderen Targets es abhängt.
- Anhand des Datums der letzten Dateiänderung stellt das Programm *make* fest, ob ein Target älter ist als jene, von denen es abhängt. Ist dies der Fall, so muss das Target neu erzeugt werden. Im *Makefile* muss festgehalten werden, wie diese Erzeugung zu geschehen hat.
- Rekursiv stellt *make* durch Auswertung des *Makefiles* die Abhängigkeiten fest. Auf der untersten Ebene befinden sich z.B. die Quell- und Headerdateien, die ja zumeist durch Editieren aktualisiert werden.

Aufbau von Makefiles

- Ein Target wird den Objekten, von denen es abhängt, durch einen Doppelpunkt : getrennt.
- Die Liste der Kommandos zur Erzeugung des Targets folgt in einer neuen Zeile, **die stets mit einem Tabulatorzeichen beginnen muss.**
- Ein Target kann mehrmals auftreten.
- Ruft man an der Kommandozeile *make* auf, so wird das erste Target im *Makefile* erzeugt. Möchte man gezielt ein anderes Target erzeugen, so verwendet man

  make Name des Targets

Für das Beispielprojekt hat das Makefile die Darstellung:

Beispiel 8.1 (Makefile 1)

```
Matrix : main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o
         matrix_multiplizieren.o matrix_allokieren.o
	gcc -o Matrix main.o matrix_speichern.o matrix_laden.o\
	   matrix_anzeigen.o matrix_multiplizieren.o matrix_allokieren.o

main.o : main.c matrix.h
	gcc -c main.c -Wall

matrix_speichern.o : matrix_speichern.c matrix.h
	gcc -c matrix_speichern.c -Wall

matrix_laden.o : matrix_laden.c matrix.h
	gcc -c matrix_laden.c -Wall
```

```
matrix_anzeigen.o : matrix_anzeigen.c matrix.h
	gcc -c matrix_anzeigen.c -Wall

matrix_multiplizieren.o : matrix_multiplizieren.c matrix.h
	gcc -c matrix_multiplizieren.c -Wall

matrix_allokieren.o : matrix_allokieren.c matrix.h
	gcc -c matrix_allokieren.c -Wall
```

Makros mit make

Makros dienen der Abkürzung und Zusammenfassung von mehreren Objekten sowie der Unterstützung bei der plattformunabhängigen Programmierung. Muss man z.B. eine Liste von Objektdateien mehrmals im Makefile verwenden, so leistet dies

> Objekte = main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o matrix_multiplizieren.o matrix_allokieren.o

und den so zugewiesenen Inhalt des Makros *Objekte* liest man durch

> $(Objekte)

Das obige *Makefile* lässt sich mit Hilfe von Makros vereinfachen:

Beispiel 8.2 (Makefile 2)

```
PROGRAMMNAME = Matrix
OBJEKTE = main.o matrix_speichern.o matrix_laden.o matrix_anzeigen.o
          matrix_multiplizieren.o matrix_allokieren.o

$(PROGRAMMNAME) : $(OBJEKTE)
	gcc -o $(PROGRAMMNAME) $(OBJEKTE)

main.o : main.c
	gcc -c main.c -Wall matrix.h

matrix_speichern.o : matrix_speichern.c matrix.h
	gcc -c matrix_speichern.c -Wall

matrix_laden.o : matrix_laden.c matrix.h
	gcc -c matrix_laden.c -Wall

matrix_anzeigen.o : matrix_anzeigen.c matrix.h
	gcc -c matrix_anzeigen.c -Wall

matrix_multiplizieren.o : matrix_multiplizieren.c matrix.h
```

```
	gcc -c matrix_multiplizieren.c -Wall

matrix_allokieren.o : matrix_allokieren.c matrix.h
	gcc -c matrix_allokieren.c -Wall
```

Der Aufruf *make* veranlasst also die Erzeugung des ersten Targets, in diesem Fall ist es das ausführbare Programm *Matrix*.

8.4 Eigene Bibliotheken

Sind die einzelnen Objekte (Module) auf ihre Zuverlässigkeit und Geschwindigkeit hin ausreichend getestet, so kann man erwägen, sie in einer eigenen Programmbibliothek zusammenzufassen. Wie bei der C-Standardbibliothek oder der Mathematikbibliothek ist dann bei der Implementierung von neuen Programmen nur darauf zu achten, dass die zugehörige Headerdatei mit den Deklarationen an den entsprechenden Stellen im Quelltext auftaucht und die Bibliothek beim Linken eingebunden wird. Bibliotheken entstehen aus den Objektdateien, die mit dem *ar*-Kommando zusammengefügt werden.
Zur Illustration werden die Funktionen in der Bibliothek *libmatrix.a* zusammengefasst. Dies geschieht mit dem Aufruf

> ar -r libmatrix.a matrix_speichern.o matrix_laden.o matrix_anzeigen.o matrix_multiplizieren.o matrix_allokieren.o

und beim späteren Linken wird die Bibliothek einfach mit angegeben:

> gcc -o Matrix main.o libmatrix.a

Auch diese Operation lässt sich mit *make* automatisieren. Das entsprechende *Makefile* sieht zum Beispiel folgendermaßen aus:

Beispiel 8.3 (Makefile 3)

```
PROGRAMMNAME = Matrix
BIBLIOTHEK = ./libmatrix.a
VERZ = ./
BIBOBJEKTE = matrix_speichern.o matrix_laden.o matrix_anzeigen.o
         matrix_multiplizieren.o matrix_allokieren.o

$(PROGRAMMNAME) : main.o $(BIBLIOTHEK) $(VERZ)/matrix.h
	gcc -o $(PROGRAMMNAME) main.o $(BIBLIOTHEK)

main.o : $(VERZ)/main.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/main.c -Wall

$(BIBLIOTHEK) : $(BIBOBJEKTE)
```

```
	ar -r $(BIBLIOTHEK) $(BIBOBJEKTE)

matrix_speichern.o : $(VERZ)/matrix_speichern.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/matrix_speichern.c -Wall

matrix_laden.o : $(VERZ)/matrix_laden.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/matrix_laden.c -Wall

matrix_anzeigen.o : $(VERZ)/matrix_anzeigen.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/matrix_anzeigen.c -Wall

matrix_multiplizieren.o : $(VERZ)/matrix_multiplizieren.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/matrix_multiplizieren.c -Wall

matrix_allokieren.o : $(VERZ)/matrix_allokieren.c $(VERZ)/matrix.h
	gcc -c $(VERZ)/matrix_allokieren.c -Wall
```

Wird die Bibliothek anderswo untergebracht, um anderen Programmierprojekten den Zugriff zu erleichtern und handelt es sich (ähnlich wie bei den Headerdateien) nicht um ein Verzeichnis im Standardsuchpfad für C-Bibliotheken, so muss das betreffende Verzeichnis beim Linken mit der Option -L angegeben werden. Die Bibliothek wird dann mit der Option -l übergeben, wobei das Präfix lib und das Suffix .a entfallen. Da das aktuelle Verzeichnis mit . bezeichnet wird, sind folgende Formulierungen äquivalent

gcc -o Matrix main.o -L. -lmatrix
gcc -o Matrix main.o libmatrix.a

A

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

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

wobei $2 \leq B \in \mathbb{N}, N \in \mathbb{Z}$ und $x_n \in \{0, 1, \ldots, B-1\}$ für alle $n \in \mathbb{N}$ (B-adische Entwicklung von x). Diese ist eindeutig, wenn gilt:

- $x_{-1} \neq 0$, falls $x \neq 0$,
- und es existiert ein $n \in \mathbb{N}$ mit $x_{-n} \neq B-1$.

Eine Maschinenzahl dagegen hat die Form

$$\tilde{x} = sgn(x)B^N \sum_{n=1}^{l} x_{-n}B^{-n}$$

wobei die feste Größe $l \in \mathbb{N}$ die Mantissenlänge ist. Als Mantisse bezeichnet man den Ausdruck

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

Hieraus folgt sofort, dass irrationale Zahlen überhaupt nicht und von den rationalen Zahlen nur ein Teil im Rechner dargestellt werden. Man unterscheidet zwei Arten der Darstellung:

1. Festkommadarstellung
   Sowohl die Anzahl N der zur Darstellung verfügbaren Ziffern, als auch die Anzahl N_1 der Vorkommastellen ist fixiert. Die Anzahl der maximal möglichen Nachkommastellen N_2 erfüllt notwendigerweise

$$N = N_1 + N_2.$$

2. Gleitkommadarstellung
   In (A.2) ist die Mantissenlänge l fixiert und der Exponent N ist begrenzt durch

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

   Alle Maschinenzahlen liegen dabei vom Betrag her im Intervall $(B^{N_- - 1}, B^{N_+})$. Zur Normalisierung der Darstellung verlangt man, dass $x_{-1} \neq 0$, wenn $x \neq 0$. Zahlen, die kleiner als $B^{N_- - 1}$ sind, werden vom Computer als 0 angesehen. Zahlen, die größer als B^{N_+} sind, können nicht verarbeitet werden (Exponentenüberlauf).

Die Darstellung von irrationalen Zahlen erfordert eine Projektion auf die Menge der Maschinenzahlen, die so genannte Rundung. Man unterscheidet vier Rundungsarten:

1. Aufrundung:
   Zu $x \in \mathbb{R}$ wählt man die nächsthöhere Maschinenzahl $\tilde{x}$.

2. Abrundung:
   Zu $x \in \mathbb{R}$ wählt man die nächstniedrigere Maschinenzahl $\tilde{x}$.

3. Rundung (im engeren Sinne)
   Ausgehend von der Darstellung (A.1), setzt man

$$\tilde{x} := sgn(x) B^N \begin{cases} \sum_{n=1}^{l} x_{-n} B^{-n}, & \text{falls} x_{-(l+1)} < B/2 \\ \sum_{n=1}^{l} x_{-n} B^{-n} + B^{-l}, & \text{falls} x_{-(l+1)} \geq B/2 \end{cases}$$

4. Abschneiden
   Verwerfen aller x_{-n} mit $n > l$ in der Darstellung A.1.

Die Größe

$$|\tilde{x} - x|$$

heißt absoluter, die Größe

$$\frac{|\tilde{x} - x|}{|x|}$$

heißt relativer Rundungsfehler. Man wird erwarten, dass die Rundung im engeren Sinne die beste ist. In der Tat weist sie statistisch gesehen die geringsten Rundungsfehler auf.

Für die Rundung (im engeren Sinne) gilt:

a) $\tilde{x}$ ist wieder eine Maschinenzahl.

b) der absolute Fehler erfüllt

$$|\tilde{x} - x| \leq \frac{1}{2} B^{N-l}$$

c) Der relative Fehler erfüllt

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

Die Zahl eps wird auch als Maschinengenauigkeit bezeichnet.

A.2 IEEE Gleitkommazahlen

Die IEEE (Institute of Electrical and Electronic Engineers) ist eine internationale Organisation, die binäre Formate für Gleitkommazahlen festlegt. Diese Formate werden von den meisten (aber nicht allen) heutigen Computern benutzt. Die IEEE definiert zwei unterschiedliche Formate mit unterschiedlicher Präzision: Einzel- und Doppelpräzision (single and double precision). Single precision wird in C von float Variablen und double precision von double Variablen benutzt.
Intel's mathematischer Co-Prozessor benutzt eine dritte, höhere Präzision, die sogenannte erweiterte Präzision (extended precision). Alle Daten im Co-Prozessor werden mit dieser erweiterten Präzision behandelt. Werden die Daten aus dem Co-Prozessor im Speicher ausgelagert, so werden sie automatisch umgewandelt in single bzw. double precision. Die erweiterte Präzision benutzt ein etwas anderes Format als die IEEE float- bzw double-Formate und werden in diesem Abschnitt nicht diskutiert.

IEEE single precision

Single precision Gleitkommazahlen benutzen 32 Bits (4 Byte) um eine Zahl zu verschlüsseln:

Bit 1-23: Hier wird die Mantisse gespeichert, d.h. die Mantissenlänge l beträgt bei single precision 23. Im single precision Format wird zur Mantisse noch eins addiert

Beispiel A.1 (IEEE Mantisse)

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

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 precsion)
Welche Dezimalzahl wird duch den Binärcode $\underbrace{01000001}_{Byte\ 3}\ \underbrace{11011011}_{Byte\ 2}\ \underbrace{01000000}_{Byte\ 1}\ \underbrace{00000000}_{Byte\ 0}$ *dargestellt?*

$$\underbrace{0}_{\text{Zahl ist positiv}}\ \underbrace{10000011}_{N=131-127=4}\ \underbrace{10110110100000000000000}_{\text{Mantisse}}$$

Bit 32 Bit 31-24 *Bit 23-1*

$=1.101101101_2 * 2^N$
$=11011.01101_2$
$=2^4 + 2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3} + 2^{-5}$
$=27.406250$

Folgendes Beispielprogramm testet ob der Computer die float Zahl 27.040625 im IEEE single precision Format speichert.

Beispiel A.3

```
# 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  */
       /*                                                          */
```

```
        /* binär :    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 [Car].

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:

Beipiel (Auslöschnung):

Sei $B = 10$, $l = 3$. Sei $x = 0.9995$, $y = 0.9984$ und es soll im Computer die Differenz $x - y$ berechnet werden. Wie wirkt sich die Rundung auf das Resultat aus? Zunächst ist $\tilde{x} = 0.1 \cdot 10^1$ und $\tilde{y} = 0.998$. Daher ist

$$\tilde{x} - \tilde{y} = 0.2 \cdot 10^{-2},$$

das exakte Resultat ist

$$x - y = 0.11 \cdot 10^{-2}$$

und der relative Fehler ergibt sich zu 81.8% ! Dieser Effekt der Verstärkung des Rundungsfehlers tritt auf, wenn man fast identische Zahlen voneinander subtrahiert.

Eine Operation mit ähnlich instabilen Verhalten ist die Division x/y mit $x << y$. Addition zweier Zahlen mit gleichem Vorzeichen und Multiplikation sind jedoch gutartige Operationen.

A.4 Die O-Notation

Um eine objektive Grundlage für den Vergleich zweier Algorithmen zur Lösung ein und derselben Aufgabe zu schaffen, untersucht man die Eigenschaften von Algorithmen anhand der Größe (Dimension) des Problems.
Da man zum Vergleich verschiedener Algorithmen oft nur an deren asymptotischem Verhalten bezüglich Speicherbedarf bzw. Laufzeit interessiert ist und weniger an exakten Werten, hat sich zur Untersuchung die so genannte O-Notation (Landau-Symbole) bewährt und eingebürgert.

Definition A.4
Sei $G \subset \mathbb{R}$, $f, g : G \to \mathbb{R}$ und $x_0 \in G$.

a) *Die Funktion f heißt von der Ordnung $O(g(x))$ für $x \to x_0$, wenn es eine Konstante $C > 0$ und ein $\delta > 0$ gibt, so dass*

$$\forall |x - x_0| < \delta \qquad : \qquad \frac{|f(x)|}{|g(x)|} \leq C. \tag{A.3}$$

Dabei wird zusätzlich $x \neq x_0$ gefordert, falls g (und f) in x_0 nicht sinnvoll definiert werden können.

b) *Die Funktion f heißt von der Ordnung o $(g(x))$ für $x \to x_0$, wenn es für jede Konstante $C > 0$ ein δ gibt, so dass A.3 erfüllt ist.*

Beispiel A.5 (O-Notation)
1.) Ein Polynom vom Grad N

$$p(x) = \sum_{n=0}^{N} a_n x^n$$

ist asymptotisch (d.h. für $x \to \infty$) von der Ordnung $O(x^N)$ bzw. für jedes $\epsilon > 0$ von der Ordnung $o(x^{N+\epsilon})$.

2.) Der Satz von Taylor lässt sich für jede Funktion $f \in C^k(G)$ (G offen, $k \in \mathbb{N}$) schreiben als

$$f(x + h) = \sum_{n=0}^{k-1} \frac{f^{(n)}(x)}{n!} h^n + O(h^k)$$

bzw.

$$f(x + h) = \sum_{n=0}^{k-1} \frac{f^{(n)}(x)}{n!} h^n + o(h^{k-1}).$$

In dieser Form wird häufig auch die Approximationsqualität numerischer Verfahren untersucht und ausgedrückt.

Satz A.6
Das Landausymbol O hat die folgenden Eigenschaften:

(i) $f(x) = O(f(x))$.

(ii) $f(x) = o(g(x)) \Rightarrow f(x) = O(g(x))$.

(iii) *Sei* $0 < K \in \mathbb{R}$. *Dann gilt:* $f(x) = K\,O(g(x)) \Rightarrow f(x) = O(g(x))$.

(iv) *Wenn* $f_1(x) = O(g_1(x))$ *und* $f_2(x) = O(g_2(x))$, *dann gilt*

$$f_1(x) f_2(x) = O(g_1(x) g_2(x)).$$

(v) *Wenn* $f(x) = O(g_1(x) g_2(x))$, *dann gilt*

$$f(x) = g_1(x) O(g_2(x)).$$

Der Beweis dieser Eigenschaften sowie die Untersuchung der Frage, in wie weit analoge Aussagen für das Symbol $o(\cdot)$ gelten, ist als Übung überlassen.

Beispiel A.7 (Komplexität einer Polynomauswertung)
Sei p das Polynom aus Beispiel A.5. Für ein festes $x_0 \in \mathbb{R}$ soll $p_0(x_0)$ auf die zwei folgenden Arten ausgewertet werden:

/ Polynomauswertung 1.Methode */*

1.) Setze $p_0 := a_0$

2.) Für $n = 1, 2, ..., N$:

Setze Hilfsgröße $b := a_n$;
Für $m = 1, ..., n$:
multipliziere b mit x_0
$p_0 := p_0 + b$;

/ Polynomauswertung 2.Methode */*

1.) Setze $p_0 = a_0$
Setze Hilfsgröße $b := x_0$;

2.) Für $n = 1, 2, ..., N$:

Setze Hilfsgröße $c := a_n * b$;
$p_0 := p_0 + c$;
$b = b * x_0$;

Die Komplexität der 1. Methode lautet, wenn man nur die Multiplikation als wesentliche Operationen berücksichtigt:

$$\sum_{n=1}^{N} n = \frac{1}{2}N(N+1) = O(N^2)$$

Die Komplexität der 2. Methode hingegen ergibt:

$$2N = O(N)$$

A.5 Der Präprozessor

Alle C-Compiler übersetzen die Quellprogramme in mehreren Durchgängen. Im ersten Durchgang werden die Programme vom sogenannten Präprozessor verarbeitet. Dieser Teil des C-Compilers verarbeitet alle Anweisungen, die mit einem # beginnen, auch Präprozessor-Direktiven genannt. Er erzeugt aus der ursprünglichen Quelldatei einen temporären Quellcode, der anschließend von weiteren Hilfsprogrammen des Compilers verarbeitet wird (siehe Abbildung A.1).

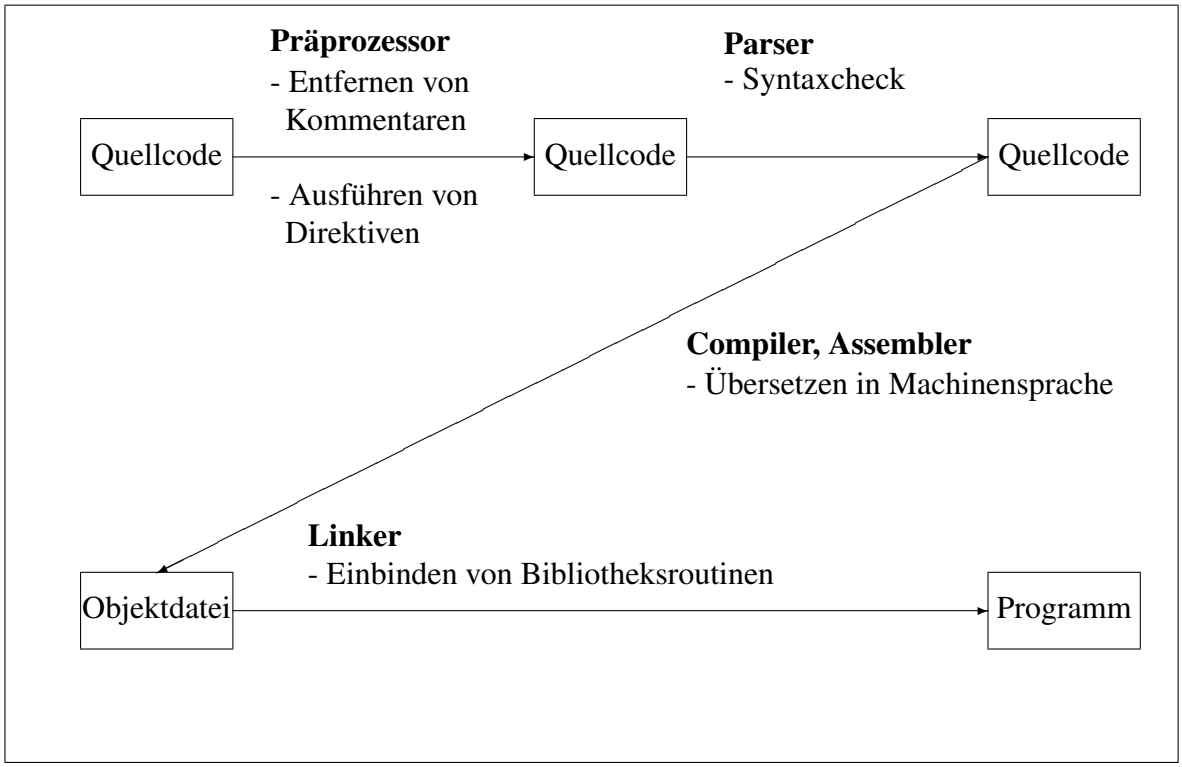

Abbildung A.1: Der Weg von der Quelldatei zum ausführbaren Programm

Dieser Vorgang bleibt für den Programmierer normalerweise unsichtbar. Man kann das Ergebnis aber auch speichern, um entweder seine Arbeit zu kontrollieren oder um sein Resultat anderweitig zu verarbeiten.

gcc *Quelltext*.c -E -o *dateiname*

Die so erzeugte Datei kann mit dem Editor betrachtet werden (vgl. Kapitel 1.6)

A.5.1 Dateien einfügen (# include)

Die wohl wichtigste Direktive für den Präprozessor ist *#include*. Mit ihrer Hilfe werden die Inhalte bestehender Dateien wie Textbausteine in das Programm eingefügt. Dabei handelt es sich im Normalfall um Headerdateien. Headerdateien enthalten üblicherweise die Endung *.h*

und dürfen keinen ausführbaren Code enthalten, sondern nur Deklarationen. Dies liegt daran, dass sie unter bestimmten Umständen in mehrere C-Dateien eingefügt und damit mehrfach übersetzt werden. Enthielten sie fertige Funktionen, dann lägen die hinterher für den Linker mehrfach vor. Dadurch kann er keine eindeutige Bindung herstellen.
Im folgenden Beispiel wird die Deklaration der Funktion *summe* in eine Headerdatei *summe.h* geschrieben. Im Quelltext des Programms wird dann die Deklaration der Funktion durch Einbinden des Headerfiles bewerkstelligt.

Beispiel A.8 (summe.h)

```
float summe (float ,float);
```

Beispiel A.9 (Einfügen des Header summe.h)

```
#include <stdio.h>
#include "summe.h"

/* Hauptprogramm */

int main()
{
        float x=2.0;
        float y=3.3;

        printf("Summe␣von␣x+y␣=␣%f␣\n",summe(x,y));
        return 0;

}

/* Definition summe-Funktion */

float summe (float x, float y)
{
        return x+y;
}
```

Bemerkung A.10

- *Die Dateinamen der Bibliotheksheader werden von spitzen Klammern <und >eingeschlossen. Derartig gekennzeichnete Dateien werden immer in den Bibliotheksverzeichnissen gesucht. Unter LINUX befindet sich die Header im Verzeichnis /usr/include.*
- *Eigene Dateien werden durch Anführungszeichen begrenzt. Sie werden dadurch vom Präprozessor im aktuellen Projektverzeichnis gesucht.*

A.5.2 Konstanten definieren (#define)

Der Präprozessorbefehl *#define* erlaubt es, eine Zeichenkette im Programm durch eine andere zu ersetzen. Die allgemeine Form sieht so aus:

define *Name Zeichenkette*

Am Beispiel des folgenden Programms können Sie die Arbeitsweise von *#define* nachvollziehen.

Beispiel A.11 (#define)

```
#include <stdio.h>

#define PI 3.14
#define Mein_PI "Mein␣Pi␣ist␣:"

int main()
{
       float x=PI;
       printf(Mein_PI);
       printf("␣%f\n",x);
}
```

Der Präprozessor manipuliert den Quellcode (vernachlässigt man die Direktive *#include <stdio.h>* zu:

```
int main()
{
       float x=3.14;
       printf("Mein␣Pi␣ist␣:");
       printf("␣%f\n",x);
}
```

Der Präprozessor kennt bereits einige eingebaute Konstanten, die nicht erst definiert werden müssen und auch nicht per *#define* überschrieben werden sollten. Dies sind:

__LINE__	für die aktuelle Zeilennummer
__FILE__	für den Namen der kompilierten Datei
__DATE__	für das aktuelle Datum
__TIME__	für die aktuelle Uhrzeit

Weitere nützliche Anwendungen von Präprozessor-Direktiven können der Literatur (z.B. in [Erl]) entnommen werden.

Literaturverzeichnis

[Knu] D.E. Knuth: The Art of Computer Programming - Vol. 2,2nd ed., Addison-Wesley

[Kof] M. Kofler: Linux - Installation, Konfiguration, Anwendung, Addison-Wesley

[Kru] G. Krüger: Go To C-Programmierung, Addison Wesley

[Oua] S. Oualline: Practical C Programming, O'Reilly

[WeDaDa] M. Welsh, M.K. Dalheimer, T. Dawson: Linux- Wegweiser zur Installation & Konfiguration, O'Reilly

[Erl] H. Erlenkötter: C - Programmieren von Anfang an, rororo

[KerRit] B.W. Kernighan, D.M. Ritchie: Programmieren in C, Hanser Verlag

[OraTal] A. Oram, S. Talbott: Managing Projects with make, O'Reilly

[PrTeVeFl] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery: Numerical Recipes in C, Cambridge University Press

[Kir] R. Kirsch: Einführung in die Programmierung für Mathematiker, Vorlesungsskript Sommersemester 2004

[Car] P. A. Carter : PC Assembly language, Online , http://www.drpaulcarter.com/pcasm/, 2003

Index