Zeiner Karlheinz

Programmieren mit

C

für den Unterricht im Vorbereitungslehrgang
und in der Fachschule

Inhaltsverzeichnis

1 Einige kleine C-Programme

Kernighan und Ritchie, die Schöpfer der Programmiersprache C, beginnen ihr C-Buch mit einem Programm, das den Text "Hello world !" am Bildschirm ausgibt.

```
/* hello.c */

#include <stdio.h>

int main(void)
{
  printf("Hello world !\n");
  return 0;
}
```

Das nächste Programm löst die Aufgabe, zwei ganze Zahlen zu addieren:

```
/* addint.c      Addition von  zwei ganzen Zahlen */

#include <stdio.h>            /* Preprozessor-Anweisung */

int main(void)
{
  int a,b,c;      /* Vereinbarung der Variablen a,b und c */

  printf("Addition zweier Zahlen\n\n");  /* Text ausgeben */
  printf("a = ");
  scanf("%i", &a);        /* Zahl einlesen und a zuweisen */
  printf("b = ");
  scanf("%i", &b);
  c = a + b;              /* Berechnung von c             */
  printf("%i + %i = %i\n", a, b, c);          /* Ausgabe */

  return 0; /* 0 an die Betriebssystemumgebung uebergeben */

} /* end main */
```

Allgemeine Form eines C-Programmes:

```
/* Filename, Autor,
   kurze Beschreibung des Programmes als Kommentar */

#include <stdio.h>
#define Suchtext Ersatztext

int main(void)
{
  Vereinbarung der Variablen,
  dabei wird ihr Datentyp festgelegt

  Anweisungen

  return 0;
}
```

Datentypen:

für ganze Zahlen (integer values)	int
für Gleitkommazahlen (floating-point-values)	float, double
für beliebige Zeichen (characters)	char

Zu den Funktionen scanf und printf:

C ist eine "kleine" Sprache. Dies ist durch folgendes Konzept möglich. Viele in Programmen häufig benötigte Schritte, wie Ein-/Ausgabe, mathematische Funktionen, die Bearbeitung von Zeichenketten usw. sind nicht Teil der Sprache sondern werden durch die Verwendung von Funktionen (*functions*) gelöst. Funktionen sind die Bausteine eines C-Programmes. Teil eines C-Compilers oder besser C-Systems ist die sogenannte Standardbibliothek. Dies ist eine Sammlung von ca. 200 Funktionen für die Lösung der verschiedensten Aufgaben. Die Funktionen printf und scanf sind zwei dieser Funktionen.

Um eine Funktion zu verwenden, wird sie "aufgerufen". Den meisten Funktionen übergibt man Daten als sogenannte Argumente. Eine Funktion kann zudem ein direktes Resultat liefern. Der Aufruf einer Funktion hat die Form:

```
function_name ( arg1, arg2, ... )
   ↑                    ↑
```

Bezeichnung der Funktion Argumentliste, kann auch leer sein

Bei einer E/A Operation handelt es sich um den Transport von Daten. Es muß daher immer geklärt sein, was wird wohin/woher transportiert. Für **printf** ist die Quelle eine Liste von Argumenten, das Ziel der Bildschirm, für **scanf** ist die Quelle die Tastatur und das Ziel sind Speicherplätze für Variablen. Die beiden Funktionen kennen unseren Vereinbarungsteil nicht, d.h. sie wissen nicht, welchen Datentyp die transportierten Daten haben. Der Datentyp legt aber auch fest, wie ein Wert in Form von Bits im Speicher abgelegt wird. Datentransfer zwischen dem Benutzer und dem Computer bedeutet immer auch Codierung bzw. Decodierung. Der Benutzer verwendet z.B. das Dezimalsystem, um Zahlenwerte anzugeben, intern werden die Zahlen jedoch binär dargestellt. Die erforderliche Umwandlung wird von den E/A Funktionen durchgeführt. Dazu benötigen sie Information. Diese Information steckt im ersten Argument der Funktionen, einer Zeichenkette.

```
scanf("%i", &b);
printf("%i + %i = %i\n", a, b, c);        /* Ausgabe */
        ↑    ↑    ↑
        a    b    c
```

Die %i sind die Umwandlungszeichen für die Argumente a, b und c. Der restliche Text der Zeichenkette, die Leerzeichen, das Plus- und das Gleicheitszeichen werden zusätzlich ausgegeben.

Umwandlungszeichen für scanf- und printf- Funktionen:

Typ	printf	scanf
int	%i	%i
float	%f	%f
double	%f	%lf
char	%c	%c

Zwischen dem %-Zeichen und dem Umwandlungszeichen kann man weitere Angaben machen. Für Ausgaben bewährt sich die Angabe einer Feldweite z.B. in der Form

`%4i` für die Ausgabe werden 4 Druckpositionen verwendet

`%8.2f` für die Ausgabe werden insgesamt 8 Druckpositionen verwendet, die Zahl wird mit zwei Nachkommastellen ausgegeben.

Syntaxnotation

Um die Regel einer Programmiersprache zu formulieren, verwendet man verschiedene formale Darstellungen. Dabei wird definiert, wie man in der Sprache Wörter oder Sätze bildet bzw. produziert. Ein Beispiel für eine solche Darstellung von Regeln ist die Online-Hilfe des Betriebssystems, wo z.B. angegeben wird, wie man eine Kommandozeile zu formulieren hat. Der überwiegende Teil der C-Literatur verwendet zur Darstellung der Syntax die Backus-Naur-Notation. Diese Form der Syntaxbeschreibung kann mit einem einfachen Textsystem dargestellt werden, weil keine graphischen Symbole verwendet werden. Syntaxdiagramme (Eisenbahndiagramme) hingegen arbeiten mit graphischen Symbolen. Sie sind sehr leicht lesbar und werden hauptsächlich im Zusammenhang mit der Programmiersprache PASCAL verwendet.

In der Backus-Naur-Notation wird ein Syntaxbegriff *`kursiv`* geschrieben. Syntaxbegriffe werden durch Syntaxregeln, man kann auch von Produktionsregeln sprechen, definiert:

`digit` `::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

`letter` `::= a | b | c | ... | z | A | B | C | ... | Z`

`identifier` `::=` *`letter`* `| _ {` *`letter`* `|` *`digit`* `| _ }`$_{0+}$

Nicht kursiv geschriebene `Symbole` sind Teil der Zeichen- oder Wortmenge der Sprache und sind genau so hinzuschreiben. Die Symbole der Syntaxnotation selber sind natürlich nicht Teil der Sätze.

Symbole und ihre Bedeutung für die Schreibweise der Syntaxregeln:

Symbol	Bedeutung
`kursiv`	Syntaxkategorie
`::=`	"definiert als" oder "läßt sich überführen in"
|	Auswahl, kann auch als "oder" gelesen werden
$\{\}_{1}$	wähle eine der eingeschlossenen Einzelangaben
$\{\}_{0+}$	wiederhole das Eingeschlossene 0 - oder mehrmals
$\{\}_{1+}$	wiederhole das Eingeschlossene 1 - oder mehrmals
$\{\}_{opt}$	optionaler Teil, d.h. kann auch weggelassen werden

Statt den { }- Klammern werden oft spitze Klammern verwendet. Es gibt auch Schreibweisen, die ohne Klammern auskommen. Leider kommen das Auswahlzeichen (|), geschwungene und spitze Klammern auch in C vor.

Die Beschreibung der oben definierten Syntaxbegriffe in Worten lautet:

`digit` eine Ziffer aus der Menge 0 bis 9

`letter` ein Klein- oder Großbuchstabe

`identifier` muß mit einem Buchstaben oder dem Sonderzeichen _ (*`underscore`*) beginnen, kann dann mit Buchstaben, Ziffern oder dem Zeichen _ fortgesetzt werden

Auch die formal vollständige Syntaxbeschreibung einer Sprache erfaßt nicht alle Regeln. Zum Beispiel legt die Definition von *identifier* nicht fest, wie lange ein Bezeichner werden darf und wieviel Zeichen signifikant sind. Solche Details müssen zusätzlich verbal beschrieben werden. ANSI-C Compiler unterscheiden mindestens 31 signifikante Zeichen, die Bezeichner dürfen auch länger sein. C unterscheidet im Gegensatz zu vielen anderen Sprachen zwischen Groß- und Kleinbuchstaben, `xmax`, `Xmax` und `XMAX` sind daher drei verschiedene Bezeichner. Üblicher Stil ist, für Bezeichner fast ausschließlich Kleinbuchstaben zu verwenden. Das _-Zeichen (Unterstrich, *underscore*) wird verwendet, um zwei Wörter oder ein Wort und einen Zusatz zu einem Bezeichner zu verbinden. Ein Bezeichner darf kein Leerzeichen enthalten. Aussagekräftige Bezeichner sind wichtig für die Lesbarkeit und deshalb ein Kriterium für die Qualität eines Programms.

Beispiele für gültige Bezeichner:

```
i, liste, sortiert, fehler, FEHLER,
text_farbe, text_hoehe, text_breite
```

Falsch "konstruierte" Bezeichner:

```
linien-breite
```

ist nicht erlaubt, weil ein Bindestrich vorkommt

```
1tezahl
```

ist falsch, weil ein Bezeichner nicht mit einer Ziffer beginnen darf.

Wir werden im Rahmen des Buches immer wieder Syntaxregeln formal darstellen, das Buch legt allerdings keinen Wert auf eine "sprachwissenschaftlich" vollständige Darstellung der C-Syntax.

Schlüsselwörter

Schlüsselwörter sind reservierte Wörter, welche nicht neu definiert werden können. ANSI-C verwendet folgende Schlüsselwörter :

```
auto        do          goto        signed      unsigned
break       double      if          sizeof      void
case        else        int         static      volatile
char        enum        long        struct      while
const       extern      register    switch
continue    float       return      typedef
default     for         short       union
```

Kommentare

In C ist die Gefahr groß, schlecht lesbaren Code zu schreiben. Deshalb ist die begleitende Kommentierung des Programmtextes besonders wichtig. Jeder erfahrene Programmierer weiß, wie schwer man nach längerer Zeit auch selber geschriebene Programme lesen kann, wenn der Programmcode keine Kommentare enthält. Es lohnt sich, Programme von Beginn an sauber zu kommentieren.

Einige Möglichkeiten der Gestaltung von Kommentaren:

```
/* Kommentar */

/*****************************************************/
/* Trennlinie als Kommentar */

/*
```

```
 * Kommentar kann auch in dieser Form
 * geschrieben werden, um ihn vom Code
 * abzuheben
 */

/***********************\
* Kommentar in einer    *
* Kommentarbox          *
\***********************/
```

Was soll kommentiert werden ?

* Dateiname, Autor, Datum und Zweck eines Programms am Beginn des Programmcodes
* die verwendeten Variablen
* die Funktion von einzelnen Programmabschnitten
* besondere Anweisungen

Nicht kommentiert werden

Anweisungen, deren Zweck ohnehin klar ist, also z.B.:

```
printf("%d\n", i);     /* i ausgeben  */
y = x*y - 3/x;         /* y berechnen */
```

2 Arithmetische Ausdrücke

Formeln, welche Variablen und/oder Konstanten zu einem Term (Ausdruck) verbinden, nennt man arithmetische Ausdrücke. Beispiele für solche Ausdrücke sind:

```
g*t*t/2        (1 - a/b)*c      rk*(1 + ALPHA*dt +
BETA*dt*dt)
```

C kennt folgende arithmetische Operatoren:

```
+  -     für die Addition und Subtraktion
*  /     für die Multiplikation und Division
%        für den Rest einer Ganzzahldivision,
         nur für zwei int-Operanden definiert
```

Das Resultat eines arithmetischen Ausdrucks ist vom Typ **int** wenn alle Operanden vom Typ **int** sind, sonst ist das Resultat vom Typ **float**. Sind in einem Ausdruck **a / b** **a** und **b** vom Typ **int**, so wird eine Ganzzahldivision ausgeführt. Ein eventueller Nachkommateil wird abgeschnitten. (4/3 = 1, 27 / 8 = 3 usw.)

Der Datentyp von Variablen wird durch die Vereinbarung festgelegt, der Datentyp einer Konstanten ergibt sich aus der Schreibweise. Enthält eine Zahl einen Dezimalpunkt (3.1415, 4.0, 5.) oder einen Exponenten (1E-6, 1e12) so ist es eine Gleitkommakonstante, sonst eine ganze Zahl.

Für die Auswertung gemischter Ausdrücke gelten die Regeln: ***** , / und **%** werden vor + und - ausgeführt. Bei gleicher Wertigkeit erfolgt die Berechnung von links nach rechts.

Arithmetische Ausdrücke stehen oft auf der rechten Seite einer Zuweisung:

```
s = g*t*t/2;
rw = rk*(1 + ALPHA*dt + BETA*dt*dt);
```

allgemein:

```
Variable = Ausdruck;
```

Der Ausdruck auf der rechten Seite wird ausgerechnet, dazu müssen alle Variablen auf der rechten Seite bereits einen definierten Wert haben. Einen definierten Wert bekommt eine Variable durch eine Zuweisung oder über eine Tastatureingabe durch die Funktion scanf(...).

Eine Zuweisung, mit einem Strichpunkt abgeschlossen, gilt als Anweisung.

Dekrement und Inkrement-Operator

In Programmen muß man oft eine ganze Zahl um eins erhöhen oder erniedrigen. Dafür gibt es in C zwei Operatoren, die dies in sehr kurzer Schreibweise ermöglichen: ++ erhöht eine Variable um 1 , -- erniedrigt eine Variable um 1.

Beispiele: `i++, k--`

Die beiden Operatoren darf man nur auf **int**- und **char**-Werte anwenden!

3 Kontrollstrukturen

3.1 Algorithmen

Computerprogramme sind eine Lösung für ganz konkrete Aufgaben. Beispiele für solche Aufgaben sind: Zahlenwerte oder andere Daten speichern, wieder lesen, die Datenwerte sortieren, einen bestimmten Datenwert suchen, Gleichungssysteme lösen etc. Die Lösung einer Aufgabe ist eine Folge von Aktionen (Anweisungen). Im einfachsten Fall sind es eine bestimmte Anzahl von aufeinanderfolgenden Anweisungen, die man genau einmal ausführen muß. Meistens muß man für verschiedene Situationen jedoch unterschiedliche Anweisungen zur Auswahl haben, oder bestimmte Anweisungen öfters wiederholen, um zum Ziel zu kommen. Eine genaue Anleitung für die Lösung einer Aufgabe nennt man einen Algorithmus. Um eine konkrete Aufgabenstellung durch ein Computerprogramm zu lösen, muß man zunächst einen Algorithmus entwerfen. Es hat sich bewährt, Algorithmen zunächst in einer leicht verständlichen Art zu formulieren (Pseudocode) oder grafisch darzustellen (Struktogramme). Pseudocode arbeitet mit Text, mathematischen Symbolen und einigen einmal festgelegten Formulierungen wie **Falls ... dann ....** sonst oder **Wiederhole ..... solange ....** .

Die Theorie der Informatik sagt uns, daß man jedes prinzipielle durch einen Computer lösbare Problem durch Anwendung von drei elementaren Verfahren lösen kann, die man durch folgende Stichworte charakterisieren kann:

- Folge, Sequenz
- Auswahl, Alternative, Fallunterscheidung
- Wiederholung, Iteration

Die Folge oder Sequenz ist eine eindeutige Abfolge von Anweisungen, und braucht hier nicht weiter behandelt zu werden.

3.2 Bedingungen

Sowohl die Auswahl als auch die Wiederholung wird durch eine Bedingung gesteuert. Eine Bedingung prüft einen bestimmten Sachverhalt in der Regel so, daß das Resultat etweder WAHR (true) oder FALSCH (false) ist.

Häufige Bedingungen sind der Vergleich zweier Werte. Für den Vergleich zweier Werte a und b gibt es folgende Möglichkeiten:

```
a <  b        kleiner
a <= b        kleiner gleich
a >  b        größer
a >= b        größer gleich
a == b        gleich
a != b        ungleich
```

Diese Ausdrücke (Vergleichsausdrücke) sind entweder erfüllt (WAHR) oder nicht erfüllt (FALSCH). Während andere Programmiersprachen für Wahrheitswerte (Boolsche Algebra) einen eigenen Datentyp verwenden, verwendet C dafür ganze Zahlen. Das Resultat eines Vergleichsausdrucks ist in C 1 oder 0 (1 für WAHR , 0 für FALSCH).

Beachte den Unterschied zwischen == und = !

Ausdrücke können zusätzlich mit UND / ODER verknüpft werden, bzw. mit NICHT verneint werden. Die zugehörigen Operatoren sind

```
&& für UND        || für ODER        !  für NICHT.
```

Beispiel:

```
( a >= 10 ) && ( a < 100 )
```

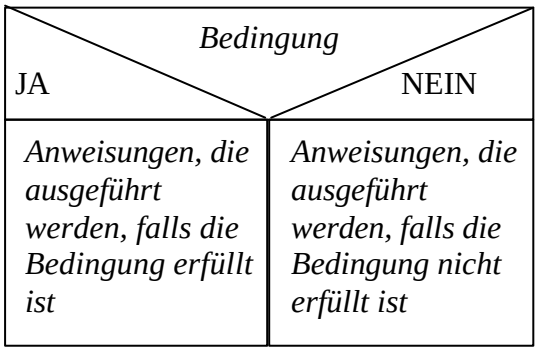
3.3 Auswahl

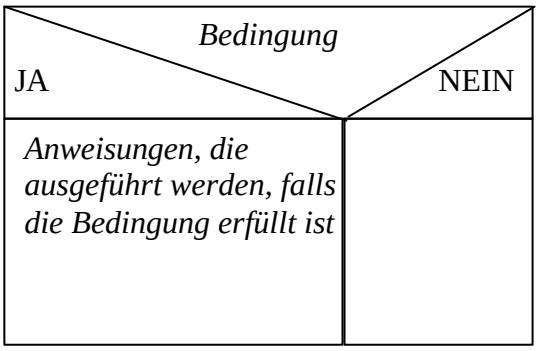
Bei der Auswahl wird durch eine Bedingung festgelegt, ob eine bestimmte Folge von Anweisungen oder eine andere Folge von Anweisungen anschließend ausgeführt wird. Ein Sonderfall ist, daß entschieden wird, ob bestimmte Anweisungen ausgeführt werden. Eine Pseudocode-Formulierung wäre:

```
Falls Bedingung
dann
  Anweisungen
sonst
  Anweisungen
```

```
Falls Bedingung
  Anweisungen
```

Im Struktogramm stellt man diese Situation folgendermaßen dar:





In C lautet die Syntax für solche Anweisungen:

```
if (Bedingung)
  Anweisungen
else
   Anweisungen
```

```
if (Bedingung)
  Anweisungen
```

Beachte:

Besteht der Schleifenrumpf oder ein Block einer Fallanweisung nur aus einer Anweisung, benötigt man keine Klammern **{** ... **}**, für zwei und mehr Anweisungen benötigt man die Klammern. Formal entsteht durch

```
{ Anweisung1;
  Anweisung2;
  ....
}
```

aus mehreren Anweisungen eine neue Anweisung, man nennt diese Konstruktion eine **Verbundanweisung** (*compound statement*).

Beispiele:

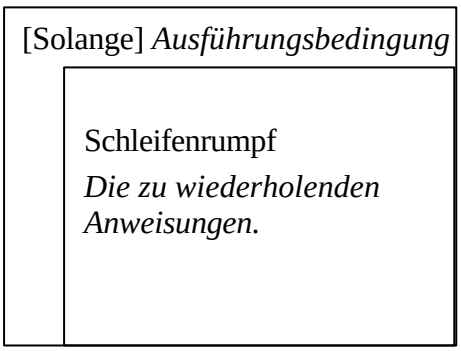
```
float a,b,h;
......
/* Tauscht die Werte a und b, falls a > b ist */
if (a > b) {
  h = b;
  b = a;
  a = h;
}

double x;
......
/* Wurzel aus einer beliebigen Zahl */
if (x >= 0.0)
  printf("Wurzel aus x = %8.2f\n", sqrt(x));
else
  printf("Wurzel aus x = %8.2f j\n", sqrt(-x));
```

3.4 Wiederholung

Oft muß man Anweisungen wiederholen, um eine Aufgabe zu lösen. Vermutlich ist die Tatsache, daß Computer ohne zu murren beliebig oft dasselbe tun, der Hautgrund für die Verwendung von Computern.

Eine spezielle Form einer Wiederholung ist die Endlosschleife. Eine einmal begonnene Endlosschleifen endet erst, wenn man das laufendes Programm gewaltsam durch das Betriebssystem unterbrechen läßt oder den Computer abschaltet. Die Anweisung, oder die Anweisungen, die wiederholt werden sollen, nennt man auch den Schleifenrumpf. In der Regel will man natürlich kontrollieren, wie oft der Schleifenrumpf wiederholt werden soll. Dabei muß man unterscheiden, an welcher Stelle geprüft wird und mit welcher Fragestellung geprüft wird, wie es weitergehen soll. Wir unterscheiden kopf- und fußgesteuerte Schleifen und Ausführungs- bzw. Abbruchbedingungen. Daneben gibt es auch noch die Möglichkeit, an einer oder mehreren beliebigen Stellen eine Schleife abzubrechen.




```
Solange Bedingung wiederhole
  Anweisung(en)
```

Dies ist eine kopfgesteuerte Schleife mit Ausführungsbedingung. Die Anweisungen werden ausgeführt, solange die Bedingung erfüllt ist. Die zugehörige Formulierung in C lautet:

```
while ( Bedingung )
  Anweisung
```

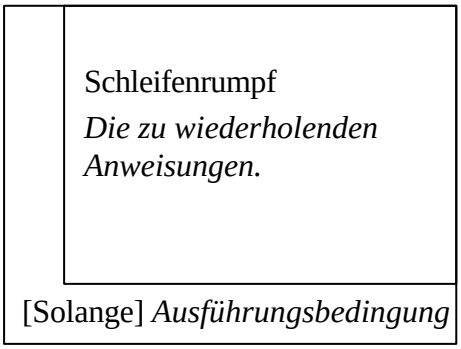
oder, falls man mehrere Anweisungen ausführen will:

```
while ( Bedingung ) {
  Anweisung1
  Anweisung2
  ....
} /* end while */
```

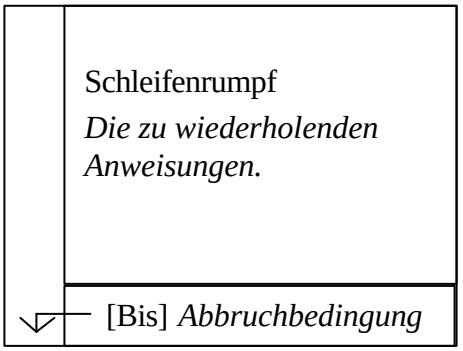
Diese Form der Wiederholung bietet auch die Möglichkeit, den Rumpf eventuell überhaupt nicht auszuführen, was manchmal nützlich sein kann.

Eine andere Form, am Fuß der Schleife gesteuert, wird folgendermaßen formuliert:

```
Wiederhole
  Anweisung(en)
solange Ausführungs-Bedingung
```




```
Wiederhole
  Anweisung(en)
bis Abbruchbedingung
```



Die Ausführungsbedingung muß erfüllt sein, damit ein weiteres Mal wiederholt wird. Die Abbruchbedingung muß erfüllt sein, damit die Wiederholung beendet, abgebrochen wird. In C kann nur die Form mit der Ausführungsbedingung codiert werden:

```
do
  Anweisung
while ( Bedingung );

  oder

do {
  Anweisung1
  Anweisung2
  ....
} while ( Bedingung );
```

Beispiel:

```
/* Zahlen einlesen, bis 0 eingegeben wird */
float z;
do {
  printf("z = ");
  scanf("%f", &z);
  .....
} while (z != 0.0);
```

3.5 Die for-Schleife:

Die for-Schleife ist eine while-Schleife, die alle zur Steuerung der Schleife relevanten Ausdrücke im Kopf der Schleife enthält.

```
for ( Initialisierung ; Ausführungsbedingung ; Kontrollvariable ändern )
  Anweisung
```

Dies entspricht einer while-Schleife der Form

```
Initialisierung
while ( Ausführungsbedingung ) {
  Anweisung
  Kontrollvariable ändern
}
```

Beispiel:

```
/* die Summe von n Zahlen berechnen */
summe = 0.0;
for (i = 0; i < n ; i++ ) {
  printf("%2i: ", i);
  scanf("%f", &x);
  summe = summe + x;
}
```

Man kann mit einer for-Schleife auch rückwärts zählen:

```
/* Countdown */
int n;
for (n = 10; n >= 0; n--)
  printf("%3i\n", n)
```

Beispiel (vollständigs Programm):

```
/* futab.c
   Funktionswerte berechnen,
   Vorzeichenwechsel (Nullstelle) entdecken */

#include <stdio.h>
#include <math.h>    /* fuer log */

int main (void)
{
  float x, xmin, xmax, dx;
  float y;
  char vorzeichen, altes_vorzeichen;

  xmin = 0.1;
  xmax = 2.0;
  dx = 0.125;
  printf("\n    x    \t    y\n");
  printf("----------------------------\n");

  /* Damit es auch beim ersten Durchgang ein
     altes_vorzeichen gibt:
     Vorzeichen fuer den ersten Wert merken */

  y = log(xmin);       /* natuerlicher logarithmus */
  if (y >= 0.0)
     altes_vorzeichen = '+';
   else
     altes_vorzeichen = '-';

  for (x = xmin; x <= xmax; x = x + dx) {
    y = log(x);
    if (y >= 0.0)
      vorzeichen = '+';
    else
      vorzeichen = '-';
    if (vorzeichen != altes_vorzeichen)
    printf("x-Achse passiert\n");
    printf("%8.2f\t%8.2f (%c)\n", x,y,vorzeichen);
    /* \t = Tabulator */
    altes_vorzeichen = vorzeichen;
  }

  getchar();
  return 0;
}
```

4 Indizierte Variablen (Vektoren) und Zeichenketten

4.1 Vektoren

Für das Arbeiten mit vielen Zahlen (oder Zeichen) verwendet man indizierte Variablen. In der C-Literatur nennt man solche Variablen Vektoren. Indizierte Variablen kennen wir auch aus der Mathematik, wo z.B. die Schreibweise

$x_i \quad i = 1,2,3,4,...n$

verwendet wird. Eine solche Schreibweise, mit tiefgestellten Indizes, ist in Programmiersprachen nicht möglich, der Index wird daher meist (C, C++, PASCAL) als Ausdruck zwischen eckige Klammern gesetzt:

```
x[i]  i = 1,2,3,4,...n
```

In C kann der Indexbereich nicht beliebig vereinbart werden. Man kann nur die Anzahl der Zahlen (Elemente) festlegen: Mit der Vereinbarung

```
float x[100];
```

wird ein Vektor mit 100 Elementen vereinbart, die erlaubten Indizes sind 0,1,2,3,4,...,98,99. Das erste von n Elementen hat immer den Index **0**, das letzte (n-te) Element hat den Index **n-1**.

Die allgemeine Form der Vereinbarung eines Vektors lautet:

Datentyp Bezeichner **[** *Anzahl der Elemente* **];**

Der Datentyp ist z.B. **float**, **double**, **int**, **char**. Als Bezeichner wählt man einen aussagekräftigen Namen, bei mathematisch orientierten Aufgaben auch einzelne Buchstaben wie x, y, a, etc.. Die Anzahl der Elemente ist eine Ganzzahlkonstante und muß daher schon beim Programmieren angegeben werden. Während der Programmausführung merkt man sich mit einer int-Variablen (z.B. n), wieviele der reservierten Elemente tatsächlich verwendet werden.

Die einzelnen Elemente eines Vektors stehen durch die Schreibweise

```
Vektor-Bezeichner [ int-Ausdruck ]
```

zur Verfügung. Vektor-Bezeichner ist der Name des Vektors, der int-Ausdruck ist eine ganze Zahl, eine int-Variable oder ein arithmetischer Ausdruck mit einem ganzzahligen Resultat. Der Wert des Ausdruckes muß im Bereich der gültigen Indizes liegen. Leider wird diese Bedingung in C-Programmen nicht überprüft und es kann zu unangenehmen Fehlern kommen.

4.2 Vektoren und die Zählschleife:

Alle Elemente eines Vektors erreicht man mit einer Schleife, in der eine Variable für den Index, die Werte 0 bis n-1 annimmt. Dies erfolgt z.B. in der Form

```
Für i = 0 bis n-1 führe aus
  Eingabeaufforderung für x[i]
  x[i] einlesen
```

In C verwendet man für eine solche Kontrollstruktur die **for**-Schleife.

```
for (i = 0; i < n ; i++ ) {
  printf("x[%2i] = ", i);
  scanf("%f", &x[i]);
}
```

Programmbeispiel Bubble-Sort:

```
/* bubble.c */

#include <stdio.h>
#define TRUE  1
#define FALSE 0

int main(void)
{
  int sortiert;       /* steuert die Schleife        */
  int i,n;            /* Index, Anzahl der Zahlen    */
  int x[51];          /* Indizierte Zahlenwerte      */
  int h;              /* für Tausch                  */

  /* Einlesen der Zahlen */

  printf("Bubble sortiert maximal 50 Zahlen !\n");
  printf("Wieviele Zahlen sollen sortiert werden ? ");
  scanf("%i", &n);
  for (i = 1; i <= n; i++) {
    printf("%2d : ",i);
    scanf("%i", &x[i]);
  }

  /* Sortieren */

  do {
    sortiert = TRUE;
    for (i = 1; i <= n-1; i++)
      if ( x[i+1] < x[i] ) {
         h = x[i];              /* Tauschen und  */
         x[i] = x[i+1];         /* Merker setzen */
         x[i+1] = h;
         sortiert = FALSE;
      }
  } while (sortiert == FALSE);

  /* sortierte Zahlen ausgeben */

  for (i = 1; i <= n; i++)
    printf("%2d : %6d\n", i, x[i]);

  return 0;

} /* end main */
```

4.3 Zeichenketten

Für Zeichenketten (*strings*) verwendet C keinen speziellen Datentyp. Zeichenketten sind Vektoren mit Elementen des Typs **char**. Die Anzahl der verfügbaren oder mit Werten belegten Elemente eines Vektors muß in C prinzipiell vom Programmierer beachtet werden. Im Prinzip gibt es dafür zwei Möglichkeiten. Wir merken uns den Index des letzten gültigen Elements, oder wir definieren einen besonderen Inhalt, der das Ende des Vektors anzeigt. Letztere Methode verwendet man in C für Zeichenketten. Das Zeichen mit dem Code 0, die Ersatzdarstellung ist '**\0**', dient als Abschlußzeichen für eine Zeichenkette. C selber kennt auch keine speziellen Operatoren für die Verarbeitung von Zeichenketten, stellt aber dafür in der Standardbibliothek eine Fülle von Funktionen zur Verfügung. Diese Funktionen beachten und verwalten in einer sehr konsistenten Weise das Abschlußzeichen. Fürs erste begnügen wir uns mit einem Programmbeispiel. Das Programm liest ein Wort und wandelt alle Kleinbuchstaben in Großbuchstaben um. Das neue Wort wird wieder ausgegeben.

```
/* string.c */

#include <stdio.h>
#include <ctype.h>
#define  MAXLENGTH 20

int main(void)
{
  char   wort[MAXLENGTH];
  int    i, laenge;

  printf("Programm string: Ein Wort eingeben: ");
  scanf("%s", wort);
  laenge = 0;
  while (wort[laenge] != '\0')
    laenge++;
  for (i = 0; i < laenge; i++)
    if (islower(wort[i])) wort[i] = toupper(wort[i]);
  printf("%s\n", wort);

  return 0;
}
```

Besprechung des Programms:

```
#include <ctype.h>
```

stellt uns die Funktionen **islower** und **toupper** zur Verfügung. Die Funktion **islower** prüft, ob ein Zeichen ein Kleinbuchstabe ist, **toupper** wandelt einen Kleinbuchstaben in einen Großbuchstaben um.

```
char   wort[MAXLENGTH];
```

vereinbart eine Zeichenkette für die Aufnahme von maximal MAXLENGTH Zeichen inklusive des Abschlußzeichens '**\0**'.

```
printf("Programm string: Ein Wort eingeben: ");
scanf("%s", wort);
```

ist ein Dialog zum Einlesen des Wortes. Die Formatspezifikation **%s** (Umwandlungszeichen **s** für **string**) ignoriert führende Leerzeichen und beendet die Eingabe mit dem Ende eines Wortes. Das Ab-

schlußzeichen '**\0**' wird automatisch angehängt. Mit dem Argument **wort** übergeben wir der **scanf**-Funktion die Anfangsadresse der Zeichenkette.

Die nächsten Programmzeilen

```
laenge = 0;
while (wort[laenge] != '\0')
  laenge++;
```

bestimmen die Länge der Zeichenkette (die Anzahl der regulären Zeichen), bzw. genauer gesagt die Position des Abschlußzeichens '\0'. Da das erste Zeichen den Index 0 hat, ist die Länge ident mit dem Index des Abschlußzeichens.

```
for (i = 0; i < laenge; i++)
  if (islower(wort[i])) wort[i] = toupper(wort[i]);
```

prüft für jedes Zeichen, ob es sich um einen Kleinbuchstaben handelt, und wandelt das Zeichen, falls nötig, in einen Großbuchstaben um.

```
printf("%s\n", wort);
```

gibt das Wort wieder aus, es enthält jetzt keine Kleinbuchstaben mehr.

5 Adressen und Zeiger

5.1 Adressen, Adreßoperator

Die kleinste Speichergröße, auf die zugegriffen werden kann, ist meist 1 Byte. Der Zugriff erfolgt über die Adresse dieses Bytes.

Die Daten unseres Sortierprogramms **bubble.c** werden im Speicher z.B. folgendermaßen verwaltet.

Adresse	Inhalte	Bezeichner
1216	27	h
1212	0	sortiert
1208	3	i
1204	10	n
1200	--	x_{50}
1012	7	x_3
1008	13	x_2
1004	27	x_1
1000	--	x_0

In C können die Adressen der Daten (Variablen) leicht ermittelt werden. Dazu wird der Adreßoperator **&** verwendet.

`&sortiert` ist (liefert) die Adresse, ab der die Variable **`sortiert`** abgespeichert wird. Genauso sind **`&i, &n`** die Adressen, ab denen die Variablen **`i`** und **`n`** abgespeichert sind. Für unser Beispiel würde **`&sortiert`** den Wert 1212, **`&i`** den Wert 1208, **`&n`** den Wert 1204 haben. Ebenso würde **`&x[0]`** den Wert 1000 haben.

Da die Adresse einer Variablen so leicht ermittelt werden kann, ist es naheliegend, einen speziellen Datentyp zu verwenden, um solche Adressen abzuspeichern. Diesen Datentyp bezeichnet man als Zeiger (*pointer*). Das Konzept der Zeiger ist für die Programmiersprache C von herausragender Bedeutung.

Durch die Vereinbarungen

```
int   *pi;          /* Zeiger auf int   */
float *pf;          /* Zeiger auf float */
```

werden die zwei Zeigervariablen **`pi`** und **`pf`** definiert. Auch die Schreibweise **`int * pi`** ist möglich. Zwischen dem Schlüsselwort für den Datentyp, dem Stern und dem Bezeichner für die Zeigervariable können beliebig viele Leerzeichen sein. Sind zusätzlich die Variablen

```
int   n;
float x;
```

definiert, so sind folgende Zuweisungen möglich :

```
n = 12;  x = 12.4;
pi = &n;
pf = &x;
```

Im Speicher ergibt sich z.B. folgende Situation:

Adressen	Inhalte	Variablenbezeichner
2012	2004	pi=&n
2008	2000	pf=&x
2004	12	n
2000	12.4	x

Natürlich kann man bei bekannter Adresse auf den Inhalt der folgenden Speicherzellen zugreifen. C stellt dafür den Inhaltsoperator (*deferencing operator)*, das Operatorzeichen dafür ist ein Stern (*), zur Verfügung. Der Inhaltsoperator ist wie der Operator & ein unärer Operator, der nachfolgende Operand muß vom Typ Zeiger sein. Ein Zeiger ist immer ein Zeiger auf einen bestimmten Datentyp. Dieser Datentyp wird in der Vereinbarung festgelegt. **pf** ist ein Zeiger auf einen float-Wert. **pi** ist ein Zeiger auf einen int-Wert. Wir werden im Kapitel 2 sehen, daß ein Speicherinhalt, je nach Interpretation, unterschiedliche Bedeutung haben kann. Die Anzahl der Bytes für den Inhalt und die Art der Interpretation des Inhaltes ergibt sich aus dem Datentyp, auf den der Zeiger zeigt.

Für unser Beispiel gilt:

```
*pi  ist 12
*pf  ist 12.4
```

Damit bekommt auch die bevorzugte Schreibweise für die Vereinbarung von Zeigervariablen

```
int *pi, i;        oder      float *pf, f;
```

eine einleuchtende Begründung. Im Ausführungsteil ist der Ausdruck ***pi** genauso vom Typ **int** wie der Bezeichner **i; *pf** und **f** sind vom Typ **float**. Beachten wir aber, daß mit obigen Vereinbarungen die Variablen **pi, i, pf** und **f** vereinbart werden. **pi** und **pf** sind Zeigervariablen, **i** und **f** sind Variablen für "normale" Zahlenwerte.

Übungsbeispiel: Die Variablen eines Programms stehen folgendermaßen im Speicher:

Typ	Variable	Adresse
int	k	2028
	a[2]	2024
	a[1]	2020
int[]	a[0]	2016
int *	pi	2012
float *	pf	2008
float	y	2004
float	x	2000

Welche Speicherinhalte ergeben sich durch folgende Zeilen C-Code ?

1.Durchgang	2.Durchgang
x=2.0;	y=2*x+5;
pf=&y;	pf=&x;
*pf=4.0;	*pf=20.7;
a[1]=1;	pi=a;
a[2]=2;	for(k=0;k<3;k++)a[k]=k*k;
pi=&k;	

5.2 Zeigerarithmetik

Eine wesentliche Bedeutung der Zeiger liegt in folgender Eigenschaft von Zeigern. Zeigerwerte kann man mit `++` inkrementieren (um 1 erhöhen) oder mit `--` dekrementieren (um 1 erniedrigen), man kann aber auch eine beliebige ganze Zahl addieren, bzw. subtrahieren. Dabei wird jedoch der Zeigerwert automatisch um Vielfache der Länge des Datentyps verändert, für den er definiert ist. Nehmen wir an die folgenden Zeiger

```
char *pc;
int *pi;
float *pf;
double *pd;
```

haben einen aktuellen Wert von 100, so gilt:

```
pc++ erhöht pc um 1
pi++ erhöht pc um 4 (oder 2, systemabhängig)
pf++ erhöht pc um 4
pd++ erhöht pc um 8
```

Diese Eigenschaft ist natürlich dann interessant, wenn mehrere gleiche Datentypen hintereinader im Speicher stehen. Dies gilt für Vektoren und Zeichenketten. Vektorindizierung ist lediglich eine Kurzschreibweise für Zeigerarithmetik. Ein Vektorbezeichner ohne Indizierung ist die Anfangsadresse des Vektors im Speicher:

```
v = &v[o]
```

Für die dereferenzierte Adresse eines Vektorelementes gilt:

```
v[i] = *(v + i * (Anzahl Bytes je Vektorelement) )
```

Der für einen Datentyp oder eine Variable verwendete Speicherplatz in Byte kann mit dem Operator sizeof ermittelt werden. Z.B. liefert `sizeof (int)` die für den Typ `int` verwendete Anzahl Bytes.

6 Modularisierung, Funktionen

6.1 Allgemeines, Begriffe

Ein wichtiges Prinzip beim Entwurf von Algorithmen ist die Modularisierung. Darunter versteht man die Zerlegung einer Aufgabe in mehrere in sich abgeschlossene Teilaufgaben. Diese Zerlegung ergibt sich auch durch das Konzept der schrittweisen Verfeinerung beim Entwurf von Algorithmen. Ein erster Grobentwurf für ein Programm zum Sortieren von Zahlen hat folgende Form:

```
Programm Bubble-Sort
{
  Zahlen einlesen;
  Zahlen sortieren;
  sortierte Zahlen ausgeben;
}
```

Damit ist automatisch eine Zerlegung in drei Teilaufgaben gegeben. Eine wichtige Forderung an eine Programmiersprache ist es, diese Vorgangsweise entsprechend zu unterstützen. Viele Programmiersprachen erlauben es, sogenannte Unterprogramme zu schreiben, welche vom Hauptprogramm aufgerufen werden, die sich aber auch gegenseitig aufrufen können. Ein Aspekt der Unterprogrammtechnik steht in unmittelbarem Zusammenhang zu einem Grobentwurf in Pseudocode. Es ist naheliegend, jede Pseudocodeanweisung durch den Aufruf eines Unterprogramms zu realisieren. Dadurch ergibt sich Code in einer quasi "virtuellen" Programmiersprache, da ein Unterprogrammaufruf syntaktisch eine Anweisung ist. Zum Beispiel wäre folgender Ausschnitt aus einem C-Programm syntaktisch richtig:

```
int main(void)           /* bubble sort */
{
  ...
  Zahlen_einlesen();
  Zahlen_sortieren();
  Zahlen_ausgeben();
  ...
}
```

C unterstützt diesen Stil des Programmierens. Bevor wir uns einen ersten Einblick in die Technik der Zerlegung in Teilaufgaben verschaffen, sollten wir einige Begriffe kennenlernen, da diese Begriffe teilweise synonym verwendet werden, bzw. auch sprachabhängige Bedeutung haben.

Unterprogramm, Prozedur, Routine, *subroutine, procedure, function*

sind solche Begriffe. Viele Sprachen kennen zwei Typen von Unterprogrammen, das "echte" Unterprogramm und das Funktionsunterprogramm.

Funktion:

In C gibt es nur einen Typ, man spricht in der engl. Literatur von ***function***. Die **C-*function*** ist

- kein "Unter"programm im Sinne einer hierarchischer Struktur (in C sind alle ***functions*** gleichwertig);
- keine Funktion im mathematischen Sinne oder im Sinne eines Funktionsunterprogramms, wenngleich sie auch für diesen Zweck verwendet werden kann.

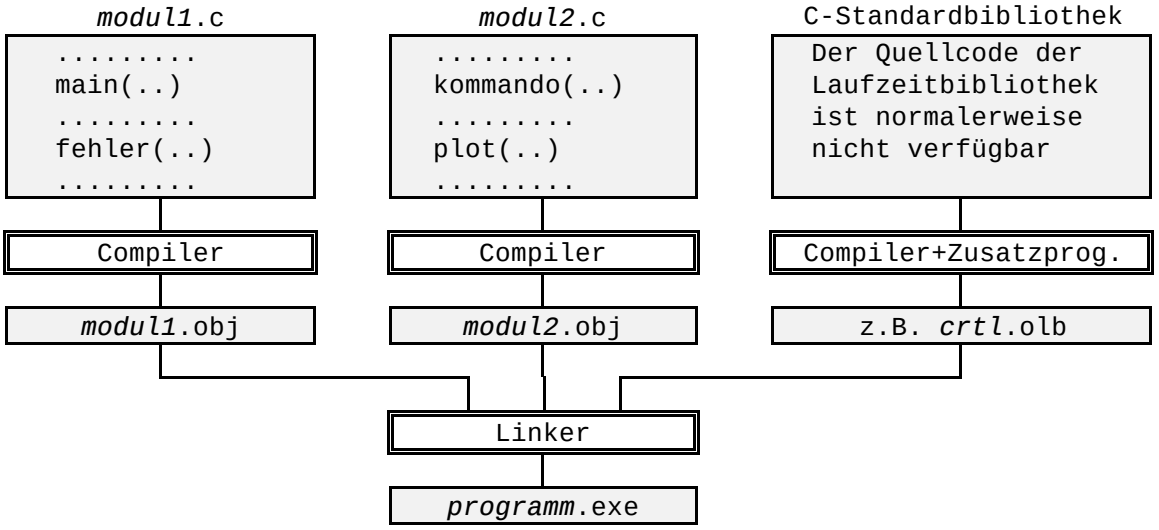
Wir verwenden daher den Begriff **Funktion** als Übersetzung für den Begriff ***function***, bzw. verwenden manchmal das Wort Routine als Synonym dafür.

Modul:

In der Terminologie des Programmentwurfs bezeichnet man ein Unterprogramm oder Teilprogramm auch als Modul. In der Terminologie der Programmiersprachen ist Modul eher einer sogenannten Über-

setzungseinheit (*translation-unit*) gleichzusetzen. Eine Übersetzungseinheit ist eine Datei mit Quellcode, die der Compiler als syntaktisch vollständige Einheit akzeptiert und übersetzt. Ein Modul kann in C (aber auch in anderen Sprachen) mehrere Funktionen enthalten.

Wir beschränken uns vorübergehend auf Programme, die zur Gänze in einer Datei (einer Übersetzungseinheit) verwaltet werden. Trotzdem soll das folgende Bild veranschaulichen, wie ein größeres Programm aus mehreren Modulen zusammengesetzt wird. Den Modul mit den Bibliotheksfunktionen verwenden wir ohnedies seit unserem ersten Programm.



Die Vorteile dieses Systems sind:

- Eine Aufgabe kann in überschaubare Teilaufgaben zerlegt werden, welche unabhängig voneinander (auch von verschiedenen Programmierern) gelöst werden können.
- Oft können bereits vorhandene Funktionen verwendet werden.
- Modularisierung bringt größere Sicherheit und leichtere Testbarkeit.
- Innerhalb einer Computerfamilie können bei Verwendung geeigneter Compiler auch Objekt-Files gemischt werden, welche aus Quellcode unterschiedlicher Programmiersprachen entstanden sind.

C verwendet das Prinzip der Modularisierung besonders ausgeprägt. Viele Möglichkeiten, welche man von einer höheren Programmiersprache erwartet - komfortable Ein/Ausgabe, mathematische Funktionen, Funktionen zur Bearbeitung von Zeichenketten etc. - werden in einer Bibliothek zur Verfügung gestellt, welche für diese Aufgaben eine Vielzahl von Funktionen enthält. Dies wiederum machte es möglich, den Sprachumfang selbst klein zu halten.

Betrachten wir einen C-Modul mit dem Code von drei Funktionen und der Verwendung von Funktionen aus der C-Bibliothek.

```
/* zweierpotenzen.c  */

#include <stdio.h>

void trennlinie(void)         /* Funktion trennlinie     */
{ int i;
  for(i = 1; i < 20; i++) putchar('-');
  putchar('\n');
}
```

```
int main(void)          /* Funktion main                  */
{
  int i, n;
  int zp;               /* Zweierpotenz                   */

  n = 12;               /* Zweierpotenzen bis 2^12        */
  trennlinie();         /* schirmloeschen und trennlinie  */
  printf(" i       2^i\n");
  trennlinie();         /* weitere Trennlinie drucken     */
  printf("%2d   %8d\n",0,1);
  zp = 1;
  for (i = 1; i <= n; i++) {
    zp = zp*2;
    printf("%2d   %8d\n",i,zp);
  }
  trennlinie();
  return 0;
}
```

Resultat und Besprechung des Programms:

```
------------------
 i       2^i
------------------
 0         1
 1         2
 2         4
 3         8
 4        16
 5        32
 6        64
 7       128
 8       256
 9       512
10      1024
11      2048
12      4096
------------------
```

Das Programm (der Modul) enthält drei Funktionen (*functions*), eine Funktion **trennlinie**, eine Funktion **schirmloeschen** und die Funktion **main**. Die Funktion **main** ist im Prinzip gleichwertig zu jeder anderen Funktion, trotzdem sind einige Besonderheiten zu beachten. **main** ist ein reservierter Bezeichner. **main** ist die Schnittstelle zum Betriebssystem. Das Betriebssystem aktiviert die Funktion **main**.

```
void schirmloeschen(void)
void trennlinie(void)
```

sind die Kopfzeilen (*header*) der zwei zusätzlichen Funktionen. Das Schlüsselwort **void** steht für "leer", "nichts", d.h. die Parameterliste (...) ist leer und die Funktion liefert keinen Wert zurück. Bei der Funktion **main** sind wir diesbezüglich noch etwas schlampig, was in diesem Falle üblicher C-Stil ist. Der Rumpf der Funktionen - das ist der Teil **{ ... }** - enthält wie der Rumpf der Funktion **main** Vereinbarungen und Anweisungen.

Die Funktion **schirmloeschen** arbeitet nur für Bildschirme korrekt, welche mit Steuerzeichen nach der ANSI-Norm arbeiten. Anmerkung: Der PC-Bildschirm verhält sich wie ein ANSI-Bildschirm, wenn der Gerätetreiber ANSI.SYS geladen ist.

```
char ESC = 27;
```

vereinbart die Variable ESC und initialisiert sie mit dem ASCII-Code des Steuerzeichens *ESCAPE*. Der ASCII-Code ist ein häufig verwendeter Code für die Codierung von Zeichen. Er enthält auch einige sogenannte Steuerzeichen (siehe Kapitel 2).

```
printf("%c%3s",ESC,"[2J");
```

gibt die Zeichenfolge aus, die den Bildschirm löscht. In der Funktion **trennlinie** verwenden wir eine **for**-Schleife, um 19 mal das Zeichen '-' auszugeben.

```
putchar(c)
```

ist der Aufruf einer weiteren Funktion der Standardbibliothek, die ein Zeichen auf den Bildschirm schreibt.

Der Aufruf der Funktionen **schirmloeschen** und **trennlinie** erfolgt in der Form

```
schirmloeschen();
trennlinie();
```

Die Anweisungen der Funktionen werden beim Aufruf ausgeführt, am Ende der Funktion er-folgt ein Rücksprung in die aufrufende Funktion **main**, wo mit der nächsten Anweisung fortgesetzt wird.

6.2 Datenaustausch zwischen Funktionen

Ein wichtiger Aspekt der Zerlegung eines Programms in Funktionen ist der Gültigkeitsbereich der Variablen innerhalb einer Funktion, man spricht auch von Sichtbarkeit. Für eine C-Funktion gilt ein strenges Lokalitätsprinzip, d.h. eine Funktion ist nach außen abgeschlossen, nichts ist außerhalb der Funktion sichtbar.

Jede Funktion verwaltet ihre Daten (Variablen) in einem eigenen Speicherbereich, die Variable **i** der Funktion `trennlinie` hat mit der Variablen `i` der Funktion `main` nichts zu tun. Die Variablen der Funktion `main` (`n` und `zp`) stehen anderen Funktionen nicht zur Verfügung, sie sind außerhalb der Funktion nicht sichtbar. Die Sichtbarkeit gilt nicht nur für die Variablen. Der gesamte Rumpf einer Funktion ist nur lokal gültig.

Der verwendete Mechanismus der Bereitstellung des Speichers beschränkt zudem die Lebensdauer der lokalen Variablen. Erst beim Aufruf einer Funktion wird für die Daten der Funktion Speicherbereich reserviert, dieser Speicher steht während der Abarbeitung der Funktion zur Verfügung, wird aber wieder freigegeben, wenn die Funktion beendet ist.

Innerhalb eines Programms muß natürlich ein Datenaustausch zwischen Funktionen möglich sein. Dafür gibt es drei Möglichkeiten:

1) Eine Funktion kann mit Argumenten aus der Parameterliste arbeiten.
2) Eine Funktion kann einen Wert (ein Resultat) an die aufrufende Funktion zurückliefern.
3) Global gültige Daten sind in mehreren Funktionen verfügbar.

Die allgemeine Form einer Funktion ist

```
function_definition ::=
type_specifier function_name(parameter_declaration_list)
{
  declarations
  statements
}
```

Die Parameterliste (*parameter_declaration_list*) ermöglicht den Transport von Daten der aufrufenden Funktion in die aufgerufene Funktion. Mit *type_specifier* wird der Datentyp des Resultates festgelegt. Für das Resultat ist die Transportrichtung umgekehrt.

6.2.1 Parameterliste

Für die Parameterliste gilt vereinfacht die Syntax

```
parameter_declaration_list ::=
      parameter_declaration {, parameter_declaration }0+
```

Die Parameterliste ist eine Liste von Vereinbarungen. Jeder Parameter muß einzeln definiert werden, die einzelnen Vereinbarungen sind durch Beistriche getrennt. Für jeden Parameter muß ein Datentyp und ein Bezeichner festgelegt sein. Für einfache Datentypen, bzw. Zeiger auf einfache Datentypen gilt für einen formalen Parameter die Syntax:

```
parameter_declaration ::= type { * }opt identifier
```

Beispiel für eine Parameterliste:

```
(int n, float x, float y, float *px)
```

Der Datentyp muß für jeden Parameter angegeben werden, eine Liste von Bezeichnern zu einem Datentyp, wie es im Vereinbarungsteil einer Funktion möglich ist, ist nicht erlaubt:

```
(int n, float x,y, *px)   /* falsch ! */
```

Innerhalb der Funktion können die in der Parameterliste vereinbarten Parameter wie Variablen verwendet werden, sie sind auch tatsächlich Teil der lokalen Variablen einer Funktion. Eine leere Parameterliste wird durch das Schlüsselwort **void** definiert.

Wir verbessern jetzt die Funktion **trennlinie** dadurch, daß die Länge der Trennlinie beim Aufruf der Funktion als Argument übergeben werden kann:

```
void trennlinie (int laenge)
{
  int i;

  for (i = 0; i < laenge; i++) putchar('-');
  putchar('\n');
}
```

Korrekte Aufrufe dieser Funktion sind

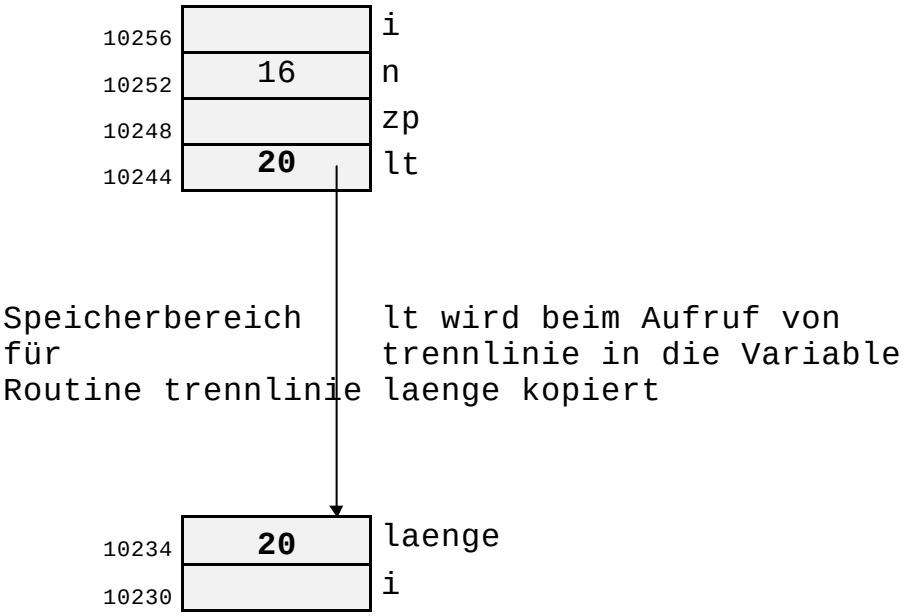
```
trennlinie (40);
```

oder wenn in der aufrufenden Funktion also z.B. in `main` zusätzlich eine Variable `lt` vom Typ `int` bekannt ist und einen definierten Wert hat:

```
trennlinie (lt);
```

Beim Aufruf einer Funktion werden den vereinbarten Parametern (man spricht auch von formalen Parametern) aktuelle Parameter zugewiesen. Für jeden formalen Parameter muß ein passender aktueller Parameter (ein Argument) übergeben werden. Die Argumente werden in den Speicherbereich der Funktion kopiert. Im Speicher gibt es einen Bereich für die Funktion `main` und einen Bereich für die Funktion `trennlinie`. Wir wollen die Situation graphisch darstellen:

Speicherbereich für Routine main:



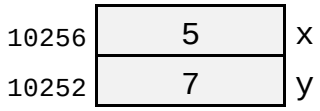
Eine eventuelle Änderung der Variablen **laenge** in der Funktion **trennlinie** hat auf das Original **lt** keinen Einfluß.

6.2.2 Daten der aufrufenden Funktion ändern

Oft will man in einer Funktion Daten der aufrufenden Funktion verändern oder auch erstmals mit Werten belegen. Ein Beispiel für eine solche Funktion ist die Bibliotheksfunktion `scanf(....)`, welche Daten von der Tastatur einliest und diese Daten im Speicher ablegt. Wie kann eine Funktion Daten der aufrufenden Funktion verändern? Ein typisches Beispiel dafür ist eine Funktion, welche zwei Zahlen vertauschen soll. Eine solche Funktion hätten wir in unserem Programm `bubble.c` verwenden können.

```
int main(void)
{
  int x, y;
  x = 5;  y = 7;
  printf("x = %3d  y = %3d\n", x, y);
  tausche ( ? )      /* x und y soll vertauscht werden */
  printf("x = %3d  y = %3d\n", x, y);
  return 0;
}
```

Anordnung der Variablen x und y im Speicher:



Die folgenden Überlegungen sind für ein Verständnis der Sprache C besonders wichtig. In der Funktion **tausche** sind die Variablen **x** und **y** nicht bekannt. Um **x** und **y** zu ändern (in unserem Falle zu tauschen) müssen wir wissen, wo **x** und **y** im Speicher stehen, d.h. wir müssen die zugehörigen Adressen kennen. Diese können wir über den Adreßoperator ermitteln. **&x**, **&y** liefert die Adressen **10256** und **10252**. Diese Adressen übergeben wir der Funktion **tausch** als Argumente in der Parameterliste. Um diese Adressen beim Aufruf als aktuelle Parameter übernehmen zu können, müssen wir zwei Zeiger als Parameter zur Verfügung stellen. Wir wissen bereits, wie man Zeigervariablen vereinbart und wie man auf über Zeiger adressierte Inhalte zugreifen kann. Der Code für die Funktion **tausche** lautet:

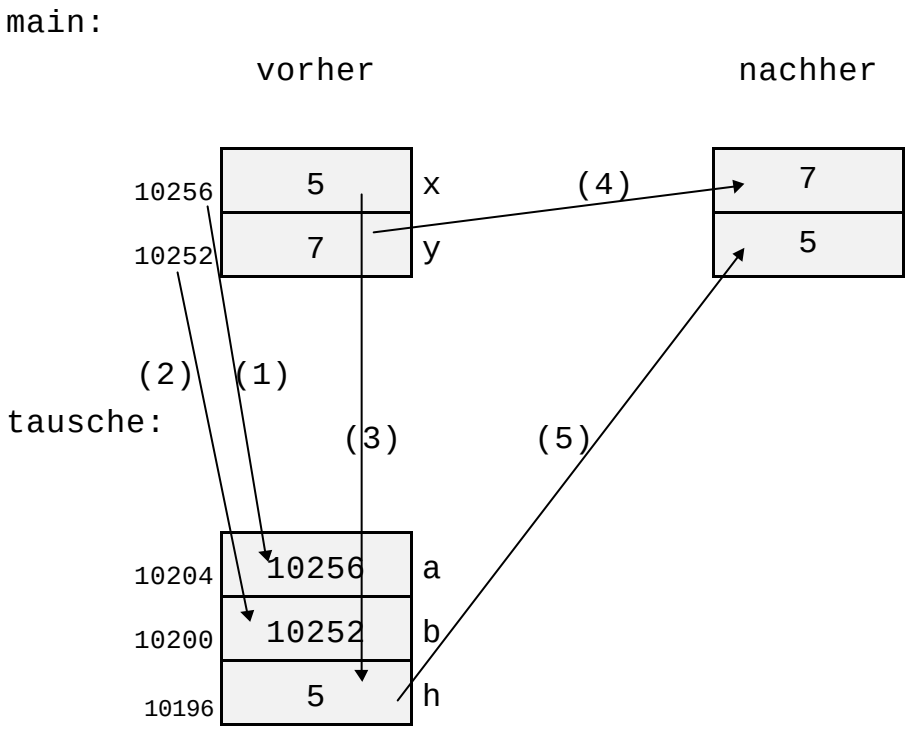
```
void tausche(int *a, int *b)
{ int h;
  h  = *a;
  *a = *b;
  *b = h;
}
```

Erinnern wir uns daran, daß mit **int *a**, **int *b** zwei Zeiger **a**, **b** definiert werden.

Der Aufruf der Funktion erfolgt in der Form:

```
tausche (&x, &y);
```

Wir studieren nun die einzelnen Vorgänge im Speicher anhand des folgenden Bildes. Die Ziffern in Klammer geben an, in welcher Reihenfolge die einzelnen Schritte ablaufen.



Die Funktion **main** hat die Variablen **x** und **y** ab den Adressen **10256** bzw. **10252** gespeichert. Beim Aufruf der Funktion **tausche** werden die Adressen dieser Variablen als aktuelle Parameter übergeben:

```
tausche (&x, &y);   (1) (2)
```

Die Adreßwerte werden in den Speicher für die Variablen **a** und **b** der Funktion **tausche** kopiert, d.h. **a** hat den Wert **10256**, **b** hat den Wert **10252**. Dies ist die Situation, bevor mit der eigentlichen Ausführung der Anweisungen der Funktion **tausche** begonnen wird. Der Zugriff auf die Variablen **x** und **y** der Funktion **main** erfolgt durch Dereferenzierung der Zeiger auf diese Variablen. Als Dereferenzierung bezeichnet man den Zugriff auf den Inhalt von Speicherzellen über Adressen. Dazu müssen wir auf die Zeiger **a** und **b** den Inhaltsoperator anwenden. Vor dem Tausch ist *a = x = 5 und *b = y = 7.

```
h  = *a;    /* (3) kopiert x in den Speicher für h */
*a = *b;    /* (4) kopiert y in den Speicher für x */
*b = h;     /* (5) kopiert h in den Speicher für y */
```

Die für den Tausch erforderliche Hilfsvariable **h** ist eine lokale Variable vom selben Typ wie die Variablen **x** und **y**. Für die Funktion **main** hat diese Hilfsvariable oder andere Details der Funktion **tausche** keine Bedeutung.

Der Aufruf einer Funktion kann auch als Delegierung einer Arbeit an eine untergeordnete Dienststelle verstanden werden. Der Umfang der Arbeit ist genau festgelegt, die benötigten Daten werden als aktuelle Parameter zur Verfügung gestellt. Wie die Arbeit erledigt wird, kümmert den Auftraggeber (die aufrufende Funktion) nicht.

Vektoren als Parameter von Funktionen:

Ein Vektor als formaler Parameter einer Funktion wird in der Form

```
type identifier[]
```

angegeben. Als aktuelles Argument wird der Bezeichner des Vektors (ohne & Operator !), also die Adresse des ersten Vektorelementes, übergeben. Eine Information über die Länge des Vektors ist damit nicht verbunden. Die Funktion kennt jedoch den Datentyp der Vektorelemente und damit die Größe eines Elementes. Diese Informationen (Adresse des ersten Elementes und Größe eines Elementes) reichen aus, um die einzelnen Elemente des Vektors über den Index auszuwählen. Da vermutlich der größte zulässige Index in der Funktion benötigt wird, muß er als zusätzliches Argument übergeben werden.

Als Beispiel implementieren wir unseren Sortieralgorithmus als Funktion.

```
/* Sortieren von n Elementen des Vektors x */

#define TRUE  1
#define FALSE 0

void Zahlen_sortieren (int x[], int n)
{
  int i, h;
  int sortiert;

  do {
    sortiert = TRUE;
    for (i=0; i < n-1; i++)
      if ( x[i+1] < x[i] ) {
        h = x[i]; x[i] = x[i+1]; x[i+1) = h;
```

```
        sortiert = FALSE;
      }
  } while (sortiert == FALSE);

}
```

Die Indizierung der Zahlen haben wir auf die in C übliche Art umgestellt. Die n zu sortierenden Zahlen sind mit 0, 1, 2, ... n-1 numeriert.

Wir haben damit eine Funktion, die beliebig viele in einem Vektor fortlaufend angeordnete **int**-Werte sortiert. Jede andere Funktion kann diese Funktion verwenden. Um die Funktion als unabhängige Bibliotheksfunktion zur Verfügung zu stellen, sind noch einige kleinere Maßnahmen notwendig, die wir im Kapitel 5 kennenlernen. Ein Programm, das die drei Aufgaben "Zahlen einlesen, Zahlen sortieren und Zahlen ausgeben" auf drei Funktionen aufteilt, hat folgende Form:

```
int main(void)
{
  int v[50];   /* Liste von Zahlen */
  int n;       /* Anzahl der Zahlen */

  Zahlen_einlesen ( v, &n);
  Zahlen_ausgeben ( v, n);
  Zahlen_sortieren (v, n);
  Zahlen_ausgeben ( v, n);
  return 0;
}
```

6.2.3 Resultat einer Funktion, return-Anweisung

Die C-Funktion kann auch ein Resultat an die aufrufende Funktion liefern. Man nützt diese Möglichkeit im wesentlichen auf zwei Arten. Die erste Anwendung entspricht dem mathematischen Funktionsbegriff. Aus einer oder mehreren unabhängigen Variablen wird eine von diesen abhängige Variable berechnet.

$y = f1(x) = 2x^2 + x - 4$
$z = f2(x,y) = x^2 + y^2$

Aus der Sicht der Funktion sind **x** und **y** Eingabewerte, das Resultat **f(x,..)** wird berechnet und zurückgeliefert. Die Realisierung in C ist einfach:

```
float f1 (float x)
{ float y;
  y = 2*x*x + x - 4;
  return y;
}

float f2 (float x, float y)
{
  return (x*x + y*y);
}
```

Der Aufruf dieser Funktionen erfolgt z.B. in der Form

```
float x, dy, y, r1, r2;
.....
x = 2.0;
```

```
y = f1(x);
dy = f1(x + 0.1*x) - f1(x);
r1 = x * f1(4.0);
r2 = f2(x, 3.);
```

Die Kopfzeile der Funktion beginnt mit einem Typbezeichner für den Datentyp des Resultates. Mit der return-Anweisung

return_statement ::= return $expression_{opt}$

wird das Resultat und die Kontrolle wieder an die aufrufende Funktion zurückgegeben.

Typische Beispiele für Funktionen dieser Form sind die in der C-Bibliothek zur Verfügung stehenden mathematischen Funktionen **sin(x), cos(x), ... sqrt(x), pow(x,n),** .... Will man sie verwenden, so ist die Datei **math.h** einzubinden:

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

`math.h` enthält die Funktionsprototypen für diese Funktionen (siehe nächstes Kapitel).

Zwei weitere Beispiele für Funktionen: Die Funktion **sign** ermittelt das Vorzeichen einer Zahl, die Funktion `quadrant` bestimmt den Quadranten, in dem ein Punkt in einem kartesischen Koordinatensystem liegt:

```
/* Bestimmt das Vorzeichen einer float-Zahl,
   bzw. liefert 0 wenn die Zahl exakt 0.0 ist  */

int sign (float x)
{
  if (x < 0)
     return -1;
  else if (x == 0.0)
     return 0;
  return 1;
}


int quadrant (float x, float y)
{
   if ( x == 0.0 && y == 0.0 ) return 0;
   if ( x >  0.0 && y >= 0.0 ) return 1;
   if ( x <= 0.0 && y >  0.0 ) return 2;
   if ( x <  0.0 && y <= 0.0 ) return 3;
   if ( x >= 0.0 && y <  0.0 ) return 4;
}
```

Beide Beispiele zeigen, wie in einer Funktion auch mehrere `return`-Anweisungen verwendet werden können. Sobald das Resultat eindeutig feststeht, kann die Funktion mit `return` beendet werden.

Häufig verwendet man das Resultat einer Funktion, um den Erfolg einer Funktion an die aufrufende Funktion zu melden. Die Funktion liefert dann typisch einen **int**-Wert. Den einzelnen Zahlenwerten kann man Erfolgs- oder Fehlermeldungen zuordnen. Ein Beispiel dafür sind die BibliotheksFunktionen **scanf** und **fscanf**. Sie liefern als Resultat die Anzahl der erfolgreich gelesenen Argumente. Wir werden dies in den Programmbeispielen zum Arbeiten mit Dateien nützen. Die bisherige Verwendung von **scanf** zeigt uns, daß das Resultat einer Funktion nicht verwendet werden muß. Das nächste Beispiel ist eine Funktion, welche eine komplexe Zahl **z** in die Inverse **1/z** umwandelt. Es gilt:

$z = a + j \cdot b \quad 1/z = (a - j \cdot b) / (a^2 + b^2) \quad j = !-1$

1/z kann nicht berechnet werden, wenn der Betrag der komplexen Zahl Null ist. Die Funktion meldet diesen Fall mittels des Wertes **0** als Resultat, im Normalfall liefert sie als Resultat **1**.

```
/* cinv.c berechnet die Inverse einer komplexen Zahl */

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

int cinv (float *a, float *b)
{
  float betragquadrat;

  betragquadrat = *a * *a + *b * *b;
  if (betragquadrat > 0.0) {
     *a = + *a / betragquadrat;
     *b = - *b / betragquadrat;
     return 1; }
  else
     return 0;
}
```

Um den mit zwei **float**-Variablen festgehaltenen Real- und Imaginärteil der Zahl zu ändern, benötigt die Funktion **cinv** zwei Zeiger auf diese Variablen. Um den Erfolg der Berechnung als Resultat der Funktion zur Verfügung zu stellen, definieren wir den Typ der Funktion mit **int**. Der Typ einer Funktion ist der Datentyp des Resultates. Den Nenner der Berechnungsformel für die Zahl **1/z** müssen wir berechnen, bevor der Real- oder Imaginärteil der ursprünglichen Zahl **z** verändert wurde. Das Zusammentreffen des Inhaltsoperators mit dem Operator für die Multiplikation wirkt etwas eigenartig, die Schreibweise ist aber richtig.

Die im Quellcode gleich anschließend folgende Funktion **main** verwendet die Funktion **cinv** und dokumentiert die Ergebnisse:

```
int main(void)
{
  float a,b;     /* Realteil und Imaginaerteil */
  int status;    /* Fehlercode                 */

  do {
    printf("\nkomplexe Zahl z = a + jb  : a b ? ");
    scanf("%f %f", &a, &b);
    status = cinv (&a, &b);
    if (status) {
      if (b >= 0) /* positiver Imaginaerteil */
        printf("\n1/z = %8.2g + j%-8.2g\n", a, b);
      else
        printf("\n1/z = %8.2g - j%-8.2g\n", a, fabs(b));
      }
    else
      printf("Fehler: |z| = 0 !\n\n");
  } while (status > 0);
  return 0;
}
```

In der Funktion **main** verwenden wir für positive oder negative Imaginärteile unterschied-liche Formatierungsangaben:

```
if (b >= 0) /* positiver Imaginaerteil */
  printf("\n1/z = %8.2g + j%-8.2g\n", a, b);
else
  printf("\n1/z = %8.2g - j%-8.2g\n", a, fabs(b));
```

Das Umwandlungszeichen **g** in der Formatspezifikation **%8.2g** erzeugt eine Ausgabe, die je nach Größe der Zahl automatisch zwischen Fixkommadarstellung (#####.##) und Exponentialdarstellung (#.##e±##) wechselt. Das Minus-Zeichen in der Formatspezifikation **%-8.2g** bewirkt, daß die Zahl immer linksbündig ausgebenen wird.

6.3 Funktionsprototypen

Beim Aufruf einer Funktion prüft der Compiler, ob die Anzahl und der Typ der übergebenen aktuellen Parameter mit der Anzahl und dem Typ der formalen Parameter der Funktion übereinstimmen. Wir haben in unseren Beispielen den Code der Funktionen immer so angeordnet, daß der Aufruf der Funktion nach dem Code der Funktion erfolgte. Den vollständigen Code einer Funktion bezeichnet man als Definition einer Funktion. Dadurch ist gewährleistet, daß der Compiler die formale Parameterliste der Funktion beim Aufruf der Funktion kennt. Wir müssen uns dazu vorstellen, daß der Compiler den Code nur einmal "liest". Er kann nicht an einer Stelle unterbrechen, um im Rest der Datei nach fehlenden Informationen zu suchen.

Der Code einer Funktion kann z.B. in einer anderen Datei stehen oder überhaupt nicht als Quellcode verfügbar sein. Dies gilt z.B. für alle BibliotheksFunktionen eines C-Systems. Falls sich zwei Funktionen gegenseitig aufrufen, kann nur eine der Funktionen definiert sein, bevor sie von der anderen Funktion aufgerufen wird. Das heißt, wir brauchen eine weitere Möglichkeit, dem Compiler die Namen und die Parameterliste der verwendeten Funktionen bekanntzumachen. Wir haben bereits festgestellt, daß für die Verwendung einer Funktion der Code der Funktion völlig uninteressant ist. Für die korrekte Verwendung einer Funktion müssen wir wissen:

- Welche Aufgabe führt die Funktion aus?
- Welchen Namen hat die Funktion?
- Wie ist die Parameterliste aufgebaut? Welche Bedeutung haben die einzelnen Parameter?
- Liefert die Funktion ein Resultat? Welchen Datentyp hat das Resultat?

Alle diese Informationen sind in der Kopfzeile einer Funktion enthalten. Man bezeichnet die Angabe einer solchen Kopfzeile in der Form

type function_name(*parameter_type_list*);

als Funktionsprototyp. Für *parameter_type_list* kann dieselbe Syntax wie für *parameter_declaration_list* verwendet werden. Die von uns in praktisch allen Programmen mit `#include` eingebundenen Dateien vom Typ `*.h` enthalten die Prototypen für die Funktionen der C-Bibliothek. Die Einbindung dieser Dateien ist also deshalb wichtig, weil der Compiler damit den korrekten Aufruf der BibliotheksFunktionen überprüfen kann. Fehlende Prototypen können zu völlig falschen Resultaten führen, dies gilt insbesondere für die Verwendung der mathematischen Bibliotheksfunktionen, deren Prototypen in der Datei `math.h` stehen. Trotzdem ist die Verwendung von Prototypen nicht zwingend erforderlich. Dies merken wir z.B. daran, daß der Compiler nur mit Warnungen reagiert, wenn wir z.B. die Funktionen `printf` und `scanf` verwenden und auf das `#include <stdio.h>` vergessen.

7 Bit-Operatoren und Ausdrücke

Bit-Operatoren und Ausdrücke sind eine der maschinennahen Seiten der Programmiersprache C. Die Wirkung der Operatoren ist in manchen Details maschinenabhängig. Als Argumente sind nur **int**-Typen zugelassen. Ist uneingeschränkte Portabilität wichtig, sollte man sich auf Argumente vom Typ **unsigned int** beschränken.

Logische Bit-Operatoren:

unäres bitweises Komplement	~
bitweises UND	&
bitweises exklusives ODER	^
bitweises ODER	|

Schiebeoperatoren:

nach links	<<
nach rechts	>>

Bitweises Komplement (Einerkomplement)

Der Operator **~** ist der 'Einerkomplementoperator", d.h. er bildet das Einerkomplement eines Bitmusters. Für die Bildung des Einerkomplements werden alle Einser durch Null ersetzt und umgekehrt, man könnte also auch von einer bitweisen Negation sprechen. Für

```
int  a = 0x00f0f724;
```

ist die binäre interne 32-Bit Darstellung

```
 a = 00000000 11110000 11110111 00100100
```

das Einerkomplement ~a ist dann

```
~a = 11111111 00001111 00001000 11011011
```

Für eine Maschine, welche die negativen ganzen Zahlen als Zweierkomplement der entsprechenden positiven Zahl darstellt, wird durch das Zweierkomplement aus **x** die interne Darstellung von **-x**. Das nächste Programm (**bit1.c**) verwendet diese Operatoren um die Zusammenhänge zwischen Einer- und Zweierkomplement zu demonstrieren. Das Programm erzeugt folgende Ausgabe :

```
Spalte 1 : Zahl
       2 : interne Darstellung der Zahl
       3 : Einerkomplement der Zahl
       4 : Zweierkomplement der Zahl
       5 : Zweierkomplement als Dezimalzahl

    0  00000000    11111111    00000000     0
    1  00000001    11111110    11111111    -1
    2  00000010    11111101    11111110    -2
    3  00000011    11111100    11111101    -3
    4  00000100    11111011    11111100    -4
    5  00000101    11111010    11111011    -5
         ...
   15  00001111    11110000    11110001   -15
   16  00010000    11101111    11110000   -16
```

```
/* bit1.c
   Einerkomplement und Zweierkomplement von Zahlen
-------------------------------------------------*/
#include <stdio.h>
#define N  16

void bitprint (int z, int nb, char endchar);
typedef signed char BYTE;

int main (void)
{
   BYTE z,   /* Zahl   */
        ekz,  /* Einerkomplement  der Zahl */
        zkz;  /* Zweierkomplement der Zahl */

    printf ("%s%s%s%s%s",
     "Spalte 1 : Zahl\n",
     "       2 : interne Darstellung der Zahl\n",
     "       3 : Einerkomplement der Zahl\n",
     "       4 : Zweierkomplement der Zahl\n",
     "       5 : Zweierkomplement als Dezimalzahl\n\n");

   for (z = 0; z <= N; z++) {
     printf ("  %3d  ",z);
     bitprint (z,1,' ');
     printf ("   ");
     ekz = ~z;
     bitprint (ekz,1,' ');
     zkz = ekz + 1;
     printf("   ");
     bitprint (zkz,1,' ');
     printf ("  %3d  \n",zkz);
   }
  return 0;
}
```

Bemerkungen zum Programm und Ausgaben des Programms:

```
void bitprint (int z, int nb, char endchar);
```

ist der Prototyp für eine Funktion, welche die niederwertigsten `nb` Bytes eines int-Wertes als Bitmuster ausgibt. Das Argument `endchar` kann zur Steuerung des Zeilenvorschubs verwendet werden.

Bitweises UND

Für alle Bits gilt: Das i-te Bit von **a & b** ist

1 , falls das i-te Bit von **a** UND das i-te Bit von **b** eins ist

0 , in allen anderen Fällen

Bitweises ODER

Für alle Bits gilt: Das i-te Bit von **a | b** ist

1 , falls das i-te Bit von **a** ODER das i-te Bit von **b** eins ist (oder beide)

0 , wenn keines der Bits 0 ist

Bitweises exklusives ODER

Für alle Bits gilt: Das i-te Bit von **a ^ b** ist

1 , falls das i-te Bit von **a** ODER das i-te Bit von **b** eins ist (aber nicht beide)

0 , wenn beide Bits gleich sind

Syntax der Bit-Ausdrücke:

```
~ expression
expression ^ expression
expression | expression
expression & expression
```

Zusammenfassende Wahrheitstabelle:

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Schiebeoperationen

Ein Ausdruck der Form

```
a << n
```

verschiebt das Bitmuster von **a** um **n** Positionen nach links. Die **n** Bits links gehen dabei verloren, von rechts werden **n** Nullen nachgeschoben.

Ein Ausdruck der Form

```
a >> n
```

verschiebt das Bitmuster von **a** um **n** Positionen nach rechts. Die **n** niederwertigsten Bits gehen dabei verloren, von links werden für **unsigned**-Ausdrücke **n** Nullen nachgeschoben. Ist der Operand **a** vorzeichenbehaftet, so ist das Resultat implementierungsabhängig.

Die allgemeine Form einer Schiebeoperation ist

```
expr1 << expr2        expr1 >> expr2
```

expr1 und **expr2** müssen Ganzzahlausdrücke sein und werden der Integererweiterung unterworfen. Das Resultat ist vom Typ des ausgedehnten Ausdruckes **expr1**. Ist **expr2** negativ oder zu groß, so ist das Resultat undefiniert.

Die Operatoren können auch mit dem Zuweisungsoperator kombiniert werden. So ist z.B. der Ausdruck

```
x >>= 1;
```

ist äquivalent zu

```
x = x >> 1;
```

Beispiele (gelten für 32-Bit **int**-Werte):

```
unsigned char c = 'Z';
  c    :                                 01011010
c << 1 :  00000000 00000000 00000000 10110100
c << 8 :  00000000 00000000 01011010 00000000
```

Die folgende Initialisierung der Variablen **a** und **b** ist ebenfalls nur für 32-Bit **int**-Werte korrekt.

```
unsigned a = 1 << 31;   /* erzeugt 1000000 ... 00000000 */
int      b = 1 << 31;   /* erzeugt 1000000 ... 00000000 */

    a,b     1000000 00000000 00000000 00000000
  b >> 3    1111000 00000000 00000000 00000000
  a >> 3    0001000 00000000 00000000 00000000
```

Für **unsigned**-Werte (a) werden von links Nullen nachgeschoben; für **signed**-Werte werden auf den meisten Systemen Einsen nachgeschoben (damit bleibt die Zahl negativ).