Teil III

Die Programmiersprache C

Kapitel 8

Einführung

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

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

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

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

8.1 Das erste C–Programm

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

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

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

[1] nach einer Vorlesung von Erik Wallacher

Folgende Strukturen finden wir in diesem ersten einfachen Programm vor:

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

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

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

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

```
gcc HalloWelt.c
```

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

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

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

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

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

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

8.2 Interne Details beim Compilieren (∗)

Der leicht geänderte Aufruf zum Compilieren

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

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

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

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

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

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

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

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

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

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

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

```
hexedit HalloWelt.o
```

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

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

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

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

Kapitel 9

Variablen, Datentypen und Operationen

9.1 Deklaration, Initialisierung, Definition

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

```
Datentyp Variablenname;
```

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

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

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

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

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

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

□

9.2 Elementare Datentypen

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

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

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

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

Beispiel 9.2 Deklaration, Initialisierung, Definition.

```
#include <stdio.h>

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

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

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

□

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

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

Bei den zulässigen Kombinationen ist die Reihenfolge

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

Beispiel 9.3 Deklaration / Definition von Variablen.
Zulässig:

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

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

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

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

Nicht zulässig:

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

□

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

Beispiel 9.4 C-Anweisung : sizeof().

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

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

□

9.3 Felder und Strings

9.3.1 Felder

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

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

Weitere Merkmale:

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

Beispiel 9.5 Felder.

```
#include<stdio.h>

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

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

□

9.3.2 Mehrdimensionale Felder

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

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

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

```
#include <stdio.h>

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

□

9.3.3 Zeichenketten (Strings)

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

`char Stringname[Länge];`

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

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

initialisiert.

Ein String kann auch durch

`char text[]="Hallo";`

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

Beispiel 9.7 Deklaration und Initialisierung eines Strings.

```
#include <stdio.h>

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

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

□

9.4 Ausdrücke, Operatoren und mathematische Funktionen

Der Zuweisungsoperator

```
operand1 = operand2;
```

weist dem linken Operanden den Wert des rechten Operanden zu.

Beispiel 9.8 Zuweisungsoperator. Zum Beispiel ist im Ergebnis der Anweisungsfolge

```
#include <stdio.h>

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

        return 0;
}
```

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

Es können auch Mehrfachzuweisungen auftreten.

Beispiel 9.9 Mehrfachzuweisung. Die folgenden drei Zuweisungen sind äquivalent.

```
#include <stdio.h>

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

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

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

□

9.4.1 Arithmetische Operatoren

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

Operator	Beschreibung	Beispiel
-	Negation	-a

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

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

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

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

Beispiel 9.10 Arithmetische Operatoren.

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

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

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

       return 0;
}
```

□

9.4.2 Vergleichsoperatoren

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

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

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

Beispiel 9.11 Vergleichsoperatoren.

```
#include <stdio.h>

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

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

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

        return 0;
}
```

□

9.4.3 Logische Operatoren

Es gibt nur einen unären logischen Operator

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

und zwei binäre logische Operatoren.

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

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

9.4.4 Bitorientierte Operatoren (∗)

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

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

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

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

Beispiel 9.12 (Short)-Integerdarstellung im Rechner.

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

□

Unäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
$\sim$	Binärkomplement	$\sim$ `a`

Binäre bitorientierte Operatoren

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

Wahrheitstafel

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

Beispiel 9.13 Bitorientierte Operatoren.

```
#include <stdio.h>

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

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

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

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

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

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

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

        return 0;
}
```

□

9.4.5 Inkrement- und Dekrementoperatoren

Präfixnotation

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

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

Beispiel 9.14 Präfixnotation.

```
#include <stdio.h>

int main()
{
        int i,j;

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

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

```
        j=i;

        return 0;
}
```

□

Postfixnotation

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

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

Beispiel 9.15 Postfixnotation.

```
#include <stdio.h>

int main()
{
        int i,j;

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

        /* oben angegebene Notatation ist aequivalent zu */

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

        return 0;
}
```

□

9.4.6 Adressoperator

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

```
&datenobjekt;
```

9.4.7 Prioritäten von Operatoren

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

Prioritäten von Operatoren beginnend mit der Höchsten

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 9.16 Prioritäten von Operatoren.

```
#include <stdio.h>

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

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

        return 0;
}
```

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

9.5 Operationen mit vordefinierten Funktionen

9.5.1 Mathematische Funktionen

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

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

Tabelle 9.2: Mathematische Funktionen

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

Beispiel 9.17 Mathematische Funktionen und Konstanten.

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

int main()
```

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

        return 0;
}
```

□

9.5.2 Funktionen für Zeichenketten (Strings)

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

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

Tabelle 9.3: Funktionen für Strings

Beispiel 9.18 Funktionen für Zeichenketten (Strings).

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

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

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

```
}
```

□

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

9.6 Zusammengesetzte Anweisungen

Wertzuweisungen der Form

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

können zu

```
op1 operator = op2;
```

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

Beispiel 9.19 Zusammengesetzte Anweisungen.

```
#include <stdio.h>

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

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

        return 0;
}
```

□

9.7 Nützliche Konstanten

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

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

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

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

Tabelle 9.4: Konstanten aus float.h

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

Tabelle 9.5: Konstanten aus limits.h

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

9.8 Typkonversion (cast)

Beispiel 9.20 Abgeschnittene Division.

```
#include <stdio.h>

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

        return 0;
}
```

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

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

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

Hierdurch wird das Ergebnis mit den Nachkommastellen übergeben. □

Achtung! bei Klammerung von Ausdrücken! Die Anweisung

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

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

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

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

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

kann man die Anweisung

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

verwenden.

9.9 Standardein– und –ausgabe

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

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

9.9.1 Ausgabe

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

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

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

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

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

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

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

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

Beispiel 9.22 Ausgabe von Gleitkommazahlen.

```
#include <stdio.h>

int main()
{
       const double pi=3.14159265;

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

       return 0;
}
```

erzeugen die Bildschirmausgabe

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

□

9.9.2 Eingabe

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

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

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

Beispiel 9.23 Einlesen einer ganzen Zahl.

```
#include <stdio.h>

int main()
{
       int a;

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

```
        return 0;
}
```

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

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

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

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

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

Ein Beispiel:

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

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

```
int zahl;
char buchstabe;

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

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

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

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

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

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

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

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

Kapitel 10

Programmflusskontrolle

10.1 Bedingte Ausführung

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

10.1.1 Die if()–Anweisung

Die allgemeine Form der Verzweigung (Alternative) ist

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

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

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

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

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

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

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

Die Kurzform des obigen Programms ist

```
#include <stdio.h>

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

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

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

□

10.1.2 Die switch()–Anweisung

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

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

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

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

Beispiel 10.2 switch–Anweisung.

```
#include <stdio.h>

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

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

□

10.2 Schleifen

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

10.2.1 Der Zählzyklus (for–Schleife)

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

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

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.2.2 Abweisender Zyklus (while–Schleife)

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

Die allgemeine Form ist

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

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

```
#include <stdio.h>

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

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

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

□

10.2.3 Nichtabweisender Zyklus (do–while–Schleife)

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

Die allgemeine Form ist

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

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

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

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

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

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

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

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

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

10.3 Anweisungen zur unbedingten Steuerungsübergabe

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

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

Kapitel 11

Zeiger (Pointer)

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

11.1 Adressen

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

Beispiel 11.1 Adressen von Variablen.

```
#include <stdio.h>

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

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

Nach dem Start des Programms erscheint folgende Ausgabe:

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

□

Bemerkung 11.2 Zu Beispiel 11.1.

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

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

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

□

11.2 Pointervariablen

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

```
Datentyp *Variablenname;
```

Das nächste Programm veranschaulicht diese Schreibweise.

Beispiel 11.3

```
#include <stdio.h>

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

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

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

Das Programm erzeugt folgende Ausgabe:

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

□

Bemerkung 11.4 Zu Beispiel 11.3.

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

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

□

11.3 Adressoperator und Zugriffsoperator

Der unäre Adressoperator `&` (Referenzoperator)

```
&Variablenname;
```

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

```
*pointer;
```

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

Beispiel 11.5

```
#include <stdio.h>

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

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

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

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

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

Das Programm erzeugt folgende Ausgabe:

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

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

□

11.4 Zusammenhang zwischen Zeigern und Feldern

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

Beispiel 11.6 Zeiger und Felder.

```
#include <stdio.h>

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


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

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

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

□

Bemerkung 11.7 Zu Beispiel 11.6.

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

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

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

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

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

5.) Die folgenden Operatoren sind auf Zeiger anwendbar:

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

11.5 Dynamische Felder mittels Zeiger

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

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

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

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

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

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

       /* Dynamische Speicher Reservierung */
```

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

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

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

□

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

Beispiel 11.9 Dynamisches 2D Feld.

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

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

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

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

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

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

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

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

□

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

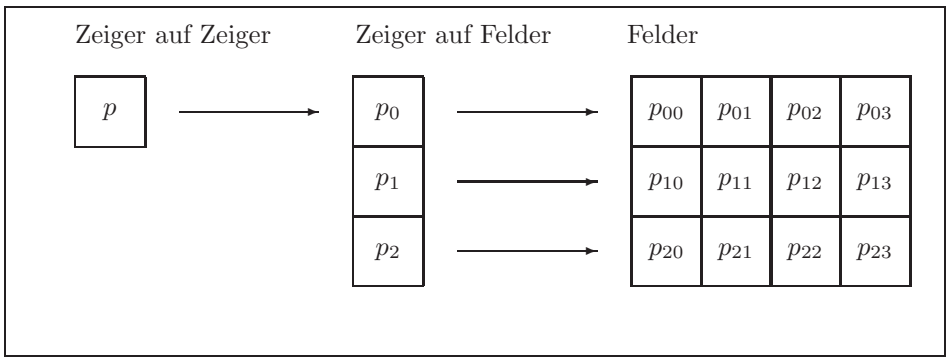

Abbildung 11.1: Dynamisches 2D Feld mit Zeiger auf Zeiger

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

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

Kapitel 12

Funktionen

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

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

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

12.1 Deklaration, Definition und Rückgabewerte

Die Deklaration einer Funktion hat die Gestalt

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

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

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

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

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

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

Folgendes ist bei Funktionen zu beachten:

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

Beispiel 12.1 Funktion.

```
#include <stdio.h>

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

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

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

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

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

□

12.2 Lokale und globale Variablen

Variablen lassen sich nach ihrem Gültigkeitsbereich unterteilen:

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

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

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

Beispiel 12.2 Gültigkeitsbereich von Variablen.

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

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

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

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

    return 0;
}

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

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

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

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

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

Die Ausgabe ist

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

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

12.3 Call by value

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

Beispiel 12.3 Call by value I.

```
#include <stdio.h>

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

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

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

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

Die Ausgabe lautet:

```
b=0
```

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

Beispiel 12.4 Call by value II.

```
#include <stdio.h>

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

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

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

Die Ausgabe lautet:

```
b=3
```

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

12.4 Call by reference

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

Beispiel 12.5 Call by reference.

```
#include <stdio.h>

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

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

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

Die Ausgabe lautet:

```
b=3
```

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

12.5 Rekursive Programmierung

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

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

```
#include <stdio.h>

/* Deklaration potenz-Funktion */
```

```
double potenz (double, int);

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

        return 0;
}

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

Dieser Rekursion liegt die Darstellung

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

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

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

12.6 Kommandozeilen–Parameter

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

Beispiel 12.7 Kommandozeilen–Parameter.

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

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

Nach dem Start des obigen Programms zum Beispiel durch

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

erscheint folgende Ausgabe auf dem Bildschirm

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

Die Summe der Kommandozeilen-Parameter : 9.100000
```

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

Zeile 3: `main` erhält vom Betriebssystem zwei Parameter. Die Variable `argc` enthält die Anzahl der übergebenen Parameter und `argv[]` die Parameter selbst. Die Namen der Variablen `argc` und `argv` sind natürlich frei wählbar, es hat sich jedoch eingebürgert, die hier verwendeten Bezeichnungen zu benutzen. Sie leiten sich von *arg*ument *c*ount und *arg*ument *v*alues ab.

Bei `argc` ist eine Besonderheit zu beachten. Hat diese Variable zum Beispiel den Wert 1 ist, so bedeutet das, dass kein Kommandozeilen–Parameter eingegeben wurde. Das Betriebssystem übergibt als ersten Parameter nämlich grundsätzlich den Namen des Programms selbst. Also erst wenn `argc` größer als 1 ist, wurde wirklich ein Parameter eingegeben.

Die Deklaration `char *argv[]` bedeutet: Zeiger auf Zeiger auf Zeichen. Man hätte auch `char **argv` schreiben können. Mit anderen Worten: `argv` ist ein Zeiger, der auf ein Feld zeigt, das wiederum Zeiger enthält. Diese Pointer zeigen schließlich auf die einzelnen Kommandozeilen–Parameter. Die leeren eckigen Klammern weisen darauf hin, dass es sich um ein Feld unbestimmter Größe handelt. Die einzelnen Argumente können durch Indizierung von `argv` angesprochen werden. `argv[1]` zeigt also auf das erste Argument (”1.4”), `argv[2]` auf ”3.2” und so weiter.

Zeile 15: Da es sich bei `argv[i]` um Strings handelt müssen die Eingabeparameter eventuell (je nach ihrer Bestimmung) in einen anderen Typ umgewandelt werden. Dies geschieht in diesem Beispiel mit Hilfe des `sscanf`–Befehls. □

12.7 Wie werden Deklarationen gelesen?

Eine Deklaration besteht grundsätzlich aus einem *Bezeichner* (Variablennamen oder Funktionsnamen), der durch einen oder mehrere *Zeiger*–, *Feld*– oder *Funktions*–Modifikatoren beschrieben wird. Wenn mehrere solcher Modifikatoren miteinander kombinieren, muss man darauf achten, dass Funktionen keine Funktionen oder Felder zurückgeben können und dass Felder auch keine Funktionen als Elemente haben können. Ansonsten sind alle Kombinationen erlaubt. Dabei haben Funktions– und Array–Modifikatoren Vorrang vor Zeiger–Modifikatoren. Durch Klammerung kann diese Rangfolge geändert werden.

Bei der Interpretation beginnt man am besten beim *Bezeichner* und liest nach rechts bis zum Ende beziehungsweise bis zu einer einzelnen rechten Klammer. Dann fährt man links vom Bezeichner mit eventuell vorhandenen Zeiger–Modifikatoren fort, bis das Ende oder eine einzelne linke Klammer erreicht wird. Dieses Verfahren wird für jede geschachtelte Klammer von innen nach außen wiederholt. Zum Schluss wird der Typ–Kennzeichner gelesen.

Beispiel 12.8

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

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

Beispiel 12.9 Die folgenden vier Beispiele verdeutlichen den Einsatz von Klammern:

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

□

12.8 Zeiger auf Funktionen

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

Beispiel 12.10 Zeiger auf Funktionen. In diesem Beispiel wird der Funktion `trapez_regel` ein Zeiger auf die zu integrierende Funktion mitgeliefert. Dadurch ist es möglich, beliebige Funktionen mit Hilfe der Trapez–Regel numerisch zu integrieren. Die Intervallgrenzen und Anzahl der Stützstellen sollen dem Programm durch Kommandozeilen–Parameter übergeben werden.

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

/*************************************************************/
/* Deklaration der Funktion trapez_regel()             */
/* Eingabeparameter : 1.) Zeiger auf Funktion mit
 *                      Eingabeparameter double-Wert
```

```
 *                      und double-Rueckgabewert
 *                  2.) double fuer linke Intervallgrenze
 *                  3.) double fuer rechte Intervallgrenze
 *                  4.) int fuer Anzahl der Stuetzstellen
 * Rueckgabewert : double fuer das Integral
*****************************************************************/
double trapez_regel(double (*f)(double), double, double, int);

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

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

       if (argc<4)
       {
              printf("Programm ben\"otigt 3 Kommandozeilenparameter :\n");
              printf("1.) Linker Intervallrand (double)\n");
              printf("2.) Rechter Intervallrand (double)\n");
              printf("3.) Anzahl der Teilintervalle fuer");
              printf(" numerische Integration (int)\n");

              return 1;
       }
       else
       {
              sscanf(argv[1],"%lf",&a);
              sscanf(argv[2],"%lf",&b);
              sscanf(argv[3],"%i",&n);


              fptr=(double(*)(double)) cos;/* Zeiger fptr auf cos-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der cos-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
              printf(" Stuetzstellen)\n");
              printf("        \t %f (exakt)\n\n",sin(b)-sin(a));

              fptr=(double (*)(double)) sin;/*Zeiger fptr auf sin-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der sin-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
              printf(" Stuetzstellen)\n");
              printf("        \t %f (exakt)\n\n",cos(a)-cos(b));

              return 0;
       }
}
```

```
/*********************************************************************/
/* Definition der Funktion trapez_regel()                            */
/*********************************************************************/
double trapez_regel(double (*f)(double ),double a,double b,int n)
{
        n=n+1;
        int k;
        double h=(b-a)/n;
        double integral=0;

        for (k=0;k<=n-1;k++) integral=integral+h/2*(f(a+k*h)+f(a+(k+1)*h));

        return integral;
}
```

□