Theorie und Praxis der Programmentwicklung am Beispiel der Programmiersprache C

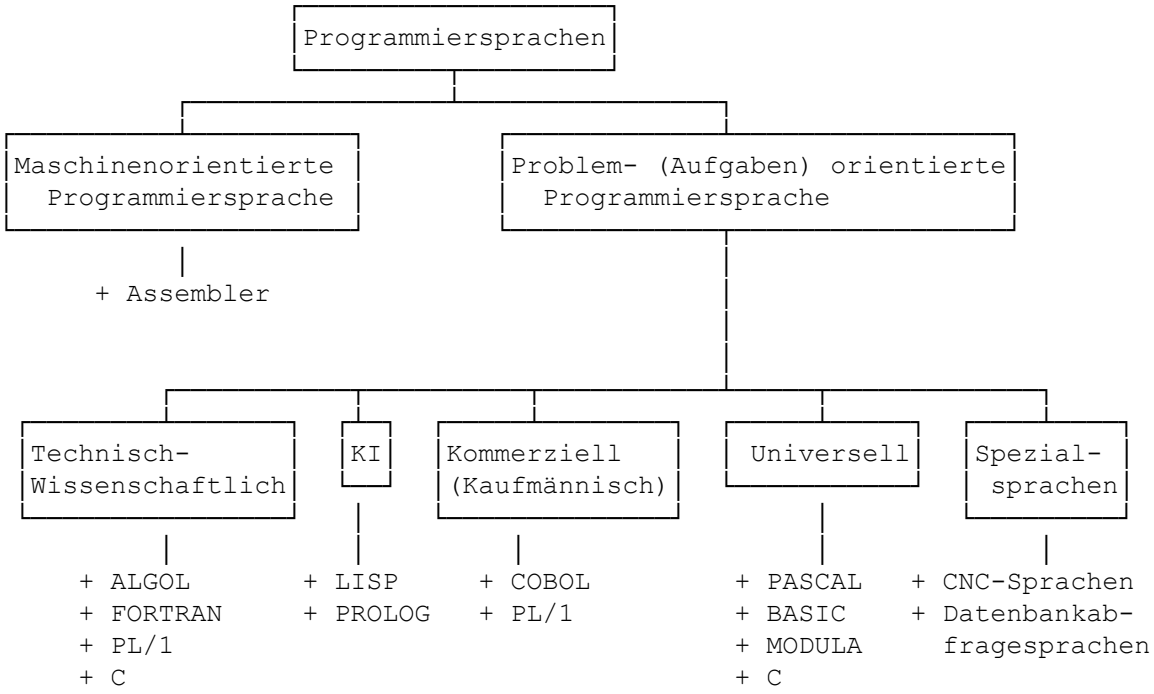
Inhaltsverzeichnis

Für den Inhalt dieses Vorlesungsskripts wird in Syntax und Semantik keine Garantie übernommen. Wer einen Fehler findet, darf ihn behalten.

1. Software-Grundlagen

1.1 Überblick der Programmiersprachen

Programmiersprachen dienen allgemein zur Kommunikation zwischen Computern und den Menschen. Vor der Entwicklung hochintegrierter Speicherbausteine wurden zur Programmierung fast ausschließlich maschinenorientierte Sprachen verwendet. Parallel verlief die Entwicklung der problemorientierten, symbolischen Programmiersprachen, weil einerseits immer neue und größere Anwendungsfälle erschlossen wurden und andererseits die maschinenorientierten Sprachen den Nachteil der schweren Verständlichkeit aufwiesen. Die problemorientierten Sprachen werden auch als höhere Programmiersprachen bezeichnet.



Abkürzungen:

+ ALGOL	Algorithmic Language One
+ FORTRAN	Formula Translator
+ LISP	List Processing
+ PROLOG	Programming in Logic
+ COBOL	Common Business Oriented Language
+ PL/1	Programming Language 1
+ PASCAL	von Nikolaus Wirth nach dem Mathematiker Blaise Pascal (1623-62) benannt
+ BASIC	Beginners All purpose Symbolic Instruction Code

1.2 Stufen der systematischen Programmentwicklung

Bei umfangreichen Problemstellungen hat sich folgende Vorgehensweise bewährt:

I. Aufgabenstellung
- Problembeschreibung

II. Problemanalyse
- Festlegung der Eingabedaten und deren Datentypen
- Festlegung der Ausgabedaten und deren Datentypen
- Zerlegung in Einzelprobleme (top down Methode)
- Formulierung des Algorithmus

III. Grafische Darstellung des Lösungsweges
- Datenflußplan (bei datenintensiven Problemstellungen)
- Programmablaufplan oder Struktogramm

IV. Codierung des Algorithmus
- Umsetzung der Algorithmen der Teilfunktionen in eine Folge von Programmanweisungen

V. Programmtest
- Prüfung auf logische Fehler
- Prüfung auf Codierfehler

VI. Programmdokumentation
- Programmbeschreibung
- Gebrauchsanweisung

bei größeren Projekten:

VII. Programmpflege bzw. -wartung
- Programmerweiterung
- Fehlerbeseitigung (bug fixing)

1.3 Vom Problem zum Programm am Beispiel: Reiskornaufgabe

I. Aufgabenstellung

Auf dem ersten Feld eines Schachbrettes befindet sich ein Reiskorn. Auf den folgenden Feldern befindet sich jeweils die doppelte Menge an Reiskörnern. Wie groß ist die Gesamtmenge der Reiskörner auf dem Schachbrett bis zu einem beliebigen Feld ?

II. Problemanalyse

- *Eingabedaten:*
  Anzahl der Felder, für die die Gesamtsumme ermittelt werden soll.
  Variable: anzahl -> Datentyp integer

- *Ausgabedaten:*
  Summe der Reiskörner bis zu einem bestimmten Feld
  Variable: summe -> Datentyp unsigned long int

- Zerlegung in Einzelprobleme (top down Methode)

```
Feld 1: 1  -> 20
Feld 2: 2  -> 21
Feld 3: 4  -> 22
Feld 4: 8  -> 23
Feld 5: 16 -> 24
  ......
  ......
Feld n:       2n-1
```

Der Algorithmus lautet somit:
Führe für den Wert n=1 bis zum Wert n=anzahl nacheinander folgende Operationen durch:

summe = summe + 2^{n-1}

III. Grafische Darstellung des Lösungsweges

- Struktogramm

Header-Dateien angeben
Deklarationen
Lies anzahl von Tastatur
summe = 0
FOR n=1 TO anzahl
summe summe + 2^{n-1}
Ausgabe von summe

IV. Codierung des Algorithmus

- Umsetzung der Algorithmen der Teilfunktionen in eine Folge von Programmanweisungen

```
/*****************************************************************/
/* Programm:  Reiskornaufgabe                                    */
/* Autor   :  Ludwig Eckert                                      */
/*****************************************************************/
#include <stdio.h>
#include <conio.h>
#include <math.h>

unsigned long int  summe;      /* Ausgabevariable */
unsigned int      anzahl;      /* Eingabevariable */

 unsigned int  Eingabe(void)
/*****************************************************************/
/* liest einen Zahlenwert (0 - 65535) von der Tastatur           */
/*****************************************************************/
{
 unsigned int  gibwertein;

 clrscr();         /* löscht den Bildschirm*/
 gotoxy(10,9);
 printf("\nBitte geben Sie die Anzahl Felder an : ");
 scanf("%u", &gibwertein);
 return (gibwertein);
}

 unsigned long int Berechnung (unsigned int wiederhole)
/*****************************************************************/
/* Führe für den Wert n=1 bis zum Wert n=anzahl nacheinander     */
/* folgende Operationen durch: summe = summe + 2n-1              */
/*****************************************************************/
{
 unsigned long int  ergebnis;
 double n;                  /* Hilfvariable n */

 ergebnis=0;                /* Initialisierung */
 for (n=0 ; n<wiederhole; n++)
   {
    ergebnis = ergebnis + 2*pow (2, n-1);
   }
 return (ergebnis);
}


Ausgabe(unsigned long int gebeaus)
/*****************************************************************/
/*     Variable gebeaus wird formatiert ausgegeben               */
/*****************************************************************/
{
printf(" \nAuf dem %3u. Feld befinden sich %10u Reiskörner",
        anzahl, gebeaus);
}

main()  /* Hauptprogramm */
{
 anzahl = Eingabe();
```

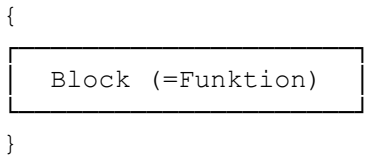
```
 summe = Berechnung (anzahl);
 Ausgabe(summe);
 getch();

}   /* Ende main */
```

2. Der Software-Lifecycle

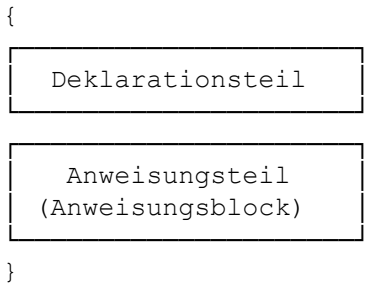
--

3. Die Architektur von Standard C Programmen

Die Sprache Standard C ist wie Standard-C eine blockstrukturierte Sprache. Die Blöcke, auch als Funktionen oder Unterprogramme bezeichnet, sind umschlossen von den reservierten geschweiften Klammern ***{*** und ***}***.



Grundsätzlich ist ein Standard C Programm nach folgendem Schema aufgebaut:

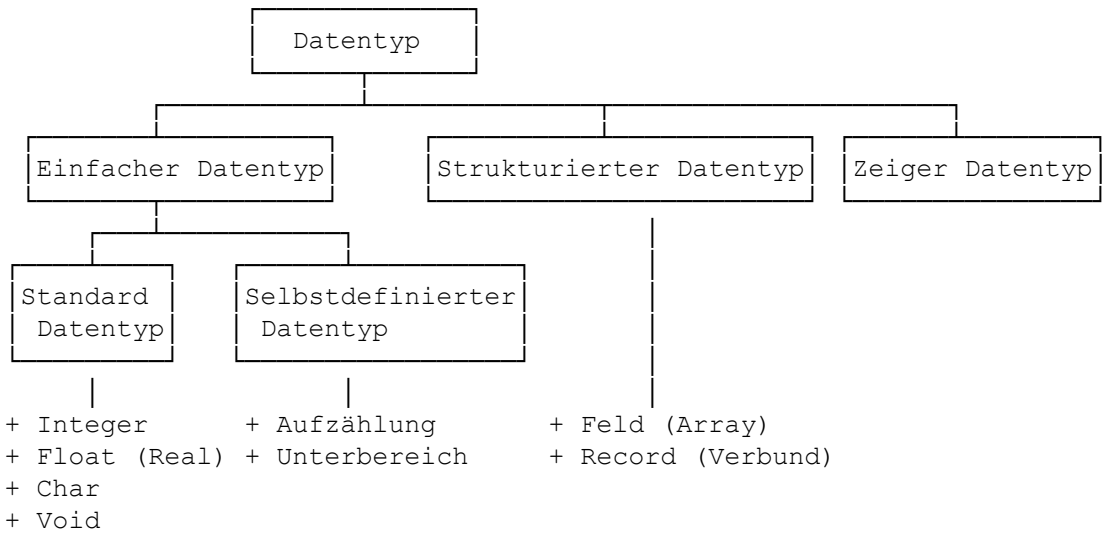


3.1 Der Deklarationsteil

Der Deklarationsteil besteht im einfachen Fall nur aus einer Variablenerklärung. Allgemein kann der Deklarationsteil folgende Komponenten enthalten:

(1) Include-Angaben für Headerdateien
(2) Konstantenerklärungen
(3) Typenerklärungen
(4) Variablenerklärungen

Jedes im Programm verwendete Datum ist eindeutig einer Wertemenge, d.h. einem Datentyp, zugeordnet. Der Datentyp definiert die Wertemenge, die eine Variable annehmen kann und die Operationen, die auf die Wertemenge ausgeübt werden können.



Die Deklaration legt somit die Wertemenge und den Speicherplatzbedarf einer Variablen oder einer Funktion mit Rückgabewert fest.

In Standard C zulässige Standarddatentypen und deren Wertebereiche:

Datentyp	Wertebereich			Bitbreite
char	-128	bis	+127	8
unsigned char	0	bis	+255	8
int	-32768	bis	+32767	16
unsigned int	0	bis	+65535	16
long int	-2147483648	bis	+2147483647	32
unsigned long int	0	bis	+4294967295	32
float	-3,4 * 10-38	bis	+3,4 * 10+38	32
double	-1,7 * 10-308	bis	+1,7 * 10+308	64
long double	-3,4 * 10-4932	bis	+3,4 * 10+4932	80
void	wertfrei			

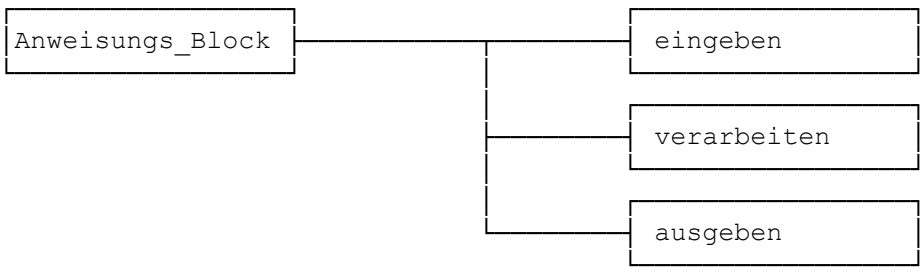
Mit den Vorworten signed und unsigned kann festgelegt werden, ob die Variable Werte mit oder ohne Vorzeichen annehmen kann.

3.2 Der Anweisungsteil

Wie oben schon erläutert bestehen Standard CÿProgramme aus Funktionen. Ein Programm enthält dabei mindestens eine Funktion, nämlich die Hauptfunktion main(), die beim Programmstart selbsttätig aufgerufen wird.

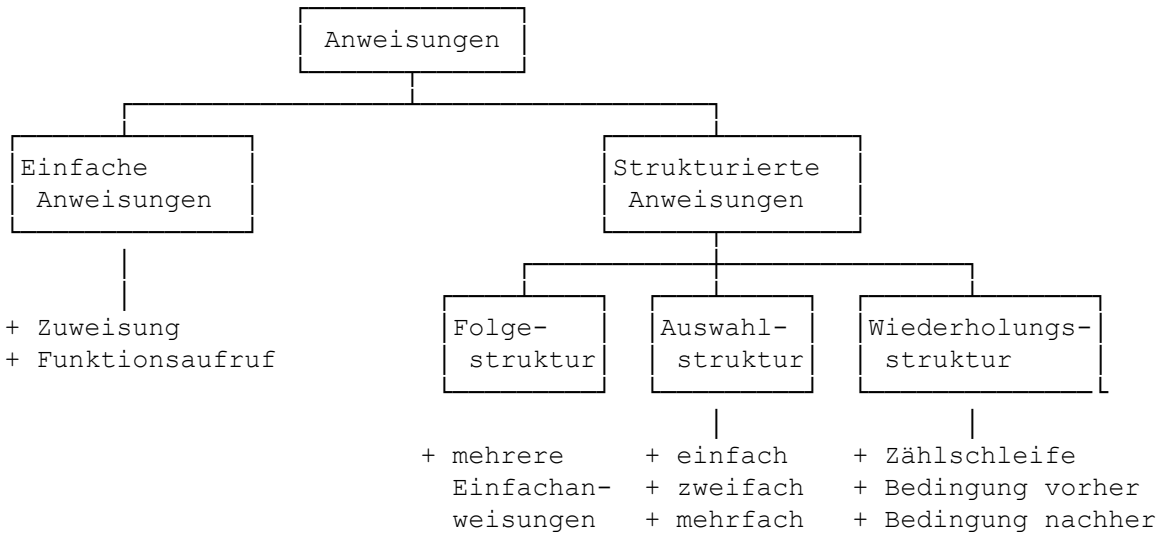
Ein Block definiert die Aktionen, die von der Funktion oder dem Unterprogramm ausgeführt werden soll als Folge von Anweisungen, sog. statements. Anweisungen werden im Programmlauf zeitlich sequentiell abgearbeitet. Durch Anweisungen ausgeführte Aktionen sind z.B. Eingabe, Verarbeitung und Ausgabe von Daten.

Die Anweisungen innerhalb einer Funktion werden jeweils durch ein Semikolon ";" voneinander getrennt.



Der Anweisungsteil setzt sich immer aus mindestens einer, meist jedoch mehreren, einfachen Anweisungen zusammen.

Einen Überblick über die in Standard C verfügbaren Anweisungsarten gibt folgende Grafik:



3.3 Regeln für die Vergabe benutzerdefinierter "Bezeichner"

Für die Vergabe von Bezeichnern müssen einige Regeln beachtet werden:

- Bezeichner müssen mit einem Buchstaben (A,B..Z,a,b..z) oder einem Unterstrich beginnen. Nachfolgende Zeichen können Buchstaben, Ziffern oder weitere Unterstriche sein.

- Deutsche Umlaute (Ä,Ö,Ü) sowie das ß sind in Standard C und fast allen bekannten C-Compilern nicht erlaubt.

- Nur die ersten 32 Zeichen werden zur Bezeichneridentifikation herangezogen; z.B.

  ```
  12345678901234567890123456789012-45
  function_ziehe_wurzel_aus_zwei_und_addiere_100
  function_ziehe_wurzel_aus_zwei_und_subtrahiere_100
  ```

  Die beiden letzten Bezeichner sind für Standard C identisch, da nur die ersten 32 Zeichen ausgewertet werden.

- Standard C unterscheidet grundsätzlich zwischen Groß- und Kleinschreibung, d.h. Eingabezahl1, EINGABEZAHL1, EingabeZahl1, eingabezahl1 stellen vier verschiedene Bezeichner dar, die völlig unabhängig voneinander sind.

- Für die Bezeichner von Variablen oder Konstanten dürfen - wie in anderen Programmiersprachen auch - keine reservierten Wörter verwendet werden. Zu den reservierten Wörtern gehören die Schlüsselwörter, bzw. Befehle, die den Sprachvorrat einer Programmiersprache ausmachen, und die Funktionsnamen.

Beispiele:

char,int,float	->	vordefinierte Datentypen, d.h. reservierte Wörter von Standard C
main	->	Name der Hauptfunktion eines Programms
printf,scanf,puts	->	Funktionen von Standard C

```
 zahl1,addiere,a,b,      ->    Variablenbezeichner
 KONST1,MAXIMAL,MINI     ->    Konstantenbezeichner
```

3.4 Reservierte Wörter und Standardnamen in Standard C

Der Sprachumfang des Standard C-Compilers besteht aus Befehlen und Funktionen. Die Befehle werden auch als Schlüsselwörter bezeichnet und stellen eine reservierte Menge von Wörtern dar, die niemals für Bezeichner-Namen (z.B. für Konstanten- und Variablenvereinbarungen) oder Funktions-Namen verwendet werden dürfen. Nach dem ANSI-Standard sind dies:

Datentypen: char, double, float, int, long, short, signed, unsigned
Abgeleitete Datentypen: enum, struct, union

Parametertypen: const, volatile

Anweisungen: break, case, continue, default, do, else, for, goto, if, switch, while

Unterprogramme (Funktionen): return, void

Speicherklassen: auto, extern, register, static, typedef

Speicherverwaltung: sizeof

zusätzlich in Standard C: main, far, near, huge, cdecl, asm, pascal, _cs, _ds, _es, _ss, interrupt

Standard C kennt nur die oben angegebenen Befehle (=Schlüsselwörter) und Standardnamen. Im Vergleich zu anderen höheren Programmiersprachen (z.B. Basic, Pascal) ist diese Zahl eher niedrig. Umso größer ist hierfür die Anzahl der Funktionen in den Funktionsbibliotheken. So werden die im Sprachumfang von Standard C fehlenden Ein- und Ausgabe-Befehle durch E/A-Funktionen ersetzt.

4. Grundlegende Programmstrukturen

Höhere problemorientierte strukturierte Programmiersprachen enthalten generell vier verschiedene Konstruktionen zur Algorithmenbildung. Mit Hilfe dieser grundlegenden Programmstrukturen lassen sich alle nur erdenklichen Programmabläufe realisieren. Diese sind:

- Folgestrukturen (linear, geradeaus, sequentiell)
- Auswahlstrukturen (vorwärts verzweigend)
- Wiederholungsstrukturen (rückwärts verzweigend, Schleife)
- Unterprogrammstrukturen (in Teile aufgliedernd)

Werden mehrere Grundstrukturen aneinandergereiht, so spricht man von einem zusammengesetzten Strukturblock. In den nächsten Unterkapiteln soll nun näher auf einzelne Kontrollstrukturen eingegangen werden.

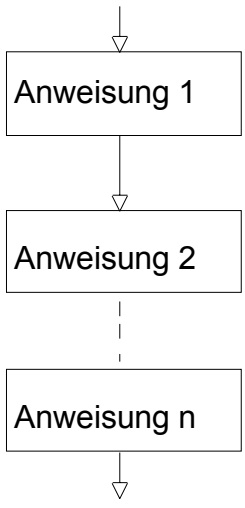
4.1 Die Folgestrukturen

Die Folgestruktur, auch Sequenz genannt, ist das einfachste Steuerkonstrukt. Sie besteht aus einer Aneinanderreihung von mindestens zwei Anweisungen. Die Folgestruktur wird vom Computer genau in der Reihenfolge ausgeführt, in der sie niedergeschrieben ist (von oben nach unten).

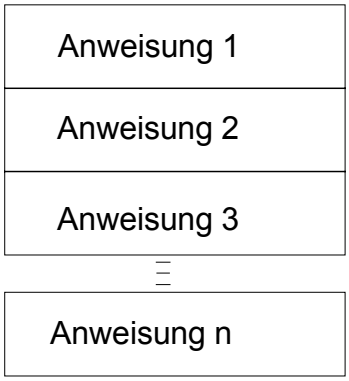
Aus einem Ausdruck wird eine Anweisung, wenn dem Ausdruck ein Semikolon folgt, z.B.
Ausdruck: x = y+1 Anweisung: x = y+1;

Die Zusammenfassung von Vereinbarungen (Deklarationen) und Anweisungen in den geschweiften Klammern { } bilden einen Anweisungsblock. So gesehen entspricht ein Anweisungsblock genau der oben beschriebenen Folgestruktur.

Im Programmablaufplan (Flußdiagramm)



Im Nassi-Schneidermann-Diagramm (Struktogramm)



Hinweise:

- Nach der geschweiften, den Anweisungsblock abschließenden Klammer darf kein Semikolon stehen.
- Statt einer Anweisung darf auch ein Anweisungsblock stehen.

Beispiel:

```
main()
{
 int x=1;              /* Vereinbarung und Initialisierung */
 int i;                /* Vereinbarung */

 i=12;                 /* Zuweisungsanweisung */
 printf("x ist %d und i ist %d",x,i);     /* Ausgabeanweisung */
}
```

4.1.1 Ein- und Ausgabeanweisungen

Die wohl am häufigsten verwendeten Anweisungen in Programmiersprachen sind Ein- und Ausgabeanweisungen. In Standard C sind dies die Funktionen printf und scanf.

- **Die Ausgabefunktion printf**

Die Funktion printf stellt eine sehr universelle Ausgabefunktion dar. Mit ihr können nicht nur Text, sondern auch Ergebnisse von Berechnungen, Variablen-inhalte und Konstanten formatiert und ggf. transformiert, d. h. unterschiedlich interpretiert, zur Ausgabe gebracht werden.

Die Syntax lautet:

printf("<Formatstring+Steueranw>", <Wert1>, <Wert2>, <Wert3>, ..);

Die printf-Funktion verarbeitet eine Vielzahl von Steuer- und Formatanweisungen. Steueranweisungen werden immer durch einen Rückstrich (backslash) " \ " eingeleitet, gefolgt von einem Buchstaben, der die Art der Steueranweisung angibt.

Beispiele:

```
printf("Guten Morgen !")          // Ausgabe von Text ohne Formatierung

printf("Guten Morgen !\n\r\a")    // Ausgabe von Text mit Formatierung: Der
                                  // Cursor wird an den Anfang der neuen Zeile
                                  // gesetzt, schließlich ertönt ein kurzer Piepton.
```

Folgende Tabelle faßt die für printf verfügbaren Formatierungsanweisungen zusammen:

Zeichen	**Bedeutung**
\a	Piepton (Bell)
\b	Cursor ein Zeichen zurück (Backspace)

```
\f          Seitenvorschub (formfeed)
\n          Zeilenvorschub (Linefeed)
\r          Wagenrücklauf (Carriage Return)
\t          Horizontaler Tabulatorsprung (HTAB)
\v          Vertikaler Tabulatorsprung (VTAB)
\xhhh       ASCII-Zeichen mit dem Wert der folgenden Hexadezimal
            zahl hhh; hhh steht für die Anzahl der Hexziffern
            z.B.:  "\x0D"  ->  Wagenrücklauf
\0DDD       ASCII-Zeichen mit dem Wert der folgenden Oktalzahl DDD
            z.B.:  "\015"  ->  Wagenrücklauf
\\          Rückstrich (Backslash)
\'          Hochkomma  '
\"          Anführungszeichen  "
\?          Fragezeichen ?
```

- **Die Eingabefunktion scanf**

Die universelle Eingaberoutine zum Lesen der Tastatur ist die Funktion scanf. Sie liest alle über Tastatur eingehenden Zeichen und wandelt diese automatisch, je nach Formatanweisung, in einen passenden internen Datentyp um. Die Funktion scanf ist das Gegenstück zur Funktion printf und weist die allgemeine Form auf:

scanf(<Formatstring>, <Adresse1>, <Adresse2>, <Adresse3>, ..);

scanf erwartet für Wertzuweisungen ausschließlich die Adressen der im Programm verwendeten Variablen, und nicht den Variablennamen selbst. Für Standarddatentypen muss hier der Adressoperator "&" verwendet werden, z.B.

```
  int Wert1;    /* Variable Wert1 ist vom Datentyp integer */

  | Wert1 |
  ┕━━━━━━━┙
     |
Variablen_Adresse= &Wert  /*Adresse der Variable Wert1 durch*/
                                /* Adreßoperator & ermittelt*/
```

Zur Verdeutlichung weitere Beispiele:

Eine Adresse soll in ein Zeichenfeld eingelesen werden:

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

Zwei Dezimalzahlen sollen durch ein Programm eingelesen werden:

```
 int Zahl1;
 int Zahl2;
 scanf("%d%d", &Zahl1, &zahl2);
 // Hinweis: Beide Zahlen müssen durch ein Leerzeichen voneinander getrennt werden.
```

Zwanzig Zeichen sollen über Tastatur eingelesen werden.

```
 char  Vorname[20];
 scanf("%20s",Vorname);
 // Hinweis: Es werden genau 20 Zeichen eingelesen, der Rest wird ignoriert.
```

Folgende Tabelle faßt die für scanf und printf möglichen Steueranweisungen zusammen:

Zeichen	Bedeutung
%c	liest ein einzelnes Zeichen
%d	liest eine vorzeichenbehaftete dezimale Ganzzahl (integer)
%i	liest eine vorzeichenbehaftete dezimale, oktale oder hexadezimale Ganzzahl (integer)
%e	liest eine Fließkommazahl (float)
%f	liest eine Fließkommazahl (float)
%h	liest einen Bytewert
%o	liest eine vorzeichenbehaftete oktale Ganzzahl (integer)
%s	liest einen String
%x	liest eine hexadezimale Ganzzahl ein (integer)
%p	liest einen Pointer (Formatangabe XXXX:YYYY)
%n	integer-Variable n gibt an, wieviele Zeichen gelesen werden sollen

4.1.2 Die Zuweisungsanweisung

Zuweisungen stellen die grundlegendsten Operationen jeder Programmiersprache dar. Standard C verwendet das Gleichheitszeichen "=", um einer Variablen auf der linken Seite den Wert der rechten Seite zuzuweisen. Eine Zuweisung wird immer von rechts nach links evaluiert, z.B.

```
int  Wert1,Wert2,Summe;    /* Variablendeklaration */

Wert1 = Wert2 = 4;         /* Mehrfachzuweisungen sind möglich */
Summe = Wert1 + Wert2;     /* Variable Summe erhält den Wert 8 */
```

Zusätzlich verfügt Standard C wie keine andere Sprache über viele mächtige Operatoren. Operatoren dienen zur Verknüpfung und Manipulation von Daten. Standard C unterscheidet folgende Operatorentypen:

Operatoren - Arithmetische Operatoren
- Zuweisungsoperatoren
- Inkrement- und Dekrementoperatoren
- Bitoperatoren
- Vergleichsoperatoren
- Logische Operatoren
- Adressoperatoren
- Der Cast-Operator

- **Arithmetische Operatoren**

Standard C stellt 6 arithmetische Operatoren zur Verfügung:

+	Addition,
-	Subtraktion,
*	Multiplikation
/	Division
-	Vorzeichen, z.B. -zahl1 (Zweierkomplementbildung)
%	Modulo-Operator nur für ganze Zahlen, z.B. 8 % 6 = 2

- **Inkrement- und Dekrementoperatoren**

Die Operatoren "++" und "--" wurden von der Assemblersprache in Standard C übernommen (INC -> ++ , DEC -> --) und inkrementieren eine Variable jeweils ukm den Wert eins oder herunter (dekrementieren).

z.B.:

```
 x++;     entspricht:    x = x + 1;   (postfix = nachstehend)
 ++x;     entspricht:    x = x + 1;    (präfix = vorrangig)

 x--;     entspricht:    x = x - 1;
 --x;     entspricht:    x = x - 1;
```

Die erste Methode der Inkrementierung arbeitet wesentlich schneller als die zweite.

Hinweis:

```
int  wert1=10,wert2;      /* Deklaration mit Initialisierung */

wert2 = ++wert1;  /* Erst inkrementieren, dann zuweisen wert2=11*/
wert2 = wert1++;  /* Erst zuweisen, dann inkrementieren wert2=10*/
```

Beispiel:

```
/*************************************************************/
/* Welche Ausgaben werden durch den Programmlauf erzeugt ?   */
/*************************************************************/
#include <stdio.h>

main()
{
 int  a=5,b=5,summe;   /* Deklaration und Initialisierung */
 char *format;         /* Pointerdeklaration */

 format = "a= %d,  b =%d,  summe = %d /n";

 summe = a   + b;   printf(format,a,b,summe);
 summe = a++ + b;   printf(format,a,b,summe);
 summe = ++a + b;   printf(format,a,b,summe);
 summe = --a + b;   printf(format,a,b,summe);
 summe = a-- + b;   printf(format,a,b,summe);
}
```

- **Zuweisungsoperatoren**

Eine erweiterte Variante des In- und Dekrementoperators sind die Zuweisungsoperatoren. Mit ihnen können Variablen beliebig manipuliert werden und zwar bei wesentlich kürzerer Ausführungszeit.

Allgemein lassen sich Ausdrücke der Form:
<variable1> = <variable1> <operator> <ausdruck>

verkürzt schreiben in der Form:
<variable1> <operator>= <ausdruck>

Beispiele:

```
zahl1 += zahl2    entspricht:     zahl1 = zahl1 + zahl2
zahl1 -= zahl2    entspricht:     zahl1 = zahl1 - zahl2
zahl1 *= zahl2    entspricht:     zahl1 = zahl1 * zahl2
zahl1 /= zahl2    entspricht:     zahl1 = zahl1 / zahl2
zahl1 %= zahl2    entspricht:     zahl1 = zahl1 % zahl2
```
Auf der rechten Seite können statt zahl2 beliebige Ausdrücke stehen.

- **Vergleichsoperatoren**

Operator	Aktion
>	größer als
>=	größer als oder gleich
<	kleiner als
<=	kleiner als oder gleich
==	gleich
!=	ungleich

Vergleichsoperatoren haben geringere Priorität als die arithmetischen Operatoren, z.B.

```
   x < y -1     bedeutet:     x < (y - 1)
```

- **Logische Operatoren**

Operator	Aktion
&&	AND
||	OR
!	NOT

Die Wirkungsweise der logischen Operatoren ist am besten anhand folgender Wahrheitstabelle erkennbar:

x	y	x AND y	x OR y	NOT x
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Beispiel:

```
/*******************************************************/
/* Gebe alle gerade Zahlen zwischen 1 und 100 aus      */
/* --> 0, 2, 4, 6, 8, 10, .....                        */
/*******************************************************/
#include <stdio.h>

main()
{
 int i;

 for (i=1; i=100; i++)
    if (!(i%2)) printf("%d%  ",i);
}
```

- **Adressoperatoren**

Standard C stellt zwei Operatoren für die Arbeit mit Zeigern (Pointern) und Adressen zur Verfügung.

&	Adresse von gibt die Adresse einer angegebenen Variablen zurück
*	Indirektion gibt den Inhalt einer Speicherzelle (Adresse) zurück

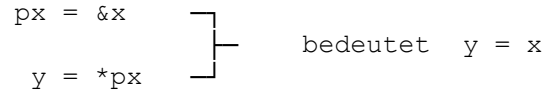
Ein Zeiger (=Pointer) ist eine Variable, die die Adresse einer Speicherstelle oder Speicherbereiches enthält, in dem der Wert einer anderen Variablen gespeichert ist.

Beispiel:

```
int  x, y;
int  *px;     /* verwende * um Zeigervariablen zu deklarieren */
              /* px ist ein Zeiger (=Pointer) auf eine integer-Variable */
```

Adresse des Speicherplatzes, in dem der Wert der Variablen gespeichert ist.

Mit der Anweisung px = &x wird die Adresse von x der Variablen px zugewiesen.
Mit der Anweisung y = *px wird der Variablen y der Inhalt der Speicherzelle zugewiesen, auf die px (=Adresse von Variable x) zeigt.



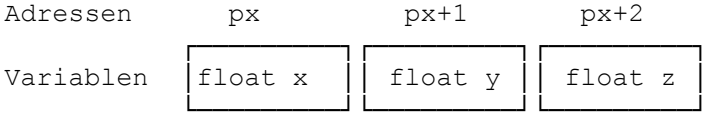
Hinweis:

Der "*"-Operator darf nur auf Pointervariablen und Konstanten angewendet werden.

Die Arithmetik von Zeigervariablen ist in beschränktem Umfang möglich. Es gelten folgende Regeln:

- Pointervariablen PVs (=Zeigervariablen) dürfen nur anderen PVs gleichen Typs zugewiesen werden.
- PVs, die keinen definierten Wert enthalten, sollte der Wert 0 zugewiesen werden.
- PVs dürfen inkrementiert, dekrementiert und um beliebige int-Werte erhöht oder vermindert werden.
- PVs dürfen voneinander subtrahiert werden, z.B. pz = py - px
- PVs dürfen miteinander verglichen werden.

Beispiel:



Beim Umgang mit Feldern ist eine etwas andere Betrachtungsweise in Bezug auf Zeiger und Arrays erforderlich. Hierauf wird jedoch näher im Kapitel Strings und Felder eingegangen.

Weiteres Beispiel für den Einsatz von Pointern:

```
#include <stdio.h>

main()
{
 int *zaehladr, zaehle, wert;

 zaehle = 100;

 zaehladr = &zaehle;    /* ermittle Adresse von zaehle */
 wert = *zaehladr;     /* ermittelt den Wert (Inhalt) der Adresse */
 printf("%d ",wert);   /* gibt 100 aus */
}
```

- **Übung:**

```
/*******************************************************/
/* Versuchen Sie das Programm nachzuvollziehen         */
/* Welche Werte werden ausgegeben ?                    */
/*******************************************************/
main()
{
 int index, *ptr1, *ptr2;

 index = 39;
 ptr1 = &index;
 ptr2 = ptr1;
 printf ("Die Werte lauten %d %d %d \n ", index, *ptr1, *ptr2);
 *ptr1 = 13;
 printf ("Die Werte lauten %d %d %d \n ", index, *ptr1, *ptr2);
}
```

- **Der Cast-Operator ()**

Wenn Variablen und Konstanten mit unterschiedlichen Datentypen (=Wertebereiche) in einem Ausdruck vorkommen, dann konvertiert Standard C automatisch alle Operanden aufwärts zum größten im Ausdruck vorkommenden Datentyp. z.B.

```
char   ch;
int    i;
float  f;
double d;

ergebnis = (ch / i)   +   (f * d)   -   (f + i);
            |   |         |   |         |   |
          int   |     double  |     double  |
            └─┬─┘         └─┬─┘         └─┬─┘
              int           double        double
               └──────┬──────┘             |
                    double                  |
                      └─────────┬──────────┘
                                |
```

double

Für eine explizite Umwandlung von Variablentypen (= casting) kann in Standard C der Cast-Operator "()" eingesetzt werden.

Die allgemeine Form lautet

(<Standard-Datentyp>) <ausdruck>

z.B.:

```
int  x=5;
  (float) x/2      /* konvertiere integer x nach float x */
                   /* Ergebnis -> 2.5  */
aber:
  (float) (x/2)    /* hier nur Integer-Division */
                   /* Ergebnis -> 2.0  */
```

Beispiel:

```
/*************************************************************/
/* Drucke x und alle x/3 Brüche der Zahlen von 1 bis 100     */
/* 0.666, 1.0, 1.333, 1.666, ......                          */
/*************************************************************/

#include <stdio.h>

main()
{int x;

 for (x=1; x<=100; ++x)
    printf("Quotient (%d / 3) ist: %f \n" ,i , (float) i/3);

}
```

Werden mehrere Operatoren in Ausdrücken verwendet, so ist besonders auf die Priorität der einzelnen Operatoren zu achten. Kommen gleichrangige Operatoren vor, so ist entscheidend, in welche Richtung die Operatoren ausgewertet werden. Da beides gravierende Auswirkungen auf das Ergebnis haben kann, wird nachfolgend eine Operatorenhierachrie angegeben. Die Priorität ist dabei von oben nach unten abnehmend !

Operator	**Richtung der Evaluierung**
()	von links nach rechts
! ~ ++ -- - Cast-Operator * &	von rechts nach links
* / %	von links nach rechts
+ -	von links nach rechts
<< >>	von links nach rechts
< > <= >=	von links nach rechts
== !=	von links nach rechts
&	von links nach rechts
|	von links nach rechts
&&	von links nach rechts
||	von links nach rechts
?:	von rechts nach links
= += -= *= /= %= Zuweisungsoperator	von rechts nach links
,	von links nach rechts

Klammerinhalte werden immer mit höchster Priorität behandelt. Bei geschachtelten Klammern wird von innen nach außen abgearbeitet. Falls unklar ist, wie der Compiler einen bestimmten Ausdruck evaluiert, so sind im Zweifelsfall immer Klammern zu setzen !

Beispiel:

```
   zahl1 = (zahl2 - zahl3 + zahl4) / 5
      |           |          |      |
      |           |          |      |
      |           |          |      |
Eval. |    zahl2 - zahl3     |      |
      |                  + zahl4    |
      |                           / 5
   zahl1 =
```

Übung:

Berechnen Sie die folgenden Ausdrücke:

```
wert = 0;
wert1 = ++wert;
zahl  = (23 + 9) / 4 + -5;
wert2 = 4 + ++wert;
wert2 = 4 + wert++
```

4.1.3 Beispiele zur Veranschaulichung der Folgestrukturen

```
/************* Uebung 00 : Ein erstes C-Programm ********************/
main()
{ printf("Dies ist mein erstes C Programm\n");
}


/*****************************************************************/
/* Uebung 01  zu Ausgabeanweisungen                              */
/*****************************************************************/
#include <stdio.h>   /* Angabe der Header-Datei, die Informationen zu */
                     /* den einzelnen Ein-/Ausgabe-Funktionen enthalten */
main()               /* Funktion: Hauptprogramm  */
{ printf("Falls der Platz in einer Zeile nicht "
       "ausreicht, kann einfach in der nächsten " /* Ausgabeanweisung */
       "weitergeschrieben werden !");    /* aus der Bibliothek stdio.h */
}


/*****************************************************************/
/* Uebung 02 zur formatierten Ausgabe Ausgabe des Textes im      */
/* Flattersatz                                                   */
/*****************************************************************/
#include <stdio.h>;  /* Angabe der Header-Datei, die Informationen zu */
                   /* den einzelnen Ein-/Ausgabe-Funktionen enthalten */
main()            /* Funktion: Hauptprogramm  */
{printf("Falls der Platz in einer Zeile nicht \n\r"
       "ausreicht, kann einfach in der nächsten \n\r"
```

```
              "weitergeschrieben werden \a");
}


/*****************************************************************/
/* Uebung 03  zur Ein- und Ausgabeanweisung                      */
/* Es soll ein Begrüßungsprogramm entworfen werden               */
/*****************************************************************/
#include <stdio.h>;
main()
{char name[30];       /* Variablendefinition */
 printf("Wie heißen Sie bitte ?  \a ");   /* Ausgabeanweisung */
 scanf("%s",name);                     /* Eingabeanweisung */
 printf("\n\n\t Hallo, %s, jetzt geht's los .\a.\a.\a.\a, name);
                                        /* Ausgabeanweisung */
}

/*****************************************************************/
/* Uebung 04  zur Ein-, Zuweisungs- und Ausgabeanweisung         */
/* Zwei INTEGER-Zahlen sollen multipliziert werden               */
/*****************************************************************/
#include <stdio.h>;

main()
{float  zahl1,zahl2, multi;       /* Variablendefinition */
 printf("Geben Sie bitte zwei Zahlen ein: \a");  /* Ausgabeanweisung */
 scanf("%f %f",&zahl1,&zahl2);                   /* Eingabeanweisung */
 multi = zahl1 * zahl2;                        /* Zuweisungsanweisung */
 printf("Der Quotient ist %f\n",multi);         /* Ausgabeanweisung */
}
```

4.1.4 Übungsaufgaben zu den Folgestrukturen

Übung 1: Minimalprogramm

Schreiben Sie ein Minimalprogramm, das die Ausgabe ihrer Adresse auf den Bildschirm realisiert.

Übung 2: Berechnung des durchschnittlichen Bezinverbrauchs

Das Programm soll nach Eingabe der gefahrenen Kilometer und der getankten Liter den Durchschnittsverbrauch pro 100 km ausrechnen. Es soll folgende Meldezeile ausgegeben werden:

```
   "Der Durchschnittsverbrauch des Kraftfahrzeuges liegt bei
              x ltr. / 100 km"
```

Übung 3: Berechnung des Umfangs und der Fläche eines Rechtecks

Das Programm soll den Umfang und die Fläche eines Rechtecks mit beliebigen Seitenlängen berechnen.

Übung 4: Darstellung eines Kreises auf dem Bildschirm

Das Programm soll Kreise an beliebigen Koordinaten(720*348 Punkte) des Bildschirms mit beliebiger Größe zeichnen.

Zur Lösung verwenden Sie folgende Standard C Grafikfunktion:

```
 clrscr()             ->  löscht den Bildschirm (Header-Datei conio.h)
 initgraph(int  far *GrafikTreiber,
           int  far *GrafikMode,
           char far *PfadzumTreiber)    initialisiert den Grafikmodus
 closegraph()         ->  beendet den GrafikTreiber  (graphics.h)
 restorecrtmode()     ->  schaltet von Grafik auf Text um  (graphics.h)
 circle(int x, int y, int radius)  ->  zeichnet einen Kreis  (grahics.h)
```

Beispiel einer Initialisierung der Herkuleskarte:

```
#include <graphics.h>

main()
{
 int GrafikTreiber=HERCMONO; GrafikModus = 0;

 initgraph(&GrafikTreiber, &GrafikModus, "");
 restorecrtmode()
 closegraph()
}
```

4.2 Die Auswahlstrukturen

Die Auswahlstruktur ist eine sehr häufig verwendete Konstruktion zur Steuerung des Programmablaufes. Sie stellt eine vorwärts verzweigende Programmstruktur dar, die dann eingesetzt wird, wenn der weitere Programmablauf von einer Bedingung abhängig gemacht werden soll.

4.2.1 Die if und if..else-Struktur

Die allgemeine Form der if-Struktur lautet:

```
if (Ausdruck)  Anweisung1;
   else  Anweisung2;
```

Gelangt der Standard C Compiler zu dieser Anweisung, so wird zuerst der Ausdruck (Bedingungsausdruck) ausgewertet. Ist dieser wahr (Ausdruck != 0), dann wird Anweisung1 bzw. Anweisungsblock1 ausgeführt, ist der Ausdruck falsch (Ausdruck =0), so wird alternativ der else-Zweig, also Anweisung2 bzw. Anweisungsblock2 ausgeführt.

Hinweis:

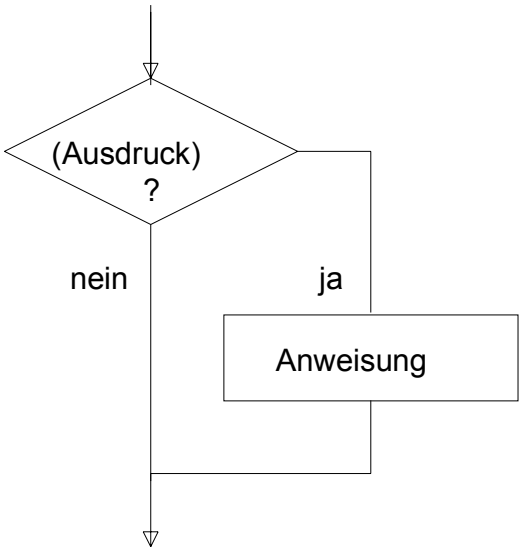
Im Gegensatz zu anderen Programmiersprachen (Pascal, etc.) kennt Standard C nicht den Standard-Datentyp BOOLEAN (true oder false). In Standard C werden diese Wahrheitswerte durch gewöhnliche int-Werte repräsentiert. Hierzu gilt:

```
Ausdruck = 0         -->  Ausdruck falsch (Bedingung nicht erfüllt)
Ausdruck ungleich 0  -->  Ausdruck  wahr  (Bedingung erfüllt)
```

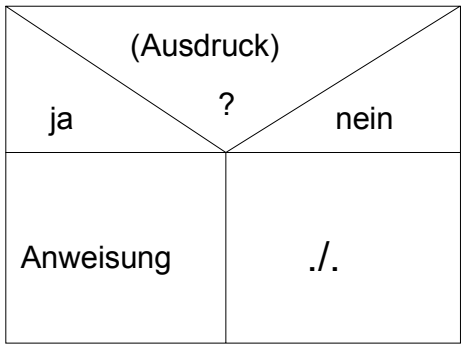
Zur Bildung des Wahrheitswertes werden üblicherweise die Vergleichsoperation herangezogen.

Die graphische Darstellung der if und if-else Struktur ist durch folgenden Graphen gekennzeichnet:

Flußdiagramm

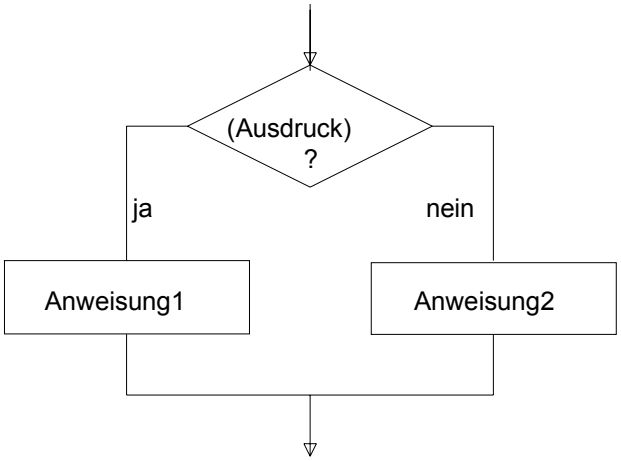


Struktogramm

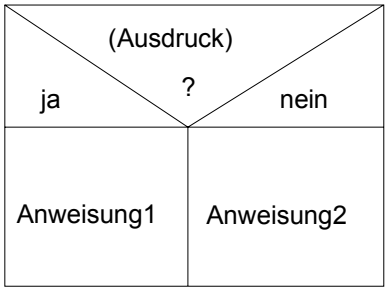


Der else-Zweig ist optional, d.h. er kann, falls er nicht benötigt wird, weggelassen werden. Man spricht in diesem Sonderfall auch von einer einseitigen Verzweigung.

Flußdiagramm



Struktogramm



Beispiele:

```
                  /* identisch mit */
 if (a<b)                                  if (a-b !=0)
    printf("Zahl a < b");                     printf("Zahl a < b");
  else                                      else
    printf("Zahl b <= a");                    printf("Zahl b <= a");


 if (a<b)
    if (a<3)
        printf("Zahl a ist kleiner als b und kleiner als 3");
  else        /* else-Zweig zur Bedingung (a<3) */
    printf("Zahl a ist kleiner als b, aber größer als 3");
```

Hinweis:

Fallunterscheidungen können beliebig verschachtelt werden. Fehlt jedoch in einer verschachtelten Folge von if-Anweisungen ein else-Zweig, dann wird der else-Zweig der vorhergehenden (letzten) if-Anweisung ohne else-Zweig zugeordnet.

else-if - Ketten

Für die Auswahl mehrerer Anweisungen unter mehreren Bedingungen können else-if-Ketten verwendet werden.

```
 if (Ausdruck1)
        Anweisung1;
 else if (Ausdruck2)
        Anweisung2;
 else if (Ausdruck3)
        Anweisung3;
```

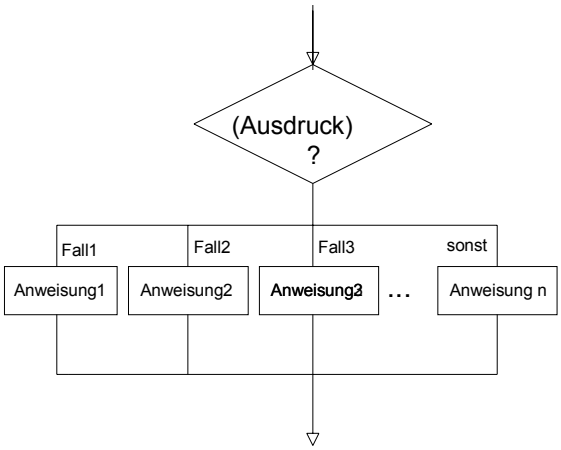
```
            :
 else
        Anweisung n;
```

Trifft keine der n-1-Bedingungen zu, so wird Anweisung n ausgeführt. Ist keine der vorhergehenden Bedingungen erfüllt und Anweisung n nicht erforderlich, so kann der optionale else-Zweig ganz weggelassen werden.

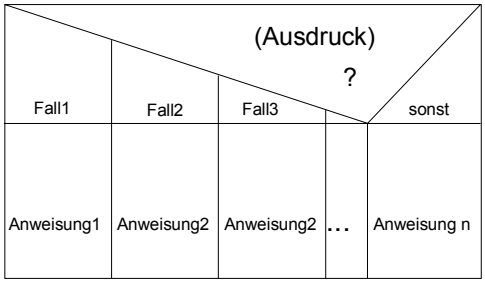
4.2.2 Die switch - Struktur

Die switch-Anweisung ist eine elegante Art unter mehreren Alternativen eine Auswahl zu treffen. In Pascal entspricht sie der case-Anweisung. Sie stellt eine Mehrweg-Verzweigung dar, in der geprüft wird, ob ein Ausdruck einen von mehreren konstanten Werten besitzt; ist dies der Fall, so wird entsprechend verzweigt.

Flußdiagramm



Struktogramm



Der als Selektor verwendete Ausdruck wird der Reihe nach mit jeder angegebenen CASE-Konstanten verglichen. Ist ein Vergleich wahr, so werden die nachfolgenden Befehle ausgeführt, und zwar solange, bis entweder ein break, oder das Ende von der switch-Kontrollstruktur erreicht wird.

Die allgemeine Form lautet

```
 switch (Ausdruck)
   { case Konst1:
           Anweisung1; break;
     case Konst2:
           Anweisung2; break;
          :
     default: Anweisung n;break;
   }
```

Beim Einsatz der switch-Kontrollstrukturen müssen folgende Punkte beachtet werden:

- (Ausdruck) ist eine Zeichenkonstante oder eine Ganzzahl (Typen: char, integer,Aufzählungstypen enum; jedoch kein float oder double)
- Der default-Zweig ist optional. Existiert kein default-Zweig und evaluieren alle Vergleiche zu false, so wird die gesamte switch-Anweisung übersprungen.
- Im Gegensatz zur case-Anweisung in Pascal ist in Standard C pro case-Marke nur eine Konstante erlaubt (Abhilfe siehe nächstes Beispiel).
- break dient zum Verlassen der switch-Kontrollstruktur, nachdem eine Alternative abgearbeitet ist. Jede case-Anweisung wird normalerweise mit break beendet.
- Ohne break wird direkt zur nächsten case-Marke gegangen, wobei diese nicht mehr neu ausgewertet wird. Es werden also solange Anweisungen nachfolgender case-Marken abgearbeitet, bis ein break oder das Ende der switch-Kontrollstruktur erreicht wird.

Beispiel:

```
switch (Ausdruck)          /* Ausdruck ist ganzzahlig */
  { case Konst_a1:
    case Konst_a2:       /* mehrere case-Anweisungen werden */
         :                           /*  ODER-verknüpft */
    case Konst_aN:
           Anweisung a1;
           Anweisung a2;
               :
           Anweisung aN;
         break;   /* Anw.block wird mit break abgeschlossen */
    case Konst_b1:
    case Konst_b2:
         :
    case Konst_bN:
           Anweisung b1;
           Anweisung b2;
               :
           Anweisung bN;
         break;
       :
    default:                /* default-Zweig ist optional */
           Anweisung c1;
           Anweisung c2;
               :
           Anweisung cN;
    break;
  }
```

4.2.3 Beispiel zur Veranschaulichung der Auswahlstrukturen

```
/*****************************************************************/
/* Das Programm konvertiert Zahlen in unterschiedliche Zahlensysteme */
/* und zwar von  dezimal nach hexadezimal, hexadezimal nach dezimal, */
/* dezimal nach oktal und  oktal nach dezimal.                     */
/*****************************************************************/

#include <stdio.h>
```

```
main()
{
 int waehle, wert;

 printf("Konvertiere: \n");
 printf("         1 : dezimal nach hexadezimal \n");
 printf("         2 : hexadezimal nach dezimal \n");
 printf("         3 : dezimal nach oktal \n");
 printf("         4 : oktal nach dezimal \n");
 scanf("%d",&waehle);

 if (waehle==1)
 {
  printf("Geben Sie eine dezimale ganze Zahl ein :");
  scanf("%d",&wert);
  printf("%d in hexadezimal ist: %x", wert, wert);
 }

 if (waehle==2)
 {
  printf("Geben Sie eine hexadezimale Zahl ein :");
  scanf("%x",&wert);
  printf("%x in dezimal ist: %d", wert, wert);
 }

 if (waehle==3)
 {
  printf("Geben Sie eine dezimale ganze Zahl ein :");
  scanf("%d",&wert);
  printf("%d in oktal ist: %o", wert, wert);
 }

 if (waehle==4)
 {
  printf("Geben Sie eine oktale Zahl ein :");
  scanf("%o",&wert);
  printf("%o in dezimal ist: %d", wert, wert);
 }
}
```

Bei Verwendung der switch-Konstruktion ist nicht nur die Ausführungszeit kürzer, sondern auch die Programmierung etwas eleganter. Um die Ausführungszeit zu reduzieren, werden die Ausdrücke mit abnehmender Wahrscheinlichkeit von oben nach unten angeschrieben.

```
/*****************************************************************/
/* Das Programm konvertiert Zahlen in unterschiedliche Zahlensysteme */
/* und zwar von  dezimal nach hexadezimal, hexadezimal nach dezimal, */
/* dezimal nach oktal und  oktal nach dezimal.                     */
/* Demonstration der switch-Kontrollstruktur                       */
/*****************************************************************/
#include <stdio.h>
```

```
main()
{
 int waehle, wert;

 printf("Konvertiere: \n");
 printf("         1 : dezimal nach hexadezimal \n");
 printf("         2 : hexadezimal nach dezimal \n");
 printf("         3 : dezimal nach oktal \n");
 printf("         4 : oktal nach dezimal \n");
 scanf("%d",&waehle);

 switch(waehle)
 {
  case 1:
  {
   printf("Geben Sie eine dezimale ganze Zahl ein :");
   scanf("%d",&wert);
   printf("%d in hexadezimal ist: %x", wert, wert);
   break;
  }
  case 2:
  {
   printf("Geben Sie eine hexadezimale Zahl ein :");
   scanf("%x",&wert);
   printf("%x in dezimal ist: %d", wert, wert);
   break;
  }
  case 3:
  {
   printf("Geben Sie eine dezimale ganze Zahl ein :");
   scanf("%d",&wert);
   printf("%d in oktal ist: %o", wert, wert);
   break;
  }
  case 4:
  {
   printf("Geben Sie eine oktale Zahl ein :");
   scanf("%o",&wert);
   printf("%o in dezimal ist: %d", wert, wert);
   break;
  }
 }  /* ende switch */
}  /* ende main */
```

4.2.4 Übungsaufgaben zu den Auswahlstrukturen

Übung 1: Maxima- und Minimabestimmung

Es sollen 4 Zahlen vom Typ integer deklariert und über Tastatur eingelesen werden. Anschließend soll durch IF-Fragen die größte (Maxima) und die kleinste (Minima) Zahl ermittelt und ausgegeben werden.

Übung 2: Benzinverbrauchsrechnung

Berechnung des durchschnittlichen Bezinverbrauchs pro 100 km.
Das Programm soll nach Eingabe der gefahrenen Kilometer und der getankten Liter den Durchschnittsverbrauch pro 100 km ausrechnen. Es soll folgende Meldezeile ausgegeben werden:

```
   "Der Durchschnittsverbrauch des Kraftfahrzeuges liegt bei
               x ltr. / 100 km"
```

Übung 3: Versicherungsprämienberechnung

Eine Schwachstromversicherung deckt alle Schäden an empfindlichen elektronischen Bürogeräten ab, die beispielsweise durch einen plötzlichen Stromausfall oder unsachgemäße Handhabung entstehen könnten.
Der Jahresbeitrag für die Versicherungssumme errechnet sich im Falle einer Einzelversicherung für Büromaschinen aufgrund der folgenden Vertragsbedingungen (Auszug):

Versicherungssumme in DM	Prämie in Prozent pro Jahr
bis 5000	16,8
von 5001 bis 10000	12,6
über 10000	8,4

- Frage: Welche Kontrollstruktur eignet sich hierfür am besten?
- Schreiben Sie ein Programm, das nach Eingabe der gewünschten Versicherungssumme die jährliche Versicherungsprämie berechnet.

Übung 4: Berechnung des Notendurchschnitts

Es soll der Notendurchschnitt einer Klasse berechnet werden. Dazu sollen beliebig viele Noten eingelesen und bei der Eingabe auf Korrektheit (Note zwischen 1 und 6) geprüft werden. Wenn für eine Note eine Null (0) eingegeben wird, soll die Eingabe beendet werden. Von allen Noten wird der Durchschnitt nach der Formel berechnet:

$$\text{Notendurchschnitt} = \frac{\text{Summe aller Noten}}{\text{Anzahl aller Noten}}$$

Übung 5: Mehrwertsteuerberechnung

Sie sollen die Netto-Preise eines Großhandels in Endbeträge (incl. 16% Mehrwertsteuer) umrechnen. Zur Erleichterung schreiben Sie sich ein Programm, das Ihnen nach der Eingabe des Nettopreises automatisch den Bruttopreis mit 2 Stellen hinter dem Komma ausgibt.

Zusatz: Die Mehrwertsteuer für Bücher und Zeitschrift beträgt nur 7 %. Erweitern Sie Ihr Programm so, dass nach Eingabe des Nettopreises und der Warenart der Bruttopreis alternativ berechnet wird.

Achten Sie auf die entsprechende Formatierung der Eingabe und Ausgabe-anweisungen.

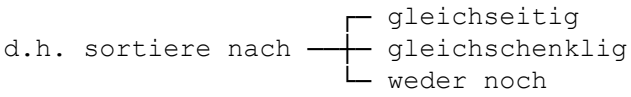
Übung 6: Zugangsberechtigung prüfen (Password-Schutz)

Die Computer-Viren breiten sich immer stärker aus. Um einer Infektion Ihres Computers vorzubeugen und unerlaubten Zugang zu Ihren Daten zu verhindern, soll ein Password-Schutz entwickelt werden.
Das Programm soll ein eingegebenes Wort mit dem im Programm vorgegebenen Codewort vergleichen. Wird das richtige Wort eingegeben, so soll eine Erfolgsmeldung und eine freundliche Begrüßung des Benutzers erfolgen. Werden jedoch drei falsche Codewörter eingegeben (geraten), so soll der unerwünschte Computerbenutzer darauf hingewiesen werden, dass er offensichtlich kein eingetragener USER ist. Alle weiteren Eingaben sollen anschließend ignoriert werden.

Übung 7: Bestimmung eines Dreiecks

Ein Programm soll die drei Seitenlängen eines Dreiecks a,b,c einlesen und anhand dieser Angaben feststellen, welche Besonderheit, in Bezug einer vereinfachenden Berechnung des Dreiecks, vorliegt.

Hinweis:



4.3 Die Wiederholungsstrukturen

Standard C stellt drei Kontrollstrukturen zur Wiederholung von Anweisungen oder Anweisungsblöcken zur Verfügung. Dazu gehören:
- die for-Schleife.
- die while-Schleife
- die do-while-Schleife

4.3.1 Die for - Schleife

Von den drei Schleifentypen for-, while- und do..while-Schleifen läuft die for-Schleife am schnellsten. Dies ergibt sich aus der Tatsache, dass bei der for-Schleife die Anzahl der Schleifendurchläufe im voraus berechnet wird. Die ermittelte Anzahl wird bei der Ausführung des Programmcodes auf Null heruntergezählt, was sich in Maschinensprache besonders einfach und schnell realisieren läßt. Sie wird deshalb häufig auch als Zählschleife oder als iterative Schleife bezeichnet.

Der formale Aufbau lautet:

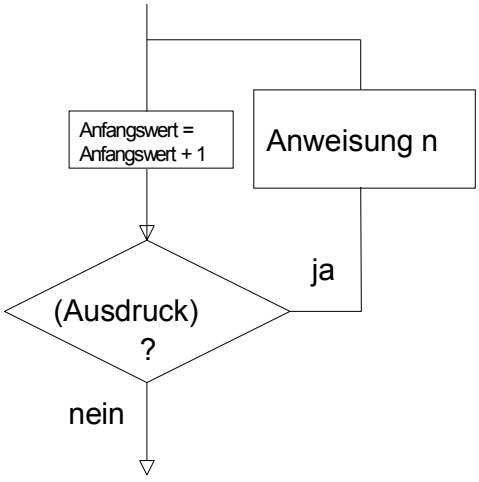
```
for (Ausdruck_1; Ausdruck_2; Ausdruck_3)
  Anweisung_n;
```

oder:

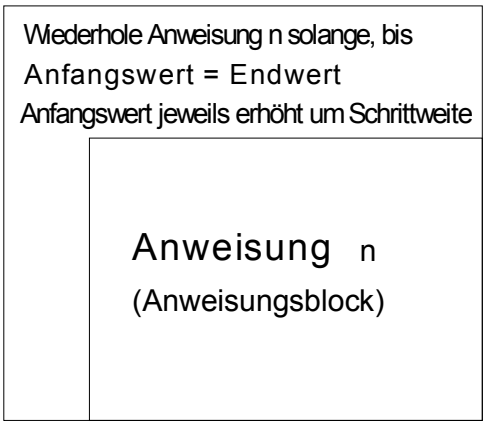
```
for (Ausdruck_1; Ausdruck_2; Ausdruck_3)
{
 Anweisung_1;
 Anweisung_2;
     :
 Anweisung_n;
}
```

Im allgemeinen wird die for-Schleife dann eingesetzt, wenn die Anzahl der Durchläufe vor Eintritt in die Schleife bekannt ist.

Flußdiagramm



Struktogramm



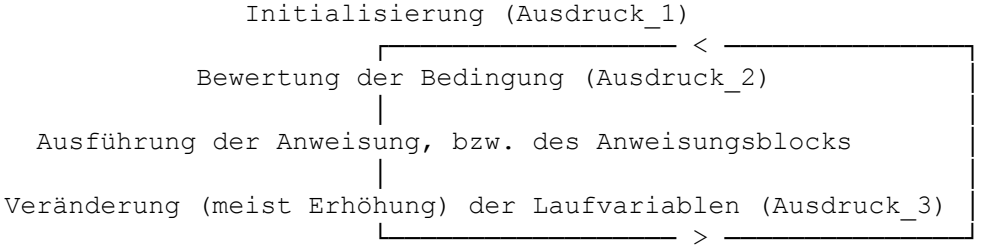
Die for-Schleife enthält drei Ausdrücke, die durch Semikolon voneinander getrennt sind:

Der erste Ausdruck dient der Initialisierung, d.h. der Variablen (Laufvariable) wird ein Startwert zugewiesen.

Der zweite Ausdruck beinhaltet eine Laufbedingung für den Endwert der Laufvariablen. Hier wird geprüft, ob die Schleife ein weiteres Mal wiederholt werden soll. Erhält Ausdruck_2, den Wahrheitswert *false* (Ausdruck_2=0), wird die for-Schleife beendet und die nachfolgende Anweisung ausgeführt.

Im Ausdruck_3 (Schrittausdruck) wird die Laufvariable manipuliert . Meistens enthält dieser Ausdruck eine Zuweisung zur In- oder Dekrementierung der Laufvariablen.

Nachstehend ist die **Evaluation** der for-Schleife angegeben:



Beispiel:

```
for (zahl=1; zahl<1000; zahl++)
      printf("%d \n" , zahl);
```

Obige Konstruktion ist äquivalent zu folgender while-Konstruktion:

```
Ausdruck_1;
while (Ausdruck_2)
   { Anweisung_1;
     Anweisung_2;
         :
     Anweisung_n;
     Ausdruck_3;
   }
```

Hinweise:

- Der Wert der Laufvariablen bleibt erhalten, wie auch immer die Schleife beendet wird.
- Die Grenze der for-Schleife kann aus der Schleife heraus verändert (beeinflußt) werden.
- Fehlt der Ausdruck_2, dann wird der Wahrheitswert true angenommen; das Ergebnis ist eine Endlosschleife (auch unendliche Schleife genannt). In diesem Fall muss die for-Schleife durch eine explizite Abbruchanweisung, wie z.B. break oder return beendet werden.

Die Definition der for-Schleife ist sehr offen. So besteht die Möglichkeit, auch mehrere Aktionen, jeweils durch ein Komma getrennt, in einem Ausdruck zu kombinieren.

Beispiel:

```
char  klein,gross;      /* Initialisierung der  Variablen */

for (klein='a', gross='A';  /* Zuw. Laufvariable = Startwert */
     klein<='z';            /* Eine Abbruchbedingung genügt! */
     gross++, klein++)      /* Verändere Laufvariable */
 {
  printf("%c%c\n", gross,klein); /*Ausg. Groß-u.Kleinbuchstab.*/
 }            /* Klammern hier nicht unbedingt erforderlich */
```

Im Beispiel wird das komplette Alphabet in Groß- und Kleinbuchstaben ausgegeben. Ausdruck_1 und Ausdruck_3 sind kombinierte Ausdrücke mit mehreren Aktionen. Innerhalb von Ausdruck_3 werden die Variableninhalte, das ist der ASCII-Wert des Zeichens, jeweils um 1 erhöht.

4.3.2 Die while - Schleife

Die while-Schleife stellt die Grundstruktur aller Wiederholungs-Konstrukte von C dar und kann somit die for-Schleife wie auch die do..while-Schleife vollständig, jedoch weniger elegant, ersetzten.
Die while-Konstruktion wird immer dann zur Programmablaufsteuerung eingesetzt, wenn Probleme ohne Initialisierung und Reinitialisierung (im Gegensatz zur for-Konstruktion) implementiert werden sollen.

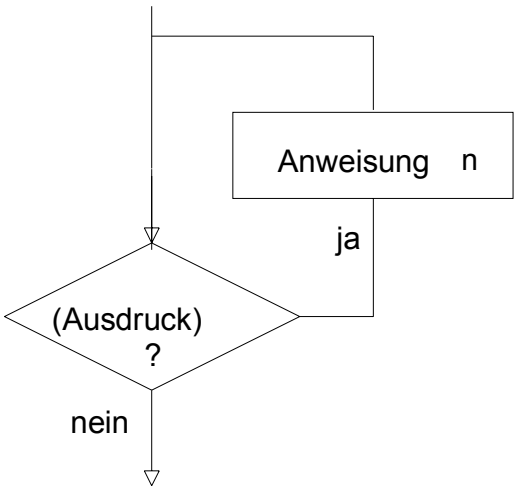
Der formale Aufbau ist folgender:

```
while (Ausdruck)
{
 Anweisung_1;
 Anweisung_2;
    :
 Anweisung_n;
};
```

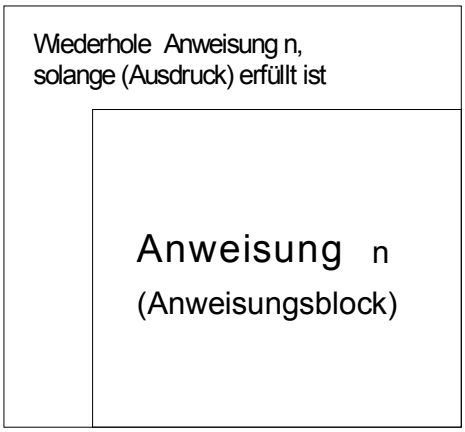
bzw. für eine Anweisung:

```
while (Ausdruck)
          Anweisung;
```

Flußdiagramm



Struktogramm



Die while-Schleife wird auch als abweisende Schleife bezeichnet, da vor dem Eintritt in die Schleife bzw. in den Anweisungsblock die Eintrittsbedingung geprüft wird. Überdies wird vor jedem Schleifendurchlauf neu geprüft, ob eine weitere Wiederholung erfolgen soll. Diese Form der Repetition heißt prechecked loop.

Zuerst wird der Ausdruck ausgewertet. Ergibt sich ein Wert ungleich 0 (Ausdruck ist true), so wird die Anweisung bzw. der Anweisungsblock ausgeführt. Anschließend wird der Ausdruck erneut auf den Wahrheitswert hin überprüft.

Die Anweisung oder der Anweisungsblock wird solange wiederholt, bis der Ausdruck einen Wert gleich 0 annimmt (Ausdruck ist false). Die Programmausführung wird mit der Anweisung nach der while-Schleife fortgesetzt.

Beispiel:

```
i=1;
while (i<=100)     /*Wiederhole solange i kleiner gleich 100)*/
{
 printf("\n    DM  %3d,- \t", i);
 printf("\n Dollar %2.2f \t", i/dollar);
 printf("\n  Lira  %6.2f \t", i/lira*1000);
 i++;             /*inkrementiere i*/
}
```

Hi**nweise:**

- Vor Eintritt in die Schleife wird der Ausdruck, die sog. Eintrittsbedingung geprüft. Damit es an dieser Stelle nicht zu Fehlern kommt, muss die Variable im Ausdruck unbedingt vorher initialisiert, d.h. einem Startwert zugeordnet, werden (im obigen Beispiel: i=1).
- Die Variable im Ausdruck (Schleifenvariable) muss innerhalb des Anweisungsblockes reinitialisiert, d. h. verändert werden, andernfalls würde die Schleife unendlich oft ausgeführt (Endlosschleife).
- Fehlt Ausdruck in der Kopfzeile, dann liegt ebenfalls eine Endlosschleife vor, z. B. while (1).

Beispiel:

```
Warte_auf_Char()                  /* Definition einer Funktion*/
{
 char ch;

 ch = '0';                       /* Initialisierung der Schleifenvariable */
 while ((ch!='Q') && (ch!='q'))  /*Warte bis Taste Q gedrückt*/
        ch = getche();           /* Anweisung */
}
```

4.3.3 Die do..while - Schleife

Bei der for- und while-Konstruktion wird die Bedingung zur Wiederholung der Schleife im Schleifenkopf berechnet und dann gegebenenfalls die Anweisung bzw. der Anweisungsblock ausgeführt. Ergibt die Schleifenbedingung von vornherein false (Ausdruck gleich 0), werden die Wiederholungskonstrukte ganz übersprungen.

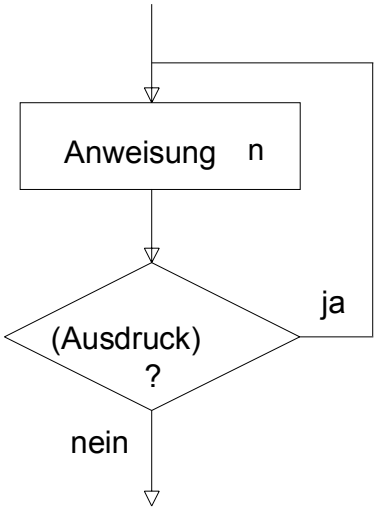
Anders bei der do..while-Konstruktion. Hier wird zuerst die Anweisung bzw. der Anweisungsblock ausgeführt und dann erst geprüft, ob eine Wiederholung der Schleife nochmals erfolgen soll. Sie wird deshalb auch als nichtabweisende Schleife bezeichnet.

Der formale Aufbau lautet:

```
do
{
 Anweisung_1;
 Anweisung_2;
    :
 Anweisung_n;
} while (Ausdruck);           /* Abbruchbedingung */
```

Die do..while-Schleife kommt dort zum Einsatz, wo die Schleifenanweisungen mindestens einmal abgearbeitet werden sollen, z. B. bei der Menüauswahl oder bei Wiederholungen des Programmablaufs. Insgesamt jedoch wird diese Wiederholungsschleife weitaus seltener als die anderen Schleifenkonstrukte verwendet.

Flußdiagramm



Struktogramm

Erst nach Ausführung der Anweisung bzw. des Anweisungsblockes wird der Ausdruck geprüft. Ist er true (Ausdruck ungleich 0), wird der Anweisungsteil wiederholt. Ergibt der Ausdruck false, (Ausdruck gleich 0), wird der Programmablauf mit der Anweisung nach der do..while-Schleife fortgesetzt.

Beispiel:

```
 do        /* Wiederhole, falls Zahl a innerhalb 1..9 liegt */
 {
  printf("Eingabe: ");
  scanf("%d", &a);
 } while (a>=1 && a<=9);
```

Hinweise:

- Die do..while-Schleife wird wiederholt, solange der Ausdruck true (Ausdruck ungleich 0) ergibt, d.h. die Schleife endet auf false (Ausdruck gleich 0). Im Gegensatz zu Pascal; dort endet die repeat..until-Schleife, wenn die Bedingung true (wahr) wird.
- Relationale Operatoren (">" und "<") haben in Standard C eine höhere Priorität als die logischen Operatoren "&&" und "||". Die logische Kombination zweier Vergleiche kommt daher ohne zusätzliche Klammer aus, z.B. while (a>=1 && a>=h).
- Da auf jeden Fall der Anweisungsblock mindestens einmal durchlaufen wird, entfällt die bei der while-Konstruktion erforderliche Initialisierungsanweisung zum Eintritt in die Schleife.

while-Schleife	do..while-Schleife
Initialisierung; *while (Bedingung)* *Anweisung;*	*do Anweisung* *while (Bedingung);*

Beispiel:

```
do
{
  scanf("%d", zahlA);
  wert= wert + zahlA;
} while (wert <= KONST);
```

4.3.4 Beispiele zur Veranschaulichung der Wiederholungsstrukturen

```
/*********************************************************************/
/* Das Programm liest einen Text über Tastatur ein und gibt ihn zen-  */
/* triert auf dem Monitor aus.                                        */
/* Demonstration der while-Schleife und des Funktionskonzepts         */
/*********************************************************************/
#include <stdio.h>
#include <string.h>

main()
{char Text[255];                    /* Variablendeklaration */
 int  laenge;
```

```
 printf("Geben Sie ein Textzeile ein: ");
 gets(Text);                          /* Eingabe */
 Zentriere(strlen(Text));             /* Zentrierung und Ausgabe */
 printf(Text);
}

/*** Funktion zur Ermittlung und Ausgabe der Leerzeichen ***/
Zentriere(int Laenge)
{
 laenge= (80-laenge)/2;

 while(laenge>0)
 {
  printf(" ");
  laenge--;
 }
}



/*********************************************************************/
/* Das Programm konvertiert Zahlen in unterschiedliche Zahlensysteme */
/* und zwar von  dezimal nach hexadezimal, hexadezimal nach dezimal, */
/* dezimal nach oktal und  oktal nach dezimal.                       */
/* Demonstration der switch-Struktur und der do..while-Schleife      */
/*********************************************************************/
#include <stdio.h>
main()
{int waehle, wert;
 do
 {
  printf("Konvertiere: \n");
  printf("         1 : dezimal nach hexadezimal \n");
  printf("         2 : hexadezimal nach dezimal \n");
  printf("         3 : dezimal nach oktal \n");
  printf("         4 : oktal nach dezimal \n");
  printf("         5 : quit\n");
  printf("Bitte treffen Sie eine Auswahl: ");
  scanf("%d",&waehle);
 } while (waehle<1 || waehle>5); /* entspricht !(waehle>=1 && waehle<=5) */

 switch(waehle)
 {
  case 1:
  {
   printf("Geben Sie eine dezimale ganze Zahl ein :");
   scanf("%d",&wert);
   printf("%d in hexadezimal ist: %x", wert, wert);
  }
  case 2:
  {
   printf("Geben Sie eine hexadezimale Zahl ein :");
   scanf("%x",&wert);
   printf("%x in dezimal ist: %d", wert, wert);
  }
  case 3:
  {
```

```
   printf("Geben Sie eine dezimale ganze Zahl ein :");
   scanf("%d",&wert);
   printf("%d in oktal ist: %o", wert, wert);
  }
  case 4:
  {
   printf("Geben Sie eine oktale Zahl ein :");
   scanf("%o",&wert);
   printf("%o in dezimal ist: %d", wert, wert);
  }
 }  /* ende switch */
}  /* ende main */


/*********************************************************************/
/* Das Programm testet Ihr Zeitgefühl. Nachdem Sie eine Taste gedrückt*/
/* haben, sollen Sie 5 Sekunden warten (abschätzen) und schließlich   */
/* wieder eine Taste drücken.                                         */
/* Demonstration der for-Schleife und der break-Anweisung             */
/*********************************************************************/
#include <stdio.h>
#include <time.h>
#include <conio.h>
main()
{long zeit;

 printf("Dieses Programm testet Ihr persönliches Zeitgefühl !\n");
 printf("Wenn Sie bereit sind, bitte eine Taste drücken, 5 Sekunden
          warten\n");
 printf("und nochmals eine Taste drücken: ");
 getche();
 printf("\n");

 zeit=time(0);
 for (;;)                 /* Endlosschleife */
   if (kbhit())  break;   /* Durch bel. Taste wird Endlosschleife abgebrochen */

 if (time(0)-zeit==5)
     printf("Prima! Sie haben es getroffen !!");
   else printf("Sie liegen leider daneben ");
}
```

4.3.5 Übungsaufgaben zu den Wiederholungsstrukturen

Übung 1: Maximabestimmung

Von einer zuvor eingegebenen Zahlenreihe mit maximal 20 Elementen soll das größte (Maxima) und das kleinste (Minima) Element bestimmt werden.

Übung 2: Berechnung der Fakultät

Schreiben Sie ein Programm, das die Fakultät einer Zahl berechnet. Die Zahl soll über Tastatur vorher eingelesen werden.

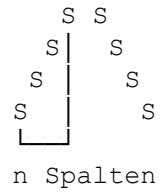
Übung 3: Erzeugung einer Tannenbaumgrafik

Es soll eine Grafik erzeugt werden, die durch mehrfache Aneinanderreihung des folgenden Grundelementes entsteht:

```
     xxxxxxx          N1 = Anzahl der "x" in der obersten Zeile
    xxxxxxxxx
   xxxxxxxxxxx        N2 = Anzahl der "x" in der untersten Zeile
```

Über Eingabeaufforderung sollen die Werte für N1 und N2 eingelesen werden. Dabei sind folgende Bedingungen einzuhalten:
- N1 und N2 müssen ganzzahlig und ungerade sein
- N1 und N2 müssen in den Grenzen zwischen 1 und max.
  (Spaltenzahl des Ausgabemediums) liegen
- Erlaubt ist hingegen, dass N1 größer als N2 ist.

Übung 4: Schreiben Sie ein Programm, das folgendes Druckbild erzeugt:



Die Zahl n ist online während des Programmlaufes einzugeben. Schreiben Sie das Programm nach Möglichkeit so, dass nach dem Start die Entscheidung getroffen werden kann, ob das Bild auf dem Drucker oder auf dem Bildschirm erscheinen soll.

Übung 5: Tilgung und Verzinsung eines Darlehens

Zur Tilgung und Verzinsung eines Darlehens D wird am Ende eines Jahres eine feste Verzinsung A (Annuität) geleistet.

Dieser Betrag A teilt sich auf in
a) Zinsen Z = D * P /100 der jeweiligen Restschuld D vor Abzug
b) Tilgung T = A - Z

Nach Abzug der Tilgung ergibt sich eine neue Restschuld D = D - T
Das Programm soll folgenden Anforderungen genügen:
Eingaben : - Darlehen D
           - Zinsfuß P
           - Annuität A

Während der Eingabe soll geprüft werden, ob eine Darlehenstilgung mit der eingegebenen Annuität möglich ist.
Ausgaben : - Tabelle der Form :

Zahl der Jahre	Zins	Tilgung	Restschuld
.....	....	.....	.....

- Histogramm der Form :

Zahl der Jahre	Restschuld in Prozent
0	################################
1	#########################
2	##########
....	.....

4.4 Strings- und Felder (Arrays)

Im Kapitel 3 wurde ein Überblick der Datentypen gegeben, die unter Standard C verfügbar sind. Bisher wurden in diesem Skript vorwiegend die einfachen Datentypen, wie z.B. char, int, usw. verwendet. Im folgenden soll nun eine Unterweisung in die strukturierten Datentypen (zusammengesetzten Datentypen) erfolgen. Standard C unterscheidet hier zwischen den Feldern (=Arrays) und dem Verbund (=Struktur).

4.4.1 Integer-Arrays

Als Array wird eine Ansammlung von Variablen mit gleichem Datentyp bezeichnet. Die Variablen nehmen die Daten auf und heißen Array-Elemente; der Zugriff erfolgt über eine weitere ganze Zahl, dem sog. Index.

Die Syntax eines eindimensionalen Arrays lautet:

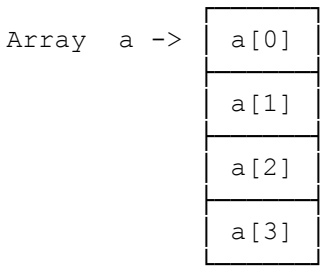
Datentyp Array_Variable[Array_Größe]

Beispiel:

```
main()
{
 int Tabelle[10];             /* Array-Deklaration */
 int t;

 for (t=0; t<10; ++t)        /* t ist Laufvariable und Index */
     Tabelle[t]=t;           /* Array-Zuweisung */
}
```

Hinweise:

- Der Index zum Zugriff auf bestimmte Array-Elemente beginnt in Standard C immer bei 0 und endet bei N-1, wobei N für die Größe des Arrays steht (Array_Größe).
- In Standard C wird nicht geprüft ob der Indexwert größer als die in der Deklaration vereinbarte Array_Größe ist.
- Arrays können schon bei der Deklaration initialisiert werden, z. B.:
  `int Tabelle[10] ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`
- Die Anzahl der Elemente (bzw. Komponente) ist statisch, d. h. nicht veränderbar.
- Array-Bezeichner sind Pointerkonstanten auf das 1. Element des Arrays, d.h. für nachstehendes Beipiel: a ist die Startadresse für einen reservierten Speicherbereich.



Der Wert des Array-Bezeichners kann nicht verändert werden, da es sich um eine Pointerkonstante handelt (a=a+1 ist nicht erlaubt).

Wei**teres Beispiel:**

```
int feld[10];
zeiger = &feld[0];  /* Zwei identische Zuweisungen für die */
zeiger = feld;      /* Array-Startadresse */
```

Beispiel:

```
main()
{
    /*Array-Deklaration und Initialisierung der ersten 5 Elemente*/
 int Tabelle[10] = {1000, 100, 50, 75, 250};
 int i, Schnitt;

 for (i=5; i<10; ++i)         /*Lies Werte von Element 5 bis 9*/
 {
  printf("Gib Wert ein für Tabelle[%d] :", i);
  scanf("%d", &Tabelle[i]);
 }

 Schnitt=0:                   /*Berechne Durchschnitt*/
 for (i=0; i<10; ++i)         /* i ist Laufvariable und Index */
     Schnitt=Schnitt+Tabelle[i];

 printf("Gesamtdurchschnitt: %d\n", Schnitt/10);
}
```

Die Syntax eines mehrdimensionalen Arrays lautet:

Datentyp Array_Variable[Array_Größe_1][Array_Größe_2] ...

Beispiel:

```
int matrix[3][5]=  /*Deklaration des multidimensionalen Arrays*/
{
 { 1,2,3,4,5 },     /* Innere Klammer sind optional /
 { 3,6,2,1,2 },     /* 2-dimensionales Feld mit 3 Zeilen und 5 Spalten */
 { 8,7,6,9,1 },
};

matrix[i][t]=0;    /*Zugriff auf das multidimensionale Array*/
```

Obige Matrix besteht aus 3 Datenfeldern zu je 5 Einträgen vom Datentyp Integer. Jedes Element muss mit Hilfe von zwei Indizes angesprochen werden. Das niedrigste Feldelement ist matrix[0][0] ,das höchste Element wird mit matrix[2][4] angesprochen. In Standard C ist es jedoch durchaus erlaubt, mit matrix[4][5] auf Daten außerhalb der Array-Deklaration zuzugreifen, da der Compiler keine Bereichsüberprüfung vornimmt!

4.4.2 Zeichen-Ketten (Strings)

Eine Zeichenkette, im folgenden als String bezeichnet, ist ein eindimensionales Array (Feld) vom Datentyp char.

Beispiel:

```
char str[7]="AAAAAAA";          // *(str)   ist char-Variable
                                // *(str)   ident. str[0]
main()                          // *(str+1) ident. str[1]
{                               // *(str+i) ident. str[i]
 int i;
 for (i=0; i<7; i++)
         str[i]='A'+i;         // ident.: *(str+i) = 'A'+i;
}
```

vorher:

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]
A	A	A	A	A	A	A	\0

nachher:

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]
A	B	C	D	E	F	G	\0

Hinweise:

- Strings können nur außerhalb von Funktionen initialisiert werden, z.B. char str[6]="STANDARD-C"
  main()
  {
  }

- Ein String wird durch Anführungszeichen eingeschlossen; im Gegensatz dazu wird ein Zeichen (char) durch Hochkommata begrenzt.

- Mann kann ein String-Element, analog zum Array-Element, entweder über seinen Index str[3], oder über den Pointer = Vektor *(str+3) ansprechen.

- Das Ende eines Strings wird durch das sog. Null-Byte \0 (ASCII-Code 0) gekennzeichnet. Ein String mit 10 Zeichen Länge benötigt somit 11 Bytes.

- Zur Ein- und Ausgabe von Strings stehen in Standard C spezielle Funktionen zur Verfügung. Diese heißen puts(..) und gets(..).

Beispiele:

```
char Zeichkette[3];    /* Deklaration eines Strings */

zeichkette[0]='D';     /* Zuweisung */
zeichkette[1]='O';
zeichkette[2]='S';
zeichkette[3]='\0';

puts(zeichkette);   /* Ausgabe eines Zeichenstrings */

char str[81];

printf("Geben Sie ihr Password ein :");
gets(str);           /* liest einen String von der Tastatur */

if (strcmp(str,"EDV-DOZENT"))       /* = 0 -> Strings ident. */
  puts("Kein Zugang - ungültiges Password !); /*Stringausgabe*/
else
  puts("Guten Tag, Viel Spaß bei Ihrer Arbeit !");
```

- Da Standard C aufgrund der speziellen Zeigerverwaltung beim Umgang mit Strings keine direkten Zuweisungen erlaubt, verfügt der Compiler über eigene Funktionen zur Stringmanipulation.

Funktion	**Aufgabe**	**Beipiel**
stpcpy(char *String1, char *String2)	Kopiert String2 nach String1	char str[9]='2.0'; strcpy(str,"TURBOC");

strcat(char *String1, char *String2)	Fügt String2 an String1 an	char str[20];
		strcpy(str,"C++");
		strcat(str,"BORLAND");

stpcmp(char *String1, char *String2)	Vergleicht String1 mit String2
	i>0 String1 ist größer
	i<0 String2 ist größer
		int i;
		i=strcmp("Win",Win");
		if(i==0) puts("beide sind gleich");

strlen(char *String)	Ermittelt Länge von String	int i; i=strlen("StandardC"); Ausgabe: 6
strlwr(char *String)	Konvertiert alle Großbuch-staben zu Kleinbuchstaben	puts(strlwr("C++"); Ausgabe: C++
strlup(char *String)	Konvertiert alle Großbuch-staben zu Kleinbuchstaben	puts(strlup("Standard-C")); Ausgabe: Standard-C

4.4.3 String-Konstanten

Eine sehr hilfreiche Möglichkeit zur Realisierung textintensiver Ausgaben (z.B. Fehlermeldungen, Hinweise zum Programmablauf) bietet Standard C durch den Einsatz von String-Konstanten.

Beispiel:

```
main()
{ char *error;              /* Definition einer String-Konstanten */

  error =" Fehler 01: "     /* Belegung */
         " Bitte beachten Sie den zulässigen Wertebereich !"

  printf("%s", error)       /* Ausgabe einer String-Konstanten */
}

oder:

static char *error[]=       /* Vereinbarung eines Zeigerfeldes */
                            /* Anzahl der Zeiger gemäß der Initialisierung */

 {
  "Fehler 01: Bitte beachten Sie den zulässigen Wertebereich !",
  "Fehler 02: Es sind nur Buchstaben von A..Z, a..z erlaubt !",
  "Fehler 03: Unerlaubter Zugriff auf externes Speichermedium !",
  "Fehler 04: Fehler in der Zuweisung von Stringkonstanten !"
 };
}
  printf("%s", error[1])    /* Ausgabe der Fehlermeldung-Nr. 1 */
  printf("%s", error[2])    /* Ausgabe der Fehlermeldung-Nr. 2 */
  printf("%s", error[4])    /* Ausgabe der Fehlermeldung-Nr. 4 */
}
```

4.4.4 Beispiele zur Veranschaulichung der Array- und Stringoperationen

```
/*************************************************************/
/* Programm liest einen String über Tastatur ein und gibt in     */
/* vertikal wieder aus.                                          */
/* Parameterübergabe -> call by reference                        */
/*************************************************************/
#include <stdio.h>

void  Druck_vertikal(char *ZeichKette);   /* Angabe des Prototyps */

main()          /* Hauptfunktion */
{ char EinStr[30];

 Druck_vertikal("EDV-DOZENT");          /* Funktionsaufruf */

 gets(EinStr);
 Druck_vertikal(EinStr);                /* Funktionsaufruf */
}

void  Druck_vertikal(char *strng);   /* Funktionsdefinition */
```

```
/*****************************************************************/
/* Funktion druckt einen String vertikal aus                     */
/*****************************************************************/
{while (*strng)
   printf("\n %c", *strng++);    /*Stringadresse wird um 1 erhöht */
}


/*****************************************************************/
/* Programm liest einen String über Tastatur, wandelt Groß- in   */
/* Kleinbuchstaben um und gibt diesen wieder aus.                */
/* Kommunikation über  call by reference                         */
/*****************************************************************/
#include <stdio.h>

main()
{ char  s[80];

 printf("Gib einen Text ein :");
 gets(s);                       /* Stringeingabe */
 Klein(s);
 printf("Text klein : %s ", s);
}

Klein(char *ct)       /* String in Kleinbuchstaben umwandeln */
{ int i;
  for (i=0; ct[i]; ++i)
  {
    if (ct[i] >= 'A' && ct[i] <= 'Z')
    ct[i] = (ct[i] + 'a'- 'A');
  }
 }
```

4.4.5 Übungsaufgaben zu den Array- und Stringoperationen

Übung 1: Suchen innerhalb eines Feldes (Arrays)

In einer sequentiellen Liste von 1-byte-Daten stehe irgendwo das Zeichen "%". Die Anzahl der Elemente soll maximal 256 betragen. Schreiben Sie ein Programm, das feststellt, an welcher Stelle relativ zum Anfang sich dieses Element befindet.

Übung 2: Sortieren eines Zahlenfeldes

Auf einem Lochstreifen sind positive Zahlenwerte - durch RETURN, LINE FEED getrennt - gestanzt. Das Ende ist durch eine Null gekennzeichnet.

a) Lesen Sie die positiven Werte (und nur diese !) in ein eindimensionales Feld ein.
b) Nun werde angenommen, das Feld enthalte 100 Elemente. Es soll nach aufsteigender Größe sortiert werden. Dies macht man so, dass man je zwei benachbarte Elemente a,b vertauscht, wenn sie noch nicht in der gewünschten Beziehung a <= b stehen.

Den Vorgang wiederholt man solange, bis das komplette Feld geordnet ist. Schreiben Sie hierfür ein Standard C Programm.

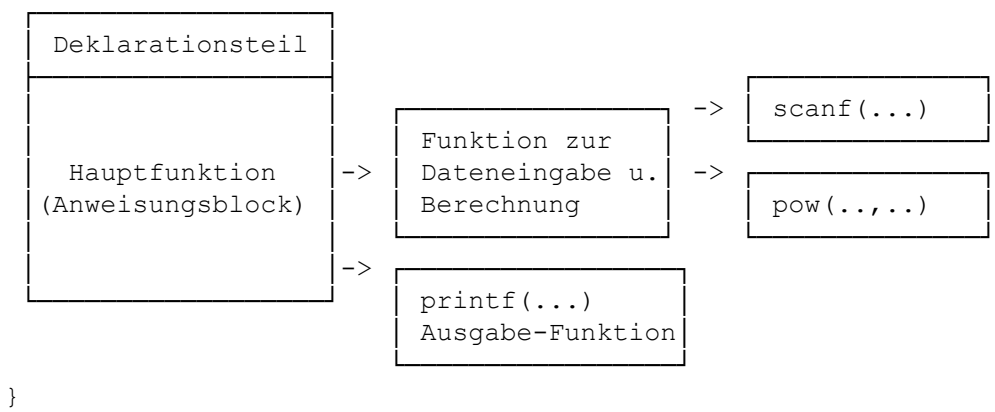
4.5 Die Unterprogrammstrukturen

Wie im Kapitel 4 schon erwähnt, besteht ein Programm in Standard C aus einzelnen Funktionen. Ein Minimalprogramm benötigt dabei nur die Hauptfunktion main(), z.B.

```
 main()
   {
   }        /* Minimalprogramm in Standard C */
```

Eine Funktion faßt gewöhnlich mehrere Anweisungen zusammen, wobei eine Anweisung auch ein Funktionsaufruf sein kann. Selbstverständlich können alle Anweisungen eines Programms in die Hauptfunktion main() geschrieben werden. Bei einem umfangreichen Programm ist es aus Gründen der leichteren Fehlersuche und der Übersichtlichkeit jedoch sinnvoll, eine Unterteilung des Gesamtprogramms in überschaubare, funktional zusammengehörende Teilaufgaben vorzunehmen. Jede Teilaufgabe wird nun in einer Funktion kodiert, mit einer eindeutigen Import- und Export-Schnittstelle zu den anderen Funktionen. Mit dem Funktionskonzept unterstützt Standard C in besonderer Weise den modularen Programmaufbau und die strukturierte Programmierung.

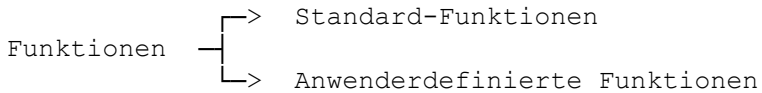
Die Grafik macht dies deutlich:



Im Vergleich zu anderen Programmiersprachen ist der Befehlsvorrat von Standard C recht bescheiden. Dies erleichtert nicht nur das Erlernen dieser Sprache, sondern erhöht auch die Portabilität, d.h. die Übertragbarkeit des Quellcodes auf andere Rechnersysteme. Die vielfältigen Möglichkeiten unter Standard C ergeben sich erst mit Hilfe der zahlreichen Funktionen (mehr als 300 !), die in den verschiedenen Funktionsbibliotheken zur Verfügung stehen.

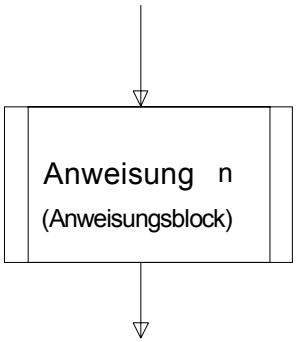
4.5.1 Funktionen

Im Umgang mit Funktionen werden prinzipiell zwei Arten unterschieden:

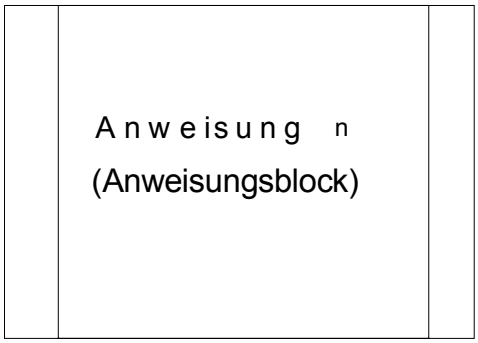


Standard-Funktionen werden von Bibliotheken bereitgestellt, die beim Kauf des Compilers oder des Entwicklungssystems mitgeliefert werden. Dazu gehören z.B. die Ein- und Ausgabefunktionen scanf(..) bzw. printf(..) oder die Bildschirm-funktion clrscr(), die den Bildschirm löscht.
Dennoch sind diese Bibliotheken offen, d. h. sie können um beliebig viele anwenderdefinierte Funktionen erweitert werden oder es können gänzlich neue erstellt werden. In diesem Zusammenhang spricht man auch von Funktions-Moduln.

Flußdiagramm



Struktogramm



Die allgemeine Form einer Funktion lautet:

```
Resultat_Datentyp  Funktions_Name(Parameterdeklarationen)
{
  Anweisung_1;
  Anweisung_2;     /* Funktionskörper (Anweisungsblock) */
      :
  Anweisung_n;
}
```

Programmtechnisch wird eine Funktion durch einen Funktionskopf und einen Funktionskörper festgelegt. Der Funktionskörper enthält die Anweisungen, die ihrerseits wiederum Funktionsaufrufe darstellen können. Im Funktionskopf wird angegeben, von welchem Datentyp der zurückgelieferte Funktionswert ist. Weiterhin wird der Funktions_Name definiert und im Anschluß daran folgt eine in Klammern (..) eingeschlossene, optionale Parameterliste.

Beispiel:

```
float Division (int Divisor, int Dividend)
/* Liste von Parametern durch Kommata getrennt; kann leer sein*/

{
         /* Deklaration der in der Funktion zusätzlich zu den Para- */
                                  /* -metern verwendeten Variablen */
                          /* Anweisungen nur innerhalb der Funktion */
 return (Divisor/Dividend)                                 /* optional */
}
```

Die Parameter im Funktionskopf werden formale Parameter genannt. Sie legen die Importschnittstelle fest, d.h. sie definieren die Eingangsgrößen, die die Funktion bei Ihrem Aufruf erwartet. Formale Parameter stellen Platzhalter für aktuelle Parameter dar, die beim Funktionsaufruf (z.B. aus dem Hauptprogramm) an die aufgerufene Funktion übergeben werden.

Beispiel:

```
void Reversi(char *s)
{
 register int t;     /*t wird in ein Prozessorregister geladen*/

 for (t=strlen(s)-1; t>-1; t--)
                  printf("%c", s[t]);
}
```

Obige Funktion Reversi gibt eine importierte Zeichenkette rückwärts wieder aus. Sie stellt eine reine Ausgabefunktion dar und liefert folglich keinen Wert an das aufrufende Programm zurück. Reversi ist vom Datentyp void zu definieren.

Hinweise:

- Funktionen sind globale Objekte, d.h. eine Schachtelung von Funktionen, wie in Pascal, ist nicht möglich.

- Ist die Parameterliste leer, so müssen trotzdem die runden Klammern () gesetzt werden.

- Eine Funktion vom Typ void liefert nie einen Wert zurück. (Sie verhält sich wie eine Prozedur in Pascal). Die Verwendung der return-Anweisung führt zum Abbruch der Funktion, ohne dass eine Wertübergabe stattfindet.

- Soll ein Funktions-Wert zurückgeliefert werden, wird hierzu die return-Anweisung verwendet. Achtung: Die return-Anweisung beendet auch die Abarbeitung der Funktion.

- Bei der Wertübergabe mit der return-Anweisung ist unbedingt darauf zu achten, dass der Typ des Ausdrucks *return(Ausdruck)*- mit dem Resultattyp der Funktion *int Feldminimum(..)*- übereinstimmt. Falls nicht, wird in Richtung Resultattyp transformiert.

Beispiel:

```
int Wert1()      /* Resultattyp ist Integer */
{
 return(11.11); /* Fehler! Nur der Wert 11 wird übergeben */
}
```

Beim Einsatz von Funktionen muss klar zwischen ihrer Definition und der Deklaration unterschieden werden.

- In der Definition wird die Implementierung der Funktion beschrieben; dies sind die eigentlichen Anweisungen, aus denen sich die Funktion zusammensetzt und die beim Funktionsaufruf abgearbeitet werden.

- Die Deklaration einer Funktion hingegen sorgt dafür, dass diese innerhalb anderer Funktionen aufgerufen werden kann. Sie teilen dem restlichen Programm mit, dass eine Funktion des angegebenen Namens existiert, die bestimmte Parameter beim Aufruf erwartet. Deklarationen erzeugen keinen zusätzlichen Maschinencode, geben jedoch dem Compiler die Möglichkeit eine Parameterprüfung bei Funktionsaufrufen durchzuführen. Wichtig: Eine Deklaration endet im Gegensatz zur Definition immer mit einem Komma !

Beipiel:

```
#define LAENGE 80
int Feldminimum(int feld[], int a);       /* Funktionsdeklaration */
                /* Angabe des Prototyps der Funktion Feldminimum */

main()                                          /* Hauptprogramm */
{int  min;
 int  matrixlne[LAENGE];
 min=Feldminimum(matrixlne,LAENGE);         /* Funktionsaufruf */
}                                          /* aktuelle Parameter */


int Feldminimum(int feld[], int a)     /* Funktionsdefinition */
             /*ident. mit: int Feldminimum(int *feld, int a) */
                                        /* formale Parameter */
/************************************************************/
/* Funktion ermittelt das Minimum in einem Integer-Array,   */
/* int feld[]  -> ist das übergebene Array                  */
/* int  a   -> Anzahl der Array-Elemente                    */
/* Hinweis: Durch die Deklaration feld[] können Integer-Arrays */
/* beliebiger Größe importiert werden !                     */
/************************************************************/
{int  minimum, i;

 minimum=feld[0];
 for (i=1; i<a; i++)
    if (feld[i] < minimum)
           minimum = feld[i];

 return(minimum);
}
```

Werden Funktionen aus anderen Modulen benutzt, sollten diese im aufrufenden Programm vor ihrem Aufruf als sog. Prototypen (Funktionsdeklaration) angegeben werden. Nur dann unternimmt der Compiler eine Typüberprüfung der Parameter.

4.5.2 Kommunikation zwischen Funktionen und aufrufendem Programm

Zum Austausch von Informationen unter Funktionen bzw. zwischen Funktionen und aufrufendem Programm gibt es in Standard C drei Möglichkeiten:

- Die return-Anweisung
- Verwendung von globalen Variablen
- die Parameterübergabe beim Funktionsaufruf

Die ersten beiden Optionen wurden im Kapitel 4.5.1 behandelt, hier soll nun näher auf die dritte Möglichkeit eingegangen werden.

In der Kopfzeile einer Funktion kann optional eine Parameterliste mit deren Typangaben (Objektdeklaratorliste) stehen. Als Typen kommen alle Standarddatentypen (int, double,..), Aufzählungstypen, Zeigertypen und Strukturtypen in Frage.

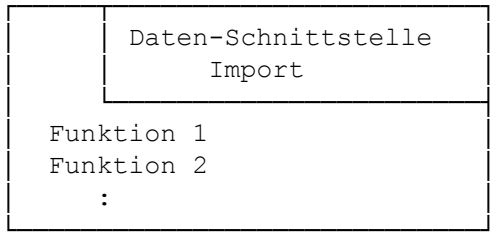
Die im Funktionsaufruf anzugebenden Parameter werden als aktuelle Parameter bezeichnet. Sie müssen in Typ, Reihenfolge und Anzahl den Vereinbarungen der formalen Parametern entsprechen.

Es gibt zwei unterschiedliche Arten der Parameterübergabe.

- Wertübergabe (call by value)
- Adressübergabe (call by reference)

Wertübergabe (call by value)

Diese Methode der Parameterübergabe kopiert die Werte der aktuellen Parameter des Funktionsaufrufes nach den formalen Parametern der aufgerufenen Funktion. Veränderungen der Parameter innerhalb der Funktion haben keinen Einfluss auf die aktuellen Parameter des rufenden Programms. Die Wertübergabe findet nur in einer Richtung statt, d.h. es werden nur Daten in die Funktion importiert (Import-Schnittstelle).

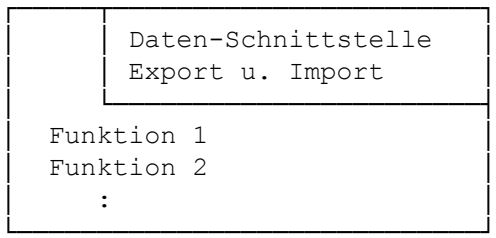


Beispiel:

```
#include <stdio.h>
Druckeaus(int zahl1,int zahl2,double mult)  /* Ausgabefunktion */
{ printf("Das Produkt von %d und %d ergibt: %f", zahl1,zahl2,mult)
}

main()                    /* Hauptfunktion */
{int  zahl1,zahl2;
 double  mult;

 scanf(&zahl1, &zahl2);
 mult = zahl1 * zahl2;

 Druckeaus(zahl1, zahl2, mult);
}
```

Adressübergabe (call by reference)

Bei call by reference werden dagegen die Adressen der aktuellen Parameter an die formalen Parameter übergeben und nicht die Werte selbst. Die formalen und die aktuellen Parameter müssen daher als Zeiger auf Variablen des entsprechenden Typs vereinbart werden. Da die aufgerufene Funktion mit den Adressen der aktuellen Parameter (Variablen) arbeitet, kann sie die Werte der aktuellen Parameter auch ändern. Die Kommunikation findet in beiden Richtungen statt, d.h. es werden Daten in die Funktion importiert und aus der Funktion exportiert (Import- und Export-Schnittstelle).



Beispiel:

Eine typische Anwendung ist die beim Sortieren von Zahlenreihen erforderliche Vertauschung zweier Elemente.

```
#include <stdio.h>

void  Tausche(int *x, int *y)          /* Funktionsdefinition */
                   /* x und y Vereinbarung als Zeiger auf int */
{ int  merke;

 merke = *x;  /*speichere Inhalt der Adresse nach Variable merke*/
 *x = *y;       /* Inhalt der Adresse y -> Inhalt der Adresse x */
 *y = merke;              /* Wert merke -> Inhalt der Adresse y */
}

main()                /* Hauptprogramm */
{ int  a=10;
  int  b=15;

 printf("Originalwerte von a und b: a= %d, b= %d \n", a, b);
 Tausche(&a, &b);
 printf("Originalwerte von a und b: a= %d, b= %d \n", a, b);
}
```

Hinweise:

- Bei call by reference ist die Übergabe von festen Werten (Konstanten) nicht erlaubt.

- Die Bezeichner (Variablennamen) der aktuellen und formalen Parameter können verschieden sein. Allein die Reihenfolge und die Typen der Parameter sind für die Datenübergabe entscheidend.

- Falls ein Array oder String einer Funktion übergeben werden soll, wird nicht das gesamte Feld kopiert, sondern nur die Anfangsadresse des Arrays oder Strings übergeben. Die Anfangsadresse ist die Adresse auf das erste Feldelement feld[0]). Das bedeutet, dass aktuelle und formale Parameter kompatibel zum Zeigertyp (Pointertyp) sein müssen.

Beispiel:

```
Display(int *num)   /* formale Parameterliste */
                    /* ident. mit:  Display(int num[]),  */
                    /* ident. mit:  Display(int num[10]) */
                    /* Obige formale Feldangaben werden zu integer-Pointer */
                    /* konvertiert */
{ int i;
```

```
 for (i=0; i<10; i++)   printf("%d " , num[i];
}

main()                      /* Hauptprogramm */
{ int tab[10];

 Display(tab);    /*aktuel. Parameter = Adresse des Feldes*/
}
```

4.5.3 Beispiele zur Veranschaulichung der Unterprogrammstrukturen

```
/*****************************************************************/
/* Berechnung von Summe und Durchschnitt einer Wertetabelle      */
/* Anwendung aller Kontrollstrukturen                            */
/*****************************************************************/
int Werte_Tab[3];         /* Definition: Globale Werte-Tabelle */


int  Einlesen();
/*****************************************************************/
/* Funktion liest Werte über Tastatur ein und übergibt diese an  */
/* die globale Wertetabelle.                                     */
/*****************************************************************/
{
 int  i;         /* Definition: Lokale Laufvariable */

 for(i=1; i<=3; i++)
 {
  printf("Wert-Nr. %d \t", i);
  scanf("%d", &Werte_Tab[i-1]);   /*Tabelle beginnt bei Index 0 */
  if (Werte_Tab[i-1]==-1)         /* Abbruch bei -1 */
      return(i-1);             /* letzte Eingabe bleibt unberücksichtigt */
 }
 return(i-1);
}


 int  Rech_und_Ausgabe(int zaehler);
/*****************************************************************/
/* Funktion berechnet die Summe und den Durchschnitt             */
/* des Integer-Arrays Werte_Tab und gibt die Ergebnisse aus.     */
/*****************************************************************/
{
 double  summe=0.0;
 double  durchschnitt;
 int     i;

 for(i=0; i<zaehler; i++)
   summe+ = Werte_Tab[i];       /* Summe ermitteln */
 durchschnitt=summe/zaehler;
 printf("\nEs wurden %d Werte eingegeben", zaehler);
 printf(""\nSumme: %f , Mittelwert: %f", summe,durchschnitt);
}
```

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

 zaehler = Einlesen();    /*Anzahl der einzulesenen Werte */
 Rech_und_Ausgabe(zaehler);
}
```

4.5.4 Übungsaufgaben zu den Unterprogrammstrukturen

Übung 1: Zylindervolumen berechnen
Schreiben Sie eine Funktion, die das Zylindervolumen berechnet. Die erforderlichen Angaben radius (rad) und Höhe (hoehe) werden über die Parameterliste in die Funktion übergeben.

Übung 2: Umwandlung Groß- in Kleinbuchstaben
Schreiben Sie eine Funktion, die einen Großbuchstaben in einen Kleinbuchstaben umwandelt.

Übung 3: Matrizen-Multiplikation
Für nachstehende Problemstellung ist ein Standard C Programm zu entwerfen.

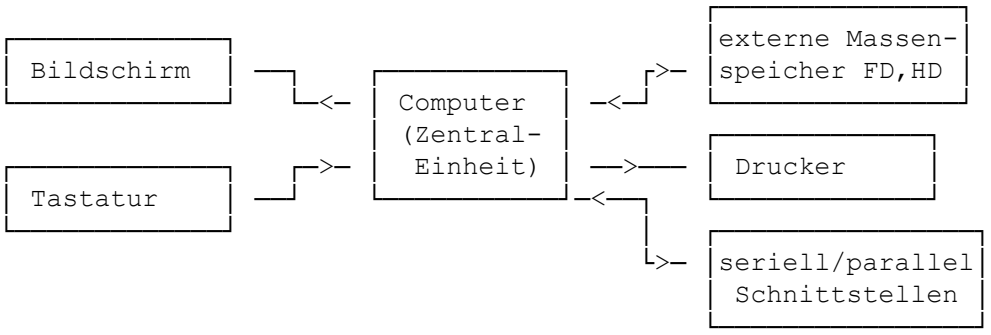
```
Gegeben:  Matrix A,
          Matrix B

Gesucht:  Matrix C = A + B
          Matrix D = A * B
```

Die Dimensionen der Matrizen A und B sind identisch und sollen innerhalb der vorgesehenen Größe frei wählbar sein.

Ausgabe: Matrizen A, B, C und D
Die Ein- und Ausgabe soll jeweils durch eigenständige Funktionen realisiert werden. Die Berechnung der Matrizenaddition, bzw. -multiplikation erfolgt in einer weiteren Funktion.

5. Die Dateioperationen unter Standard C

Eine leistungsfähige Datenverarbeitungsanlage (DV) verfügt über vielfältige Möglichkeiten der Datenein- und ausgabe. Nach dem in der EDV vorherrschenden EVA-Prinzip (Dateneingabe -> Datenverarbeitung -> Datenausgabe) findet die eigentliche Datenverarbeitung in der Zentraleinheit (oft als Prozessor bezeichnet) des Computers statt. Alle an der Zentraleinheit angeschlossenen Peripheriegeräte dienen entweder der Dateneingabe, z. B. Tastatur oder der Datenausgabe, z. B. Bildschirm, Drucker.
Eine Sonderstellung nehmen die externen Massenspeicher ein, sie gehören sowohl der Gruppe der Dateneingabe- als auch der Datenausgabegeräte an. Während im Arbeitsspeicher (RAM) die Daten nur temporär gespeichert sind und nach dem Ausschalten des Rechners unwiederbringlich verloren gehen, handelt es sich bei den externen Speichern um permanente - meist magnetische - Datenträger mit hoher Speicherkapazität.



Beabsichtigt man, die Ergebnisse seines mühevollen Schaffens der Nachwelt zu erhalten, müssen die Daten im Arbeitsspeicher auf ein externes Speichermedium ausgelagert, d.h. gespeichert bzw. gesichert werden. Unter dem Betriebssystem Windows werden die Daten immer in einer Datei abgelegt. Jede Datei hat einen Dateinamen, über den die Daten gespeichert und wieder aufgefunden, d. h. gelesen werden können.

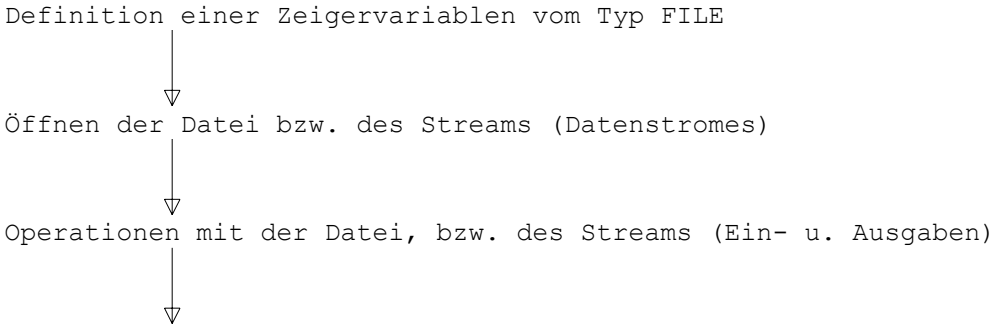
Standard C stellt aus Gründen der Portabilität keine Dateiverwaltungsbefehle zur Verfügung. Die Dateiverwaltung wird ausschließlich unter Zuhilfenahme von speziellen Ein- und Ausgabefunktionen der Standardbibliothek durchgeführt. Sie muss deshalb grundsätzlich bei Dateioperationen am Anfang des Quellprogrammes eingebunden werden.

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

In Standard C existieren zwei Arten von Dateiverwaltungsfunktionen:
- die gepufferte Dateisystem (Ein- und Ausgabe über Streams = Datenstrom) und das
- ungepufferte Dateisystem

Das ungepufferte Dateisystem entstand bei der Implementierung der C-Compiler auf Unix Systemen. Da sich das gepufferte und das ungepufferte Dateisystem recht ähnlich sind, wird nach neuen Normungsbestrebungen des ANSI Komitees in Zukunft nur noch das gepufferte Dateisystem verwendet (erweitert um die Bearbeitung von Text- und Binärdateien) und auf die ungepufferte Dateiverwaltung verzichtet. Ersteres soll nun näher betrachtet werden.

Die nachstehende Grafik verdeutlicht den Einsatz des gepufferten Dateisystems:



- Definition einer Zeigervariablen vom Typ FILE

Im Programm wird auf die Datei über einen Zeiger vom Typ FILE zugegriffen:

```
  FILE  *filevar      /* filevar ist logischer Dateiname */
```

FILE ist ein strukturierter Datentyp, der in der Headerdatei stdio.h definiert ist. Mit Hilfe des FILE-Datentyps verwaltet die Standardbibliothek die Bearbeitung einer Datei.

- **Öffnen der Datei bzw. des Streams (Datenstromes)**

Bevor Lese- und Schreiboperationen auf eine Datei ausgeführt werden können, muss sie geöffnet werden. Hierbei findet eine Zuordnung der ein- und ausgehenden Daten zu der vorher im Programm definierten Zeigervariablen (logischen Dateivariable) statt. Die Eröffnung erfolgt über die Funktion fopen(..), die gleichzeitig einen Schreib-/Lesepuffer (im Arbeitsspeicher RAM) einrichtet. Die eigentliche Datenübertragung erfolgt physikalisch jedoch erst dann, wenn der Puffer voll ist, oder explizit die Funktion fclose(..) aufgerufen wird:

```
  filevar = fopen(filename, mode)
  char  *filename            /* physikalischer Name der Datei */
  char  *mode                /* Eröffnungsmodus */
```

Der Variablen filevar wird ein Zeiger auf die Datei filename zugewiesen. Alle weiteren Dateioperationen geschehen von jetzt an über filevar.
mode ist ebenfalls ein String und legt den Eröffnungsmodus fest.

Modus	**Erklärung**
"r"	Öffnen zum Lesen
"w"	Öffnen zum Schreiben
"a"	Öffnen zum Erweitern
"r+"	Öffnen zum Schreiben und Lesen Die zu eröffnende Datei muss bereits bestehen
"w+"	Öffnen zum Schreiben und Lesen; Eine bereits bestehende Datei wird überschrieben
"a+"	Öffnen zum erweitern, Besteht die Datei noch nicht, so wird diese erst angelegt; zusätzlich können weitere Parameter angegeben werden:
"b"	eröffne als Binärdatei
"t"	eröffne als Textdatei (default-Modus)

Bei der Bearbeitung von Dateien unterscheidet Standard C nochmals zwei Modi:

Text: Die von einem Stream gelesene Zeichenfolge CR/LF (ASCII 13 und 10) werden automatisch auf ein einzelnes LF reduziert. Umgekehrt wird bei Schreibaktionen statt eines einzelnen LF die Zeichenfolge CR/LF gesetzt.

Binär: Es findet hier keine Umwandlung von LF in CR/LF und umgekehrt statt.

Beipiel:

```
filevar = fopen("ROHDATEN.DAT","r+t")
```

Die physikalische Datei ROHDATEN.DAT wird dem Zeiger filevar zugeordnet, und gleichzeitig für den Lesezugriff ("r") geöffnet; die Datei wird im folgenden wie eine Textdatei behandelt. Tritt ein Fehler bei der Dateieröffnung auf, z.B. wenn die angegebene Datei nicht existiert, dann erhält filevar den Wert NULL.

Beispiel:

```
#include <stdio.h>

main()
{char filename[25];
 FILE *filevar;

 printf("Gebe Dateiname an: ");
 scanf("%s",filename);  /* Liest den gewünschten Dateinamen */
 if ((filevar=fopen(filename,"r"))==NULL)
   printf("Fehler bei der Dateieröffnung");
 else
   printf("Die Datei %s wurde eröffnet.\n", filename);
}
```

Eine Besonderheit der Sprache Standard C ist die Handhabung der Peripheriegeräte: Jedes externe Gerät wird als Datei betrachtet, d.h. Ein- und Ausgaben in und von Dateien werden genauso behandelt wie von der Tastatur oder auf dem Bildschirm. In der Bibliothek stdio.h von Standard C sind sehr oft benötigte Streams (Standardkanäle) bereits vordefiniert. Beim Start eines Standard C Programms werden diese automatisch eröffnet.

Name	**Modus**	**Erklärung**
stdout	Text	Standardausgabe Bildschirm
stdin	Text	Standardeingabe Tastatur
stdprn	Text	Standardausgabe auf Drucker
stdaux	Binär	Standardein- und ausgabe, serielle Schnittstelle
stderr	Binär	Standardausgabe für Fehlermeldungen

- **Operationen mit der Datei, bzw. des Streams (Ein- u. Ausgaben)**

Beim Aufruf von Dateifunktionen muss immer die in fopen(..) gesetzte Zeigervariable *fp als Argument mit angegeben werden.

Übersicht der wichtigsten Dateioperationen:

```
fputc(char Zeichen, FILE *fp)     // Gibt ein Zeichen auf den Stream, falls
                                  // erfolgreich, wird Wert >0 zurückgegeben, sonst -1

fgetc(FILE *fp)                   // Liest ein Zeichen vom Stream; das
                                  // Zeichen ist vom Typ int

fputs(char *String, FILE *fp)     // übergibt Zeichenkette zum Stream;
                                  // wenn Ausgabe erfolgreich 0; sonst -1

fgets(char *String,               // Liest Zeichenkette mit Länge von Stream
      int Länge, FILE *fp)        // Rückgabe: Zeiger auf Zeichenkette

fprintf(FILE *fp,                 // wie Funktion printf(..); jedoch erfolgt
        char *Formatstring,       // formatierte Ausgabe auf Datei
        void Argumente)

fscanf(FILE *fp,                  // wie Funktion scanf(..); jedoch wird von
        char *Formatstring,       // Stream gelesen; Rückgabe ist die Anzahl
        void Argumente)           // der gelesenen Argumente

ferror(FILE *fp)  // prüft ob eine Dateioperation erfolgreich ausgeführt wurde;
                  // falls Fehler auftreten, wird ein Wert <>0 zurückgegeben,
                  // sonst wird 0 zurückgegeben. Die Funktion sollte nach jeder
                  // Dateioperation aufgerufen werden)

remove(char *filename)   // löscht eine Datei physikalisch; falls
                         // erfolgreich wird 0 zurückgegeben, sonst ein Wert <>0
```

Allgemein erfolgt der Zugriff auf eine geöffnete Datei zum Lesen bzw. Schreiben sequentiell, d. h. es werden alle Daten der Reihe nach ausgegeben. Speziell für eine formatierte Aus- und Eingabe in Textdateien stellt Standard C die Funktionen fprintf(..) bzw. fscanf(..) zur Verfügung.

Soll der Zugriff frei wählbar auf bestimmte Stellen in der Datei erfolgen (wahlfreier Zugriff: Random-Access), so kann hierzu die Position des Dateizeigers bewegt werden, ohne das gelesen oder geschrieben wird.

```
feof(FILE *fp)    // prüft, ob Ende der Datei (EOF) erreicht
                  // wenn Ende erreicht wird ein Wert <> 0
                  // zurückgegeben, sonst 0

ftell(FILE *fp)   // ermittelt die Position des Dateizeigers
                  // in der Datei

fseek(FILE *fp,            // positioniert den Dateizeiger neu
      long int Adresse     // Adresse= Anzahl der Bytes, um die der
      int mode)            // Zeiger bewegt werden soll:
                           // pos. für vorwärts, neg. für rückwärts;
                           // mode gibt die Relation an:
                           // 0 -> rel. zum Dateianfang,
                           // 1 -> rel. zur aktuellen Position
                           // 2 -> rel. zum Dateiende

rewind(FILE *fp)   // setzt die Position des Dateizeigers auf
                   // den Anfang der Datei
```

Die Funktionen fread(..) und fwrite(..) sind neu im gepufferten Dateisystem und wurden erst durch das ANSI Komitee definiert.

```
int fread(char *puffer, int groesse, int anzahl, FILE *fp);
```

Die Funktion fread(..) liest von der Datei, spezifiziert in fp, eine bestimmte Datenmenge in den Arbeitsspeicher des Computers. Die Startadresse der Daten wird durch den Pointer puffer angegeben. Die Größe der zu lesenden Datenmenge durch die Variablen anzahl und groesse vorgegeben.

Beispiel:

Es sollen vier float-Zahlen nach puffer gelesen werden. Der Funktionsaufruf muss demnach lauten:

```
fread(puffer,sizeof(float),4,fp);
```

Das Funktionsergebnis ist die Anzahl der wirklich gelesenen Daten.

Die Umkehrung, d.h. das Schreiben in eine Datei erledigt die Funktion fwrite(..). Die Angaben haben dieselbe Bedeutung wie fread(..). Zurückgegeben wird die Anzahl der wirklich geschriebenen Daten. Diese kann bei Erreichen des Dateiendes kleiner sein als die Anzahl im Funktionsaufruf.

```
int fwrite(char *puffer, int groesse, int anzahl, FILE *fp);
```

- **Schließen der Datei bzw. des Streams**

Nach der Benutzung müssen Dateien wieder geschlossen werden. Dies geschieht entweder bei der Programmbeendigung oder, wenn die maximale Zahl der gleichzeitig zu öffnenten Dateien erreicht ist (wird unter MS DOS in CONFIG.SYS festgelegt: z.B. FILES=10). Die Funktion fclose(..) leert den Datenpuffer des Streams und schließt dann die Datei. Sie gibt den Wert 0 zurück, wenn die Datei erfolgreich geschlossen werden konnte, sonst wird der Wert -1 zurückgegeben.

```
  filecount = fclose(filevar);       /* vorher: FILE *filevar */
```

Werden viele Dateien während des Programmlaufes geöffnet, so können diese allesamt mit der Funktion fcloseall(..) geschlossen werden. Die Funktion übergibt zudem die Anzahl der geschlossenen Dateien.

```
  filecount= fcloseall(filevar);
```

5.1 Beispiele zur Veranschaulichung der Dateioperationen

```
/*********************************************************************/
/* Liest eine Datei und gibt sie zeilenweise auf dem Monitor aus.    */
/* Demonstration der Funktion fgets(..)                              */
/*********************************************************************/
#include <stdio.h>
main()
{ FILE *fp;
  char zeile[80], filename[25];
  char *str;                    /* str=  Zeigervariable auf ein Zeichen */

  printf("Geben Sie den Dateinamen ein -> ");
  scanf("%s", filename);          /* liest den Dateinamen ein */
  fp=fopen(filename,"r");      /* öffnet Textdatei zum Lesen */
```

```
  do
  {
   str = fgets(zeile,80,fp);     /* holt eine Zeile der Datei */
   if (str != NULL)
      printf("%s\n",zeile);    /* stellt eine Zeile auf Monitor dar */
  } while (str != NULL);         /* wiederhole bis Dateiende (->NULL) erreicht
*/

  fclose(fp);
}



/*********************************************************************/
/* Das Programm schreibt den über Tastatur eingegebenen Text in eine  */
/* Datei. Zieldatei muss in der Kommandozeile angegeben werden,        */
/*  z.B. auto.bat.  Aufruf von MS-DOS: EDIT auto.bat                  */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
 FILE *filepointer;
 char  ch;

 if (argc != 2)     /* argc gibt die Anzahl der übergebenen Parameter an */
 {
  printf("Es wurde keine Zieldatei angegeben !");
  exit(1);            /* gibt an die MS-DOS Umgebung den Wert 1 zurück */
 }

 if((filepointer=fopen(argv[1], "w"))==NULL)  /* übergebene Datei wird geöffnet*/
 {
  printf("\nSorry, ich kann die Datei %s nicht öffnen !", argv[1]);
  exit(1);
 }
 do
 {
  ch=getchar();             /* Liest ein zeichen von Standardgerät stdin */
  putc(ch,filepointer);     /* Schreibt zeichenweise nach Stream */
 } while (ch !='$');

 fclose(filepointer);       /* Schließen der Datei */
}



/*********************************************************************/
/* Programm zum Kopieren von Dateien                                  */
/* Quell- und Zieldatei müssen in der Kommandozeile angegeben werden  /*
/* Die Übertragung erfolgt Zeichen für Zeichen                        */
/* Aufruf von MS-DOS: KOPIERE config.sys config.bak                   */
/*********************************************************************/
#include <stdio.h>
#define LESEN  "rb"           /* Binäre Datei zum Lesen */
#define SCHREIBEN  "wb"           /* Binäre Datei zum Schreiben */

main(int argc, char *argv[])
{
 FILE *in, *out;
```

```
 char  ch;

 if (argc != 3)     /* argc gibt Anzahl der übergebenen Parameter an */
 {
  printf("Übergebene Parameter sind nicht vollständig !");
  exit(1);            /* gibt an die MS-DOS Umgebung den Wert 1 zurück */
 }

 if((in=fopen(argv[1], LESEN))==NULL)   /* übergebene Datei wird geöffnet */
 {
  printf("\nSorry, ich kann die Datei %s nicht öffnen !", argv[1]);
  exit(1);
 }

 if((out=fopen(argv[2], SCHREIBEN))==NULL)  /* Übergebene Datei wird geöffnet */
 {
  printf("\nSorry, ich kann die Datei %s nicht öffnen !", argv[2]);
  exit(1);
 }


 while (!feof(in))
 {
  putc(getc(in),out);  /*liest und schreibt zeichenweise von/nach nach Stream */
 };

 fclose(in);           /* Schließen der Dateien */
 fclose(out);
}




/*****************************************************************/
/* Disketten-Utility: DUMP <filename>                            */
/* Das Programm zeigt den Inhalt einer Datei in ASCII-Darstellung     */
/* und hexadezimal an.                                           */
/* Demonstration: wahlfreier Dateizugriff mit fseek(..)          */
/*****************************************************************/
#include <stdio.h>
#include <ctype.h>      /* enthält Funktion isprint(..) */
#include <stdlib.h>     /* enthält Funktion exit(..) */
#define SEKTORSIZE  128
#define LESEN  "rb"

void Anzeige(int liessekt);      /* Prototyp der Funktion */

char  buffer[SEKTORSIZE];       /* globale Variablendeklaration */


main(int argc, char *argv[])
{
 FILE *in *out;
 int sektor, liessekt;

 if (argc != 2)     /* argc gibt Anzahl der übergebenen Parameter an */
 {
  printf("übergebene Parameter sind nicht vollständig !");
  exit(1);            /* gibt an die MS-DOS Umgebung den Wert 1 zurück */
```

```
 }

 if((fp=fopen(argv[1], LESEN))==NULL)   /* übergebene Datei wird geöffnet */
 {
  printf("\nSorry, ich kann die Datei %s nicht öffnen !", argv[1]);
  exit(1);
 }

 do
 {
  printf("Gib Sektor-Nr. : ");
  scanf("%1d", &sektor);
  if (fseek(fp,sektor*SIZE,SEEK_SET))    /* SEEK_SET = Konstante für Dateianfang
*/
      printf(" SEEK Error !\n");          /* positioniere Dateizeiger */

  if ((liessekt= fread(buffer,1,SIZE,fp)) != SIZE)
      printf(" Dateiende erreicht !\n");  /* Lesezugriff */

  Anzeige(liessekt);
 } while(sektor>=0);
}    /* Ende Hauptfunktion */


void Anzeige(int sektnr)
/*************  Zeigt einen Sektor auf dem Bildschirm an **************/
{
 int i,j;

 for (i=0; i<=sektnr/16; i++)
 {
  for (j=0; j<16; j++)  printf("%3X", buffer[i*16+j];
  printf("   ");
  for (j=0; j<16; j++)
  {
   if (isprint(buffer[i*16+j]))  printf("%c", buffer[i*16+j]);
   else print(".");
  }
  printf("\n");
 }
}


/*********************************************************************/
/* Beispiel für die Lösung des Problems, eine Textdatei (in diesem    */
/* Fall TEST_PRN.TXT) auf den Drucker umzuleiten d.h. auszudrucken.   */
/*********************************************************************/
#include <stdio.h>
main()
{
 FILE *beispiel;      /* Erklärung von Pointern */
 FILE *printer;
 char ch;

 beispiel = fopen("TEST_PRN.TXT","r");    /* Öffne Eingabe-Datei */
 printer = fopen("PRN","w");              /* Öffne Druck-Datei */

 do
 {
  ch = getc(beispiel);    /* Hole ein Zeichen von Datei TEST_PRN.TXT */
```

```
  if (ch != EOF)
   {
    putchar(ch);          /* Gib Zeichen auf Monitor aus */
    putc(c,printer);      /* Gib Zeichen auf Drucker aus */
   }
 } while (ch != EOF;      /* Wiederhole bis EOF (end of file) */

 fclose(beispiel);        /* Schließen der Dateien */
 fclose(printer);
}
```

5.2 Übungsaufgaben zu den Dateioperationen

Übung 1: Dateien vergleichen

Schreiben Sie ein Programm, das zwei Textdateien Quelldatei und Zieldatei vergleicht.
Wenn die beiden Dateien gleichen Inhalt haben soll, soll die Meldung
- Quelldatei und Zieldatei sind identisch !
herausgegeben werden. Ist der Inhalt verschieden, soll bei dem ersten unterschiedlichen Zeichen ausgegeben werden:

- Quelldatei: gelesenes Zeichen <das Zeichen> in Zeile <Zeilennr.>
- Zieldatei: gelesenes Zeichen <das Zeichen> in Zeile <Zeilennr.>

Beachten Sie, dass auch die Zeilenstruktur übereinstimmen muss.

Übung 2: Dateien kopieren

Schreiben Sie ein Programm, das mit drei Dateien gleichzeitig arbeitet,
- eine Eingabedatei (Input)
- eine Ausgabedatei (Output)
- eine Protokolldatei zur Druckausgabe (Output)

Zuerst sollen die Namen der Ein- und Ausgabedateien über Tastatur eingelesen werden. Anschließend wird Zeichen für Zeichen die Eingabedatei gelesen und in die Ausgabedatei (Zieldatei) kopiert, gleichzeitig erfolgt Protokolldruck auf Drucker.

Übung 3: Dateien lesen und auflisten

Das Programm soll einen Dateinamen über Tastatur einlesen. Der Text wird zeilenweise auf dem Monitor ausgegeben unter Angabe der absoluten Zeilennummer, die jeder Zeile voran gestellt werden soll.

Übung 4: Schülerdatei anlegen

Schreiben Sie ein Standard C-Programm, welches eine Datei Schülerdaten erzeugt. Die Datei soll für jeden Schüler einer Klasse
- den Namen des Schülers,
- bis zu zehn Noten und
- die Durchschnittsnote enthalten.

Die Anzahl der Schüler und Noten pro Schüler soll über die Datei Input eingelesen werden. Für jeden Schüler existiert eine Eingabezeile in der Datei Input und zwar zuerst der Name mit maximal 20 Zeichen, dann ein Leerzeichen und schließlich die Noten (max. 10 Noten). Die Durchschnittsnote ist nicht angegeben und muss berechnet werden. Jede Eingabezeile wird durch ein Zeilenendezeichen (eoln = end-of-line) abgeschlossen.

6 Anhang

6.1 ASCII-Tabelle

6.1.1 Nicht druckbare Zeichen

Dez	Okt	Hex	Zeichen	Code	Bemerkung
0	000	0x00	Ctrl-@	NUL	Null prompt
1	001	0x01	Ctrl-A	SOH	Start of heading
2	002	0x02	Ctrl-B	STX	Start of text
3	003	0x03	Ctrl-C	ETX	End of Text
4	004	0x04	Ctrl-D	EOT	End of transmission
5	005	0x05	Ctrl-E	ENQ	Enquiry
6	006	0x06	Ctrl-F	ACK	Acknowledge
7	007	0x07	Ctrl-G	BEL	Bell
8	010	0x08	Ctrl-H	BS	Backspace
9	011	0x09	Ctrl-I	HT	Horizontal tab
10	012	0x0A	Ctrl-J	LF	Line feed
11	013	0x0B	Ctrl-K	VT	Vertical tab
12	014	0x0C	Ctrl-L	FF	Form feed
				NP	New page
13	015	0x0D	Ctrl-M	CR	Carriage return
14	016	0x0E	Ctrl-N	SO	Shift out
15	017	0x0F	Ctrl-O	SI	Shift in
16	020	0x10	Ctrl-P	DLE	Data link escape
17	021	0x11	Ctrl-Q	DC1	X-ON
18	022	0x12	Ctrl-R	DC2	
19	023	0x13	Ctrl-S	DC3	X-Off
20	024	0x14	Ctrl-T	DC4	
21	025	0x15	Ctrl-U	NAK	No achnowledge
22	026	0x16	Ctrl-V	SYN	Synchronous idle
23	027	0x17	Ctrl-W	ETB	End transmission blocks
24	030	0x18	Ctrl-X	CAN	Cancel
25	031	0x19	Ctrl-Y	EM	End of medium
26	032	0x1A	Ctrl-Z	SUB	Substitute
27	033	0x1B	Ctrl-[	ESC	Escape
28	034	0x1C	Ctrl-\	FS	File separator
29	035	0x1D	Ctrl-]	GS	Group separator
30	036	0x1E	Ctrl-^	RS	Record separator

31	027	0x1F	Ctrl-_	US	Unit separator
127	0177	0x7F		DEL	Delete or rubout

6.1.2 Druckbare Zeichen

Dez	Okt	Hex	Zeichen	Bemerkung
32	040	0x20		Leerzeichen
33	041	0x21	!	Ausrufungszeichen
34	042	0x22	"	Anführungszeichen
35	043	0x23	#	Doppelkreuz
36	044	0x24	$	Dollarzeichen
37	045	0x25	%	Prozentzeichen
38	046	0x26	&	Kaufmännisches UND
39	047	0x27	'	Apostroph
40	050	0x28	(	Runde Klammer auf
41	051	0x29	)	Runde Klammer zu
42	052	0x2A	*	Stern
43	053	0x2B	+	Pluszeichen
44	054	0x2C	,	Komma
45	055	0x2D	-	Minuszeichen
46	056	0x2E	.	Punkt
47	057	0x2F	/	Schrägstrich (Slash)
48	060	0x30	0	
49	061	0x31	1	
50	062	0x32	2	
51	063	0x33	3	
52	064	0x34	4	
53	065	0x35	5	
54	066	0x36	6	
55	067	0x37	7	
56	070	0x38	8	
57	071	0x39	9	
58	072	0x3A	:	Doppelpunkt
59	073	0x3B	;	Semikolon
60	074	0x3C	<	Kleiner-als-Zeichen
61	075	0x3D	=	Gleichheitszeichen
62	076	0x3E	>	Größer-als-Zeichen
63	077	0x3F	?	Fragezeichen
64	0100	0x40	@	Klammeraffe ("at")

65	0101	0x41	A	
66	0102	0x42	B	
67	0103	0x43	C	
68	0104	0x44	D	
69	0105	0x45	E	
70	0106	0x46	F	
71	0107	0x47	G	
72	0110	0x48	H	
73	0111	0x49	I	
74	0112	0x4A	J	
75	0113	0x4B	K	
76	0114	0x4C	L	
77	0115	0x4D	M	
78	0116	0x4E	N	
79	0117	0x4F	O	
80	0120	0x50	P	
81	0121	0x51	Q	
82	0122	0x52	R	
83	0123	0x53	S	
84	0124	0x54	T	
85	0125	0x55	U	
86	0126	0x56	V	
87	0127	0x57	W	
88	0130	0x58	X	
89	0131	0x59	Y	
90	0132	0x5A	Z	
91	0133	0x5B	[	Eckige Klammer auf
92	0134	0x5C	\	Umgekehrter Schrägstrich (Backslash)
93	0135	0x5D	]	Eckige Klammer zu
94	0136	0x5E	^	Caret (Hut)
95	0137	0x5F	_	Unterstrich
96	0140	0x60	`	"Back quote"
97	0141	0x61	a	
98	0142	0x62	b	
99	0143	0x63	c	
100	0144	0x64	d	
101	0145	0x65	e	
102	0146	0x66	f	
103	0147	0x67	g	

104	0150	0x68	h	
105	0151	0x69	i	
106	0152	0x6A	j	
107	0153	0x6B	k	
108	0154	0x6C	l	
109	0155	0x6D	m	
110	0156	0x6E	n	
111	0157	0x6F	o	
112	0160	0x70	p	
113	0161	0x71	q	
114	0162	0x72	r	
115	0163	0x73	s	
116	0164	0x74	t	
117	0165	0x75	u	
118	0166	0x76	v	
119	0167	0x77	w	
120	0170	0x78	x	
121	0171	0x79	y	
122	0172	0x7A	z	
123	0173	0x7B	{	
124	0174	0x7C	|	
125	0175	0x7D	}	
126	0176	0x7E	~	

6.2. Literaturangaben

"C"-Gesamtwerk
zu Quick C, Turbo C, MS C etc.
H. Herold, W. Unger
tewi-Verlag, 1994, DM 79,-

1x1 der C-Programmierung
Ulrich Cuber, Heino Wenzel
incl. Symantec C-Compiler auf CD-ROM
ECON-Taschenbuch Verlag, 1994, DM 36,90

Programmiersprache C - Eine strukturierte Einführung
Quick C, MS C/C++, Turbo-, Visual, Borland C/C++
Helmut Erlenkötter, Volker Reher
Rowohlt Taschenbuch Verlag, 1994, DM 16,90

"C" in der Praxis
K. Schröder
Oldenbourg Verlag, 1993, DM 48,-

Computerpraxis Schritt für Schritt: C Programmiergrundlagen
F. Wendler
Europa-Lehrmittel, 1994, DM 29,80

Programmieren in C
1755 Aufgaben und Lösungen, 42 vollständige Beipiele
kompatibel zum ANSI-Standard
B. S. Gottfried
Schaum Verlag, 1994, DM 48,-