PEARSON Education

ADDISON-WESLEY

Wolfgang Kinzel

Programmierkurs für Naturwissenschaftler und Ingenieure

Schnelleinstieg in Linux, C, Java und Mathematica/Maple

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 C

Zur Lösung rechenzeitintensiver numerischer Probleme werden vor allem zwei Programmiersprachen eingesetzt: Fortran und C. Beide Sprachen sind elementar; sie greifen direkt auf numerische Werte im Speicher zu und deshalb können sie einfache Operationen sehr schnell ausführen. Umfangreiche Bibliotheken bieten außerdem weit entwickelte Routinen zur Lösung spezieller numerischer Probleme an.

Wir haben uns in diesem Schnelleinstieg für die Sprache C entschieden. Das Betriebssystem Unix/Linux ist in dieser Sprache geschrieben, ebenso wie die höhere Programmiersprache Mathematica. Die objektorientierten Sprachen Java und C++, die sich immer weiter verbreiten, bauen auf C auf. Und auch gegenüber Fortran schließlich hat C einige Vorteile: Programme lassen sich kompakter und modularer formulieren, C erlaubt rekursive Funktionen und maschinennahes Programmieren. Wir wollen aber auch nicht die Nachteile verschweigen: C erlaubt viele Fehler. Ohne Warnung kann man beispielsweise den Speicher des eigenen Programms überschreiben. Außerdem lassen sich C-Programme völlig unverständlich formulieren. Jedes Jahr gibt es einen internationalen Wettbewerb um das unverständlichste C-Programm, und wer das Programm des Gewinners versteht, kann wirklich gut in C programmieren.

3.1 Programmstruktur und Notation

Ein C-Programm besteht aus einer Folge von Funktionen. Es muss wenigstens die Funktion `main` enthalten, die beim Start des Programmes zuerst aufgerufen wird. Ein einfaches C-Programm könnte folgendermaßen aussehen:

```
/* Berechnung des Sinus einer Zahl x */
#include<math.h>
main()
   {  double x=1.3;
      printf("Der Sinus von %f hat den Wert %f\n",x,sin(x));
   }
```

Dieser Quelltext wird im ASCII-Format in eine Datei, beispielsweise mit dem Namen `sinus.c`, geschrieben und abgespeichert. Dann müssen diese Zeilen in maschinennahe Anweisungen übersetzt werden, das macht der Compiler `gcc`:

```
gcc –o sinus sinus.c –lm
```

Nun existiert eine Datei `sinus`, die eine für uns nur unverständliche Folge von Bits enthält, ein sogenanntes *binary file*. Diese Datei hat das Zugriffsrecht `x`, sie ist daher ausführbar. Gibt man jetzt `sinus` ein, so wird auf dem Bildschirm der folgende Text geschrieben:

```
Der Sinus von 1.300000 hat den Wert 0.963558
```

Wir wollen nun das obige Programm erläutern. Die erste Zeile ist ein Kommentar; alles was zwischen die Symbole `/*` und `*/` geschrieben wird, ignoriert der Compiler. Mit dem Befehl `#include` können Sie weitere Dateien hinzufügen. Hier wird ein sogenanntes *Header-File* , die Datei `math.h`, dazu geladen. Wenn der Dateiname wie hier zwischen eckigen Klammern steht, wird die Datei in beonderen Verzeichnissen gesucht. Wenn Sie Ihre eigene Datei hinzufügen wollen, sollten Sie deren Namen in Anführungsstriche setzen:

```
#include "deklarationen.h"
```

Danach wird die Funktion `main` definiert. Sie enthält hier wegen der leeren Klammer `()` keine Argumente. Die geschweiften Klammern `{...}` fassen alle Anweisungen der Funktion `main` zusammen. Zunächst wird eine Variable mit dem Namen `x` deklariert. Sie enthält eine reelle Zahl mit 8 Byte = 64 Bit Speicherplatz, und es wird ihr sofort der Wert 1.3 zugewiesen. Dann wird die Funktion `printf` (drucke formatiert) aufgerufen, der drei Argumente übergeben werden: eine Zeichenkette, die zwischen die Anführungsstriche `"..."` eingeschlossen wird, die Variable `x` und die Funktion `sin` mit dem Argument `x`. Die Zeichenkette enthält Text und Steuerzeichen. Das erste Zeichen `%f` wandelt den Wert von `x` in Textzeichen um, das zweite Zeichen `%f` wandelt den Wert von $\sin(x)$ in Text um. Das Steuerzeichen `\n` ist das Symbol für die Return-Taste (neue Zeile). Weitere Steuerzeichen werden im Abschnitt 3.6 erklärt.

Noch einige Bemerkungen zur Notation: Jede einzelne Anweisung muss mit einem Semikolon abschließen. Am Anfang werden Sie häufig die Fehlermeldung *parse error* erhalten; dann haben Sie vermutlich entweder ein Semikolon vergessen oder Ihre Klammern nicht richtig geschlossen. C unterscheidet zwischen Groß- und Kleinschreibung. Die Variable mit dem Namen `xY` belegt daher einen anderen Speicherplatz als die Variablen `Xy` und `xy`. Bei der Namenswahl haben Sie große Freiheiten, Namen dürfen allerdings nicht mit einer Ziffer beginnen und Punkte sind für Strukturen reserviert (siehe Abschnitt 3.4).

Leerzeichen und Zeilentrennung haben im Quelltext keine Bedeutung, außer in einer Zeichenkette. Das vorherige Beispiel können Sie auch als eine einzige Zeile schreiben, jedoch wird es dadurch unverständlich. Man sollte den Text immer gut strukturieren: Zusammengehörende Klammern sollten untereinander stehen und die dazwischenliegenden Anweisungen sollten entsprechend eingerückt werden.

Abschließend noch ein Hinweis auf die verschiedenen Arten von Klammern in C: Geschweifte Klammern `{...}` fassen mehrere Anweisungen zu einem Block zusammen. Runde Klammern `(...)` kennzeichnen Argumente von Funktionen oder legen die Reihenfolge algebraischer Auswertungen fest. Eckige Klammern `[...]` dagegen bezeichnen Indizes für Felder. Alle diese Notationen werden wir noch in vielen Beispielen kennen lernen.

Wenn Sie nun den Quelltext in eine Datei geschrieben haben, müssen Sie ihn in Maschinenbefehle übersetzen (*kompilieren*). Wir werden hier den GNU-Compiler `gcc` verwenden, der meistens im Linux-Paket und auf UNIX-Systemen vorhanden ist. Mit der

Option `-o` können Sie Ihrer ausführbaren Datei einen Namen geben, ohne diese Option erhält sie den Namen `a.out`. Danach folgt der Name der Quelldatei, der immer mit `.c` enden muss. Am Ende der Befehlszeile wird mit der Option `-lm` die Mathematik-Bibliothek dazu geladen. Ohne diese Option kennt das Programm nicht die Funktion `sin(x)`. Nun wird das Programm einfach durch Eingeben seines Namens gestartet. Falls es lange läuft, sollten Sie es im Hintergrund laufen lassen:

```
sinus &
```

Falls mehrere Kollegen den Rechner benutzen, sollten Sie nett sein und Ihrem Programm mit der Anweisung `nice` eine geringere Priorität geben:

```
nice sinus &
```

Falls Sie die Ergebnisse nicht auf den Bildschirm, sondern in eine Datei mit dem Namen `daten` schreiben wollen, können Sie die Ausgabe mit dem Befehl

```
sinus > daten &
```

umleiten. Ein Compiler übersetzt Ihren Quelltext in Maschinenanweisungen. Schon vor dem Kompilieren haben Sie jedoch in C mit dem *Präprozessor* die Möglichkeit, Symbole im Quellcode zu überschreiben. Dazu gibt es die `define`-Anweisung.

```
#define N 1000
```

am Anfang des Quelltextes ersetzt beispielsweise den Text `N` durch den Text `1000`, ohne dabei auf die Bedeutung der Symbole zu achten. Diese Konstruktion hat den Vorteil, schon zu Beginn des Programmes einige feste Parameter wie Feldgrößen usw. festlegen zu können, ohne spezielle Variablen dafür zu deklarieren. Es gibt zahlreiche weitere Präprozessordirektiven, von denen wir jedoch nur die vorige verwenden werden.

3.1.1 Variablen

Jedes Programm holt Daten aus einem Speicher, führt mit ihnen Rechnungen durch und speichert die Ergebnisse wieder ab. Sie als Programmierer brauchen sich nicht darum zu kümmern, wo Ihre Daten im Computer stehen, sondern Sie müssen jedem Speicherplatz nur einen Namen geben und seinen Typ festlegen, also die Größe des Speicherplatzes und die Art, wie die einzelnen Bits interpretiert werden sollen. Es kann nützlich sein zu wissen, dass der gesamte Speicher, einschließlich der eigenen Binärdatei des Programmes und sämtlicher Zahlen, Bilder und Texte nichts anderes als eine lineare Kette von Bits (Einsen und Nullen) ist.

Sie müssen dem Compiler mitteilen, wie viele Speicherplätze Sie von welchem Typ benötigen. Das geschieht mit den *Deklarationen* am Anfang eines Blocks. Wenn Sie beispielsweise zwei ganze Zahlen mit den Namen `rechts` und `links` abspeichern wollen, so müssen Sie am Anfang des Anweisungsblocks der Funktion `main` die Anweisung

```
int rechts, links;
```

setzen. Sämtliche Variablen müssen in C deklariert werden, sonst erkennt der Compiler diese Ausdrücke nicht und liefert Fehlermeldungen. Das gilt auch für Felder und Funktionen (siehe die Abschnitte 3.4 und 3.5):

```
int matrix[10][20], funktion(int x);
```

Je nach Inhalt und Länge des Speicherplatzes gibt es in C verschiedene Standardtypen, einige davon sind in der Tabelle 3.1 aufgelistet.

Tabelle 3.1: Typen von Variablen und deren Wertebereiche

Name	Länge in Bytes	Wertebereich
char	1	$-128 \ldots 127$ oder ASCII-Zeichen
long	4	-2147483648 bis 2147483647
short	2	-32768 bis 32767
int	2 oder 4	entsprechend
float	4	$-10^{38} \ldots 10^{38}$
double	8	$-10^{308} \ldots 10^{308}$

Variablen vom Typ `char` enthalten 8 Bit, die entweder als ganze Zahl zwischen -128 und 127 oder als Textzeichen interpretiert werden können. Als Textformat verwendet UNIX/Linux den sogenannten ASCII-Code; jedes Zeichen hat damit eine Nummer zwischen 0 und 255. Dabei werden die Zeichen der Tastatur in einfache Anführungsstriche wie bei `'a'` eingeschlossen. Die Return-Taste hat beispielsweise das ASCII-Zeichen 10, das als Zahl 10 oder als Zeichen `'\n'` eingegeben werden kann. Das Zeichen `'0'` kann als Zahl 48 interpretiert werden, während die Zahl 0 dem Zeichen `'\0'` entspricht.

In C geht die Analogie zwischen Zahlen und Zeichen so weit, dass Sie mit den Zeichen rechnen können. Zum Beispiel können Sie Großbuchstaben in die entsprechenden Kleinbuchstaben umwandeln, wenn Sie die Zahl 32 (eine Eins beim sechsten Bit) hinzu addieren, `'A'+ 32` ergibt den Wert `'a'` (oder die Zahl 97). Der Ausdruck `'B' < 'b'` ist eine wahre Aussage (mit dem Wert *eins*). Auf der beiliegenden CD gibt es die C-Programme `ascii` und `tasten`, die sämtliche ASCII-Zeichen ausdrucken bzw. ein eingetipptes Zeichen als Zeichen, ganze Zahl, Oktalzahl, Hexadezimalzahl und als Bitkette darstellen.

Ganze Zahlen der Typen `int, short` und `long` können auch vorzeichenlos definiert werden wie bei `unsigned int`. Damit wird das erste Bit nicht als Vorzeichen interpretiert, und der positive Zahlenbereich verdoppelt sich. Außerdem können sie ganze Zahlen in verschiedenen Zahlensystemen schreiben. Neben der üblichen Dezimalschreibweise ist auch die Angabe als oktale (Basis 8) oder hexadezimale (Basis 16) Konstante möglich. Eine oktale Zahl wird durch eine Null und eine hexadezimale durch `0x` am Anfang dargestellt. Die Zahl 74 kann beispielsweise als 0112 ($= 1 \cdot 8^2 + 1 \cdot 8^1 + 2 \cdot 8^0$) oder als 0x4a ($= 4 \cdot 16^1 + 10 \cdot 16^0$) geschrieben werden. An der Oktalzahl kann man auch schnell die

Bitdarstellung (jedenfalls bei positiven Zahlen) ablesen, wobei jede Ziffer jeweils durch drei Bit geschrieben wird.

```
0   1   1   2
    ↓   ↓   ↓
  001 001 010
```

Reelle Zahlen (Fließkommazahlen) vom Typ `float` oder `double` werden durch einen Dezimalpunkt dargestellt. Es gibt weder ein Dezimalkomma noch ein Komma zur Trennung von Tausenderstellen. Allerdings dürfen relle Zahlen ein `E` oder ein `e` für eine folgende Zehnerpotenz enthalten. Gültige Zahlen sind beispielsweise -1.5; 10.26E4 für 102 600. und .5e-3 für 0.0005.

In C gibt es keinen besonderen Datentyp für die logischen Werte Wahr und Falsch. Diese Werte werden einfach durch ganze Zahlen dargestellt, und zwar erhält Falsch den Wert Null und Wahr einen Wert ungleich Null.

```
int i;
i = 5<2;
printf("%d",i);
```

Dieses Beispiel zeigt, dass der Wert des logischen Ausdrucks $(5 < 2)$ einer ganzen Zahl zugewiesen werden kann, das Ergebnis ist der Wert Null.

3.1.2 Zeiger

Wir haben schon erwähnt, dass man in C sehr maschinennah programmieren kann. So gibt es Variablen für die *Adresse* eines Speicherplatzes, sie werden *Zeiger* genannt. Solche Zeigervariablen werden mit einem Stern (`*`) deklariert, wobei der Typ der zugehörigen Variablen, die diese Adresse hat, angegeben werden muss.

```
int * z;
```

Der Speicherplatz mit dem Namen `z` enthält hier eine Adresse einer Variablen vom Typ `int`. Ihm kann ein Wert, nämlich eine Adresse einer ganzzahligen Variablen, zugewiesen werden. Die Adresse einer gewöhnlichen Variablen erhält man mit dem Operator `&`.

```
int a, * z;
z = &a;
```

Hier ist `a` eine Variable vom Typ `int`. Ihre Adresse `&a` wird dem Zeiger `z` zugewiesen. Der Wert auf dem Speicherplatz mit der Adresse `z` kann mit `*z` abgefragt werden. Wird also der Variablen `a` der Wert 5 zugewiesen:

```
a = 5;
```

so liefert der Ausdruck `*z` den Wert 5. Die Tabelle 3.2 soll dies verdeutlichen.

Tabelle 3.2: Zeiger und Variablen

`int a;`	Deklaration einer Variablen vom Typ `int`
`int * z;`	Deklaration eines Zeigers, der eine Adresse einer Variablen vom Typ `int` enthält
`z=&a;`	Der Zeiger `z` enthält nun die Adresse von `a`
`&a`	Adresse der Variablen `a`
`a`	Wert der Variablen `a`
`z`	Adresse der Variablen `a`
`*z`	Wert der Variablen `a`

Auch bei unseren späteren einfachen Beispielen können wir auf Zeiger nicht ganz verzichten. Zeiger werden beispielsweise als Argumente an Funktionen übergeben, damit die entsprechenden Variablen überschrieben werden können. Doch dazu später mehr.

3.1.3 Gültigkeit der Variablen

Alle Variablen müssen deklariert werden, aber wo überall kennt das Programm den Namen der Variablen? Das hängt davon ab, an welcher Stelle im Programm die Variablen deklariert werden. Grundsätzlich gilt: Der Name der Variablen ist nur in derjenigen Funktion sichtbar, in der er deklariert wurde. Wenn er dagegen außerhalb der Funktionen deklariert wurde, so gilt er für alle der Deklaration folgenden Funktionen.

Der folgende Quellcode zeigt den Unterschied:

```
int gl, x=5;

int func1(int f11)
{   int f12;  /* Anweisungen */ }

main()
{   int m, x=3, func2(int);  /* Anweisungen */ }

int func2(int f21)
{   int f22; /* Anweisungen */ }
```

Die Variable `gl` ist global gültig, d.h., sie ist für alle drei Funktion sichtbar und kann dort jeweils gelesen und verändert werden. Die Variablen `f11, f12, m, f21` und `f22` sind dagegen nur lokal sichtbar; sie gelten nur innerhalb der jeweiligen Funktionen und verlieren ihren Wert, sobald die zugehörige Funktion verlassen wird. Bei der Variablen `x` ist es komplizierter. Sie ist einmal global definiert und gilt deshalb für die Funktionen `func1` und `func2`. Aber in `main` wird derselbe Name für eine lokale Variable verwendet. Der Compiler legt nun zwei verschiedene Speicherplätze an und kennt innerhalb von `main` nur die lokale Variable `x`.

Auch bei Funktionen ist es ähnlich: Deren Sichtbarkeit hängt von der Stelle ihrer Definition ab. `func1` ist in `main` bekannt, da sie vorher definiert wurde. `func2` dagegen ist in `main` erst dann sichtbar, wenn sie dort wie im obigen Quelltext deklariert wurde.

Globale Variablen sind nützlich, wenn es Parameter gibt, die für alle Funktionen gelten sollen und nur selten geändert werden. Aber man verliert damit leicht den Überblick. Deshalb sollte man davon nur wenige verwenden und die Parameter besser als Argumente an die entsprechenden Funktionen übergeben. Insbesondere wenn Sie Ihre Funktionen auch in anderen Programmen verwenden wollen, sollten Sie alle Variablen vom Rest des jeweiligen Programmes abschirmen, also lokal definieren.

3.2 Operatoren

Bisher haben wir erfahren, wie Variablen deklariert und Konstante geschrieben werden. Nun wollen wir uns anschauen, wie man mit diesen Variablen und Konstanten rechnen kann. Die Verknüpfung von einem oder mehreren Werten zu einem neuen Wert nennt man *Operatoren*.

3.2.1 Arithmetische Operatoren

In C werden die Grundrechenarten wie in der Schulmathematik durchgeführt. Dabei können reelle oder ganzzahlige Werte — auch zusammen — verwendet werden. Die elementaren Operatoren sind in der Tabelle 3.3 zusammengefasst.

Tabelle 3.3: Grundrechenarten

Addition	`a + 5`
Subtraktion	`a - 5`
Multiplikation	`a * 5`
Division	`a / 5`
Restwert (Modulo)	`a % 5`

Bei der Division ganzer Zahlen wird nur der ganzzahlige Anteil ausgegeben, der Rest wird abgeschnitten. Wenn Sie dagegen das reelle Ergebnis haben wollen, sollten Sie beide Zahlen in `double` umwandeln, entweder mit einer Typenumwandlung oder bei einer Konstanten mit einem Dezimalpunkt:

```
(double)a / 5.
```

Beim Restwert-Operator dürfen nur ganze Zahlen benutzt werden; er liefert als Ergebnis den (ganzzahligen) Rest bei der Division der Zahl a durch 5. Zum Beispiel ergibt `40 % 5` den Wert 0, während `42 % 5` den Wert 2 liefert.

Wie in der Schulmathematik können Ausdrücke in runden Klammern zusammengefasst werden, und es gilt „Punkt-vor-Strichrechnung“. `(2+3)*4` liefert also den Wert 20, dagegen hat `2+3*4` den Wert 14.

Leider gibt es in C keinen Operator für die Potenzierung, sondern man muss dazu die Potenzfunktion aus der Mathematik-Bibliothek benutzen.

Potenz x^y : `pow(x,y)`

3.2.2 Logische Operatoren

Operatoren können auch die logischen Resultate Wahr (1) und Falsch (0) als Resultat liefern. Zum Vergleich zweier Zahlen `a` und `b` kennt C die Operatoren der Tabelle 3.4

Tabelle 3.4: Operatoren zum Vergleich zweier Zahlen

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

Vorsicht: Ein häufiger Anfängerfehler besteht darin, anstelle des logischen Gleichheitszeichens `==` den Zuweisungsoperator `=` zu schreiben! Das ergibt einen gültigen C-Ausdruck und damit keine Fehlermeldung des Compilers.

Für die Verknüpfung logischer Argumente `p` und `q` (ganze Zahlen mit den Werten Null für Falsch und ungleich Null für Wahr) gibt es folgende Operatoren:

Und : `p && q`
Oder : `p || q`
Nicht : `!p`

Der Operator `&&` liefert nur dann den Wert Wahr, wenn beide Argumente `p` und `q` den Wert Wahr haben, und der Operator `||` liefert nur dann Wahr, wenn eines oder beide Argumente Wahr sind.

Zusammengesetzte logische Rechnungen können Sie ebenso wie bei arithmetischen Ausdrücken klammern. Es gibt auch eine Standardreihenfolge der Auswertungen: Vergleiche werden zuerst ausgewertet, danach kommt Nicht, dann Und und zuletzt Oder.

Leider gibt es keinen Operator für das Exclusive Oder (Xor), das den Wert Wahr liefert, wenn nur eines der beiden Argumente `p` und `q` Wahr ist, aber nicht beide zusammen. Diesen Operator müssen Sie sich selbst definieren, beispielsweise durch

```
!p&&q || p&&!q
```

Die folgenden Beispiele sollen Sie mit den logischen Operatoren vertraut machen. Versuchen Sie zunächst selbst herauszufinden, ob der Wert des jeweiligen Ausdrucks 0 oder 1 ist.

```
int a=5, b=3, c=0, d=1;
```

1. `a<b || c<d`
2. `!(a<b)&&d`
3. `!c&&b<a || d&&a<b`
4. `!(a==b) && !d || !c&&d`
5. `!(c&&d) == (!c || !d)`
6. `!(c||d)==(!c&&!d)`

Beim ersten Ausdruck ist `a<b` FALSCH und `c<d` ist WAHR, daher gibt „FALSCH ODER WAHR“ den Wert WAHR (=1). Beim zweiten Ausdruck hat `!(a<b)` den Wert WAHR, ebenso wie die Zahl `d`, daher gibt „WAHR UND WAHR“ den Wert WAHR (=1). Ebenso kann man zeigen, dass 3. und 4. jeweils den Wert 1 haben. Die Ausdrücke 5. und 6. sind die DeMorganschen Regeln der logischen Algebra, sie sind für beliebige Argumente `c` und `d` richtig. Zusammenfassend haben also alle Ausdrücke den Wert 1.

3.2.3 Bitweise Operatoren

Wie schon erwähnt, können Sie in C maschinennah programmieren. Dazu gehört, dass Sie mit einzelnen Bits rechnen können. Jedes Bit einer Variablen kann die Werte 0 oder 1 annehmen, die entweder als WAHR oder FALSCH interpretiert werden können. Deshalb gibt es die entsprechenden logischen Operatoren, die bei zwei Variablen *bitweise* wirken.

Am besten verwenden Sie dazu nur vorzeichenlose ganze Zahlen, denn sonst können Sie Schwierigkeiten mit der Darstellung des Vorzeichens bekommen.

```
unsigned long  a,b;
```

Beide Variablen `a` und `b` haben damit je 32 Bits, die Sie mit den Operatoren aus der Tabelle 3.5 verknüpfen können.

Wir wollen die Wirkungen der Operatoren an kleinen Beispielen vorstellen, wobei nur die rechten acht Bit der Variablen gezeigt werden.

Mit dem Operator `&` können wir einzelne Bits einer Variablen ausschalten. Die Zahl 52 hat beispielsweise die Bit $(52 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)$

```
00110100
```

Tabelle 3.5: Bitweise logische Operatoren

Bitweises Und	`a&b`
Bitweises Oder	`a|b`
Bitweises Xor (Exclusives Oder)	`a^b`
Bitweises Nicht	`~a`
Bitweises Verschieben um b Bits nach links	`a<<b`
Bitweises Verschieben um b Bits nach rechts	`a>>b`

Wenn wir nun das fünfte Bit von rechts ausschalten wollen, so können wir die Bitfolge

```
00010000
```

definieren, also die Zahl 16, und die Verknüpfung `52 & ~16` bilden:

```
      52   :   00110100
     ~16   :   11101111
52 & ~16   :   00100100
```

Das Ergebnis ist die Zahl 36. Um ein Bit anzuschalten, kann man den Oder-Operator `|` benutzen. Wollen wir beispielsweise bei der Zahl 52 das zweite und vierte Bit anschalten, so verknüpfen wir sie mit der Zahl 10, die nur an den entsprechenden Stellen eine Eins hat:

```
      52   :   0010100
      10   :   0001010
52 | 10   :   0011110
```

Das Ergebnis ist die Zahl 30. Um die Bits einer Variablen `a` zu lesen, können wir mit `>>` die Bitkette um `n` Bit nach rechts schieben und das erste Bit mit `&1` lesen; dabei läuft `n` von Null bis zur Länge von `a` minus eins:

```
(a >> n) & 1
```

3.2.4 Bedingungsoperator

Es gibt in C einen Operator, der eine logische Bedingung auswertet und – abhängig von deren Ergebnis — einen von zwei möglichen Werten übergibt. Es ist der einzige Operator, der drei Argumente hat: einen logischen Wert und zwei numerische Werte. Dieser Bedingungsoperator hat die Form:

```
Logischer Wert ? Wert 1 : Wert 2
```

Wenn der logische Wert Wahr ist, dann wird der erste numerische Wert zurückgegeben, ansonsten der zweite. Zum Beispiel können Sie das Maximum zweier Zahlen folgendermaßen programmieren:

```
max = (a>b)? a : b;
```

Der nächste Ausdruck liefert den Betrag einer Zahl:

```
(x>0)? x : -x;
```

3.2.5 Zuweisung

Bisher haben wir erfahren, wie mit Konstanten und Variablen gerechnet werden kann. Nun sollen die Ergebnisse der Rechnungen gespeichert werden, d.h., sie sollen einer neuen oder schon verwendeten Variablen zugewiesen werden. Dazu dient der Operator `=`. Hierzu einige Beispiele:

```
int a = 5, c;
double b = 2.6E-3;

   c = 5*(a+2) + 3;
   a = a+2;
   a = b*1000;
```

Grundsätzlich wird immer die rechte Seite des Gleichheitszeichens ausgewertet und der Variablen auf der linken Seite zugewiesen. Dabei wird der rechte Wert auf den Typ der linken Variablen angepasst.

Die obige Zuweisung an `c` ist problemlos; die rechte Seite ergibt die Zahl 38, die der Variablen `c` vom Typ `int` zugewiesen wird. Bei der nächsten Zuweisung sehen Sie, dass das Rechenergebnis, für das die Variable `a` verwendet wurde, wieder derselben Variablen `a` zugewiesen werden kann. Denken Sie daran, das ist keine mathematische Gleichung. Wenn Sie die Gleichheit zweier Ausdrücke feststellen wollen, müssen Sie dagegen `a==a+2` verwenden; dieser Ausdruck liefert hier natürlich den Wert 0 für FALSCH.

Bei der letzten Zuweisung finden zwei Typenumwandlungen statt. Zunächst wird eine reelle mit einer ganzen Zahl multipliziert, und das Ergebnis ist eine reelle Zahl, hier mit dem Wert 2.6. Danach wird dieser Wert der ganzzahligen Variablen `a` zugewiesen. Dabei wird nur der ganzzahlige Anteil, hier der Wert 2, verwendet. Bei solchen Zuweisungen wird also nicht gerundet, sondern abgeschnitten.

Für einige Zuweisungen kennt C Abkürzungen. Folgende Ausdrücke sind zeilenweise identisch:

```
a = a+b;     a += b;
a = a+1;     a += 1;    a++;
a = a-1;     a -= 1;    a--;
```

Bei der Abkürzung `a+=b` können Sie anstelle des Operators `+` auch viele andere Operatoren verwenden. Bei den Abkürzungen `a++` und `a--` wird erst der Wert von `a` verwendet und zurückgegeben, danach wird die Variable `a` um den Wert eins erhöht. Beim folgenden Beispiel erhalten die Variablen `b` und `c` also jeweils die Werte 10 und 12.

```
int a=5, b, c;
b = 2*a++;
c = 2*a;
```

Die Abkürzung `a++` wird oft bei Schleifen verwendet; dabei ist a ein Zähler, der schrittweise bis zu einem Endwert erhöht wird.

3.3 Verzweigungen und Schleifen

Ein Computer kann nicht nur rechnen, sondern er kann die Rechnungen sehr schnell und sehr oft durchführen und dabei auf die Ergebnisse der einzelnen Rechenschritte reagieren. Dazu sind zwei Arten von Anweisungen erforderlich: Verzweigungen (bedingte Anweisungen) und Schleifen (Wiederholungen). Fast alle Programme enthalten diese beiden Grundelemente jeder Programmiersprache, die in C in verschiedenen Formen geschrieben werden können.

3.3.1 Bedingte Anweisungen

Die am häufigsten verwendete Form einer Verzweigung ist die `if`-Anweisung:

```
if(test) {Aktion1} else {Aktion2}
```

Diese Anweisung erklärt sich fast von selbst: Wenn `test` den Wert WAHR hat, dann werden die Anweisungen des ersten Blocks ausgeführt, sonst die des zweiten Blocks. Den Teil `else { ...}` können Sie auch weglassen. Falls der jeweilige Block nur aus einer einzigen Anweisung besteht, können Sie die geschweiften Klammern ebenso weglassen. Hierzu einige Beispiele

```
if(a<b) printf(" a<b \n");   else printf(" a>=b \n");
if(x<0) { x=-x; y=sqrt(x);}  else y=sqrt(x);
if(x==100) zaehler++;
```

Vorsicht: 1. Jede einzelne Anweisung, auch die letzte in einem Block, muss mit einem Semikolon abgeschlossen werden, sonst meldet der Compiler `parse error`.
2. Sollten Sie versehentlich in der letzten Anweisung `if(x=100)` schreiben, so meldet der Compiler keinen Fehler. Die Zuweisung `x=100` gibt den Wert 100 zurück, der als WAHR interpretiert wird. In diesem Fall wird daher die Anweisung `zaehler++` bei jedem Aufruf des `if`-Befehls ausgeführt.

Falls Sie je nach Wert einer Variablen mehrfache Verzweigungen programmieren möchten, können Sie dazu — anstelle von `if-else`-Befehlen — die `switch`-Anweisung verwenden. Sie lautet

```
switch(Ausdruck)
   { case K1: Anweisungen_1; break;
     case K2: Anweisungen_2; break;
     case K3: Anweisungen_3; break;
```

```
    ...
    default: Anweisungen;
}
```

Zunächst wird der Ausdruck ausgewertet. Falls er mit einer der Konstanten K1, K2,... übereinstimmt, werden die Anweisungen hinter dem entsprechenden `case`-Befehl ausgeführt. Andernfalls werden die `default`-Anweisungen ausgeführt; sie können auch weggelassen werden. Vergessen Sie nicht die abschließenden `break`-Anweisungen, sonst werden die folgenden Befehle bis zum nächsten `break` ausgeführt. Im folgenden Beispiel werden je nach Wert des Zeichens `c` verschiedene Anweisungen ausgeführt:

```
switch(c)
   { case '*': d=a*b; break;
     case '+': d=a+b; break;
     case '-': d=a-b; break;
     case '/': d=a/b; break;
     default : printf(" Operator %c unbekannt \n",c);
   }
```

3.3.2 Schleifen

Für häufige Wiederholungen von Anweisungen gibt es die `while`, `for` und `do`-Befehle. Die `while`-Schleife erklärt sich von selbst:

```
while(test) {Anweisungen}
```

Solange der Wert von `test` wahr (ungleich Null) ist, wird der Anweisungsblock ausgeführt.

```
run=100;
while(run--) printf("%i\n",run*run);
```

Mit diesen Anweisungen werden die Quadrate der Zahlen 100 bis 1 ausgedruckt.

```
sum=0.;
while( (r=drand48()) <= 0.99 )
  {  zaehler++;
     sum+=r;
  }
```

Hierzu erzeugt die Funktion `drand48()` eine reelle gleichverteilte Zufallszahl vom Typ `double` im Intervall von 0 bis 1. Die `while`-Schleife erzeugt so lange Zufallszahlen, bis deren Wert größer als 0.99 ist, diese Werte werden gezählt und aufaddiert.

Die `for`-Schleife wird noch häufiger verwendet als die `while`-Schleife, allerdings muss man sich erst an ihre Form gewöhnen. Sie lautet

```
for (start;test;inkrement) {Anweisungen}
```

Beim Aufruf dieser Schleife wird zunächst die Startanweisung ausgeführt. Danach wird der Ausdruck `test` ausgewertet; wenn er den Wert WAHR (ungleich Null) hat, wird der Anweisungsblock ausgeführt. Danach wird die Anweisung `inkrement` ausgeführt und wieder der Ausdruck `test` überprüft, gegebenenfalls mit anschließender Ausführung der Anweisungen. Dies wird nun so lange wiederholt, bis `test` den Wert FALSCH (Null) hat.

Am häufigsten werden wir die `for`-Schleife benutzen, um eine feste Anzahl von Wiederholungen durchzuführen:

```
for( i=0; i<100; i++) printf(" %i \n", i*i);
```

Hier werden die Quadratzahlen von 0 bis 99 ausgedruckt. Die `for`-Schleife kann auch verschachtelt werden:

```
for( i=0; i<100; i++)
for( j=0; j<100; j++) matrix[i][j]=i-j;
```

Dieses Beispiel weist den Elementen der 100 $\times$ 100 Matrix `matrix` die Werte i−j zu. Weil hinter dem ersten `for()` nur ein einziger Befehl steht, nämlich das zweite `for()...;` brauchen wir keine geschweiften Klammern.

Die dritte Form einer Schleife ist der Befehl

```
do {Anweisungen} while(test)
```

bei dem der Anweisungsblock mindestens einmal durchlaufen wird.

Manchmal möchten Sie die Schleife abbrechen, bevor sie abgelaufen ist. Dazu gibt es zwei Befehle: `continue` und `break`. Der erste Befehl überspringt alle Anweisungen bis zum Ende des Blocks und fährt dann mit der Schleife fort. Der `break`-Befehl dagegen springt sofort aus dem Block heraus zur folgenden Anweisung und beendet die Schleife nicht.

3.4 Felder und Strukturen

Sie können nicht nur für einzelne Speicherplätze, sondern auch für ganze Bereiche des Speichers einen Namen definieren und damit den entsprechenden Platz im Computer reservieren. Solche Speicherbereiche werden *Felder* (auch *Arrays*) genannt.

3.4.1 Felder

Jedes Feld muss am Anfang des Programmes deklariert werden, dazu müssen der Typ der einzelnen Speicherplätze und die Länge des Feldes angegeben werden:

```
int a[100];
double b[10];
```

Damit stehen Ihnen 100 Plätze für ganze Zahlen und 10 Plätze für relle Zahlen zur Verfügung, mit insgesamt $100 \cdot 4 + 10 \cdot 8 = 480$ Bytes Speicher. Diese Plätze haben einen Index, der bei dem Wert 0 beginnt:

```
a[0], a[1],..., a[99]
b[0], b[1],..., b[9]
```

Vorsicht: Ein verbreiteter Anfängerfehler besteht darin, von 1 bis 100 bzw. 10 zu indizieren. Der Compiler meldet dann keinen Fehler, sondern das Programm überschreibt unbekannte Plätze im Speicher.

Im folgenden Beispiel werden im Feld `a` die Quadrate von 1 bis 100 gespeichert:

```
for( i=0; i<100; i++) a[i]=(i+1)*(i+1);
```

Bei den meisten Compilern werden alle Elemente eines Feldes automatisch mit dem Wert Null initialisiert. Bei der Deklaration eines Feldes können Sie dessen Werte aber auch selbst initialisieren:

```
int c[] = {3,8,10,1,9};
```

Das erspart Ihnen das Schreiben der sechs Anweisungen:

```
int c[5];
  c[0] = 3;
  c[1] = 8;
  c[2] = 10;
  c[3] = 1;
  c[4] = 9;
```

Bei der obigen Deklaration wird die Länge des Feldes aus der Länge der Initialisierung bestimmt.

```
int c[9]={3,8,10,1,9};
```

dagegen würde ein Feld der Länge 9 reservieren, die Werte von `c[0]` bis `c[4]` entsprechend zuweisen und `c[5]` bis `c[8]` auf Null setzen. Eindimensionale Felder werden im mathematischen Bereich häufig zur Darstellung von Vektoren benutzt. Die folgenden Anweisungen berechnen beispielsweise das Skalarprodukt `produkt` zweier zehndimensionaler Vektoren $\vec{v}$ und $\vec{w}$.

```
produkt = 0;
for( i=0; i<10; i++) produkt += v[i]*w[i];
```

3.4.2 Zeichenketten

In Feldern werden auch auch Texte, also Zeichenketten, gespeichert. Dabei ist eine Zeichenkette eine Folge von ASCII-Zeichen, die von dem Nullzeichen ('\0') beendet wird. Eine Zeichenkette kann auch als zusammenhängender Text geschrieben werden, der von doppelten Anführungszeichen ("...") begrenzt wird. Wenn wir beispielsweise den Text *Guten Morgen!* im Feld `gm` speichern wollen, so können wir

```
char gm[] = "Guten Morgen!\n";
```

schreiben. Dann wird ein Feld mit 15 Speicherplätzen vom Typ `char` (8 Bit Länge) mit dem Namen `gm` reserviert, das folgende Zeichen enthält:

```
'G' 'u' 't' 'e' 'n' ' ' 'M' 'o' 'r' 'g' 'e' 'n' '!' '\n' '\0'
 0   1   2   3   4   5   6   7   8   9   10  11  12  13   14
```

Sie könnten auch — wie oben bei den Zahlen — jedes einzelne Zeichen initialisieren oder sogar einzeln zuweisen, aber einfacher geht es natürlich als Zeichenkette. Mit den Funktionen `printf` und `scanf` zum formatierten Schreiben und Lesen von Zahlen können Sie beispielsweise mit dem Steuerzeichen `%s` ganze Zeichenketten lesen und schreiben:

```
char str[100];
 printf(" Text eingeben: \n");
 scanf("%s",str);
 printf(" Sie haben %s eingetippt. \n",str);
```

Eine Zeichenkette ist übrigens ein Ausdruck, der die Adresse des entsprechenden Textes enthält. Deshalb kann sie nur einer Zeigervariablen zugewiesen werden, aber nicht einem Feld, dessen Adresse schon festgelegt ist.

```
char str[100], *text;
 str  = "Hallo";          /* FEHLER !  */
 text = "Hallo";          /* RICHTIG ! */
 printf(" %s Freunde. \n",text);
```

Zur Bearbeitung von Zeichenketten gibt es eine Vielzahl von Funktionen, die Sie mit `man -k string` finden können.

3.4.3 Matrizen

Bei mathematischen Beschreibungen treten oft Größen auf, die nicht nur einen, sondern mehrere Indizes haben. Auch solche Größen — mit beliebig vielen Indizes — lassen sich in C definieren. Wir wollen hier nur Matrizen diskutieren, also Größen mit zwei Indizes. Zum Beispiel soll folgende Matrix gespeichert werden:

$$\begin{pmatrix} 8 & 2 & 0 & 0 \\ -4 & 3 & -2 & 0 \\ 1 & 1 & 0 & -1 \end{pmatrix}$$

Sie hat drei Zeilen und vier Spalten, deshalb deklarieren wir ein Feld

```
int a[3][4];
```

und weisen die Werte der Matrix den entsprechenden Elementen zu:

```
a[0][0]=8;   a[0][1]=2;  ...  a[2][3]=-1;
```

Alternativ dazu können wir auch schon bei der Deklaration die Werte initialisieren:

```
int a[3][4] = { {8,2,0,0}, {-4,2,-2,0}, {1,1,0,-1} }
```

Leider müssen Sie in C alle Matrixoperationen selbst programmieren. Hierzu ein kleines Beispiel: `b` sei eine weitere 4×3-Matrix und `v` ein Vektor mit vier Elementen. Dann wollen wir die beiden Matrizen `a` und `b` miteinander multiplizieren und ebenso `a` mit `v`:

$$c_{ij} = \sum_{k=1}^{4} a_{ik} b_{kj} \qquad w_i = \sum_{k=1}^{4} a_{ik} v_k$$

Diese Gleichungen lauten in C:

```
for( i=0; i<3; i++)
for( j=0; j<3; j++)
 {  sum=0.;
    for( k=0; k<4; k++) sum+= a[i][k]*b[k][j];
    c[i][j] = sum;
 }

for( i=0; i<3; i++)
 {  sum=0.;
    for( k=0; k<4; k++) sum+= a[i][k]*v[k];
    w[i] = sum;
 }
```

Denken Sie daran, vorher alles zu deklarieren:

```
int i, j, k, a[3][4], b[4][3], c[3][3], v[4], w[3];
```

Vorsicht: 1. Auch bei Matrizen laufen beide Indizes von 0 bis n−1!
2. Indizes werden in doppelte eckige Klammern gesetzt, wie bei `a[2][1]`. `a[2,1]` gibt keine Fehlermeldung, sondern liefert die Adresse der zweiten Zeile, also `a[1]`.

Abschließend wollen wir noch erwähnen, dass auch Felder Adressen im Speicher besitzen. Jedes Feld — auch ein mehrdimensionales — ist als Kette von Bits im Speicher abgelegt. Wenn der Rechner also den Anfang der Kette und die Dimensionen des Feldes kennt, so kann er auf alle Elemente des Feldes zugreifen. Die Anfangsadresse eines Feldes steht in einem Zeiger, der den Namen des Feldes trägt. Bei den Deklarationen

```
int v[10], a[3][4];
```

sind daher `v` und `a` Adressen, die beispielsweise an Funktionen übergeben werden können. Diese Funktionen können dann auf die Elemente der Felder zugreifen.

Beim eindimensionalen Feld `v` enthält die Variable `v` die Anfangsadresse einer Folge von 10 ganzzahligen Speicherplätzen, `v` ist daher identisch zu `&(v[0])`. Beim zweidimensionalen Feld `a` dagegen enthält die Variable `a` die Anfangsadresse eines Feldes der drei

Zeiger `a[0]` bis `a[2]`. Jeder dieser Zeiger enthält die Adresse der drei Zeilen der Matrix; `a[1]` ist daher die Anfangsadresse des zweiten Zeilenvektors `(a[1][0],..., [1][3])`. Sie können diese Konstruktion benutzen, um einzelne Zeilen einer Matrix an eine Funktion zu übergeben. Hier noch einmal die einzelnen Variablen für die (3×4)-Matrix `a`:

$a \downarrow$				
$a[0] \longrightarrow$	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1] \longrightarrow$	$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2] \longrightarrow$	$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[1][3]$

Diese Konstruktion zeigt, dass Felder auch als Zeiger von Zeigern definiert werden können:

```
int **a;
```

Die Variable `a` enthält damit eine Adresse von einem Feld von Adressen, und jede davon zeigt auf ein Feld von ganzen Zahlen. Allerdings existieren durch diese Deklaration noch keine Felder von Adressen und ganzen Zahlen, sie müssen erst angelegt werden. Das macht die Funktion `malloc`. Aber diese Konstruktionen gehören zu den fortgeschrittenen Programmiertechniken und sollen hier nur erwähnt werden.

3.4.4 Strukturen

Strukturen sind Erweiterungen der Feldkonstruktionen. Ebenso wie Felder bezeichnen Sie einen ganzen Bereich von Speicherplätzen, allerdings können der Typ und die Größe der zugehörigen Variablen unterschiedlich sein. Das ist bei Datenbanken besonders wichtig, wo beispielsweise Name, Geburtsdatum und Gehalt eines Angestellten mit einer einzigen Variablen zusammengefasst werden sollen. Aber auch im naturwissenschaftlichen Bereich werden Strukturen eingesetzt, etwa um die Koordinaten des Ortes und der Geschwindigkeit eines Teilchens unter einem einzelnen Begriff zu speichern, anstatt für jedes Teilchen sechs Variablen zu benutzen.

Mit Strukturen haben Sie außerdem die Möglichkeit, eigene Datentypen zu definieren. Am besten wir zeigen das an einem Beispiel aus der Physik. Wir definieren einen neuen Typ `atom`, der die Masse des Atoms und die (x,y,z)-Koordinaten des Ortes und der Geschwindigkeit enthält. Zunächst benennen wir eine einzige Variable `teilchen` von diesem Typ `atom`:

```
struct atom{ char * sorte;
             float masse, r[3], v[3];
           } teilchen;
```

Diese Anweisung hat also die Form

```
struct Typ {Deklarationen} Name;
```

wobei entweder Typ oder Name weggelassen werden können. Nun können wir unter dem Namen `teilchen` auf alle Komponenten zugreifen, dazu dient das Zeichen Punkt (.):

```
teilchen.sorte = "Silizium";
teichen.masse = 28.086;
teilchen.r[0] = 120.;
teilchen.v[2] = 12.;
...
```

Selbstverständlich wollen wir nicht nur ein einziges, sondern viele miteinander wechselwirkende Atome berechnen. Dazu können wir ein Feld von Strukturen definieren:

```
struct atom t[1000];
```

Nun haben wir ein Feld für tausend Teilchen, und zu jedem gehören die entsprechenden Variablen, die wir vorher definiert haben. Beispielsweise enthält die Variable `t[5].v[2]` die z-Komponente der Geschwindigkeit des sechsten Teilchens.

3.5 Funktionen

Funktionen sind ein vielseitiges Werkzeug der modernen Programmierung. Funktionen können dieselbe Rechenvorschrift auf verschiedene Argumente anwenden, sie geben dem Programm eine übersichtliche Struktur und können sich selbst wieder aufrufen. Ein C-Programm besteht aus einer Folge von Funktionen. Es muss wenigstens die Funktion `main` enthalten, die beim Aufruf des kompilierten Programmes zuerst ausgeführt wird. `main` kann dann weitere Funktionen aufrufen, die wiederum andere aufrufen können.

Oberhalb von `main` werden gegebenenfalls zusätzliche Funktionen definiert:

```
Typ Name (Typen und Namen der Variablen) {Anweisungen}
```

Jede dieser Funktionen hat ein oder mehrere Argumente und gibt einen Wert zurück. Sie wird mit ihrem Namen aufgerufen:

```
Name (Werte der Variablen);
```

Dieser Ausdruck steht für den berechneten Wert der Funktion, der wie ein üblicher Wert je nach Typ weiterverarbeitet werden kann. Hierzu ein einfaches Beispiel: Eine Funktion `betrag` soll den Betrag des Arguments zurückgeben.

```
double betrag(double x)
{
     if(x<0) x=-x;
     return x;
}

main()
{
     double x;
```

```
        x = -1.23E-5;
        printf(" Der Betrag von %f ist %f \n", x, betrag(x));
}
```

In der Definition der Funktion `betrag` wird der Wert, der beim Aufruf der Funktion zurückgegeben wird, mit der `return`-Anweisung bestimmt. Eine Funktion kann mehrere `return`-Anweisungen enthalten; sobald ein derartiger Befehl bearbeitet wird, wird die Funktion beendet und der entsprechende Wert zurückgegeben. Fehlt dagegen die `return`-Anweisung, wird die Funktion nach dem Anweisungsblock ohne Rückgabe eines Wertes beendet.

In der Funktion `main`, die hier ohne Typ und Argumente definiert wird, wird die Funktion `betrag` aufgerufen und ihr Wert wird sofort an die Funktion `printf` weitergereicht.

Beachten Sie, dass die Variable `x` in beiden Funktionen lokal definiert ist. Für das Symbol `x` werden daher — trotz des gleichen Namens — zwei verschiedene Speicherplätze reserviert. Wenn die jeweilige Funktion verlassen wird, so ist die entsprechende Variable wieder undefiniert. Sie können aber auch eine globale Variable `x` definieren, die für das gesamte Programm gilt. Das gleiche Beispiel lautet dann

```
double x = 1.23E-5;

void betrag()
{     if(x<0) x=-x; }

main()
{
        double x_alt;
        x_alt = x;
        betrag();
        printf(" Der Betrag von %f ist %f \n", x_alt, x);
}
```

Jetzt hat die Funktion `betrag` den Typ `void`, d.h., sie gibt keinen Wert zurück. Sie hat auch kein Argument, sie ist daher nur eine Abkürzung für einen Anweisungsblock. Dennoch ist diese Konstruktion besonders dann sinnvoll, wenn der Block aus vielen Anweisungen besteht. In diesem Fall kann damit die aufrufende Funktion, hier `main`, sehr übersichtlich geschrieben werden.

In den beiden obigen Beispielen haben wir die Funktion `betrag` *vor* der Funktion `main` definiert. Wir können jedoch Funktionen auch *hinter* `main` definieren, allerdings kennt dann `main` diese Funktionen noch nicht; sie müssen dort erst deklariert werden. Sowohl der Typ der Funktion als auch der aller ihrer Argumente müssen deklariert werden, wie in der dritten Version unseres Beispiels zu sehen ist:

```
main()
{
        double x, betrag(double);
```

```
    x = -1.23E-5;
    printf(" Der Betrag von %f ist %f \n, x, betrag(x));
}

double betrag(double x)
{
    if(x<0) x=-x;
    return x;
}
```

3.5.1 Übergabe der Argumente

Grundsätzlich gilt in C: Die Argumente einer Funktion werden nur mit ihrem Wert übergeben. Mit dem Aufruf `betrag(y)` wird nur der Wert der Variablen `y` an die lokale Variable `x` der Funktion übergeben; Sie haben daher keine Möglichkeit, die Variable `y` zu ändern. Was machen Sie aber, wenn Sie beispielsweise den Inhalt der Variablen `x` und `y` mit einer Funktion `tausche` ändern wollen? Der Trick ist: Sie übergeben die *Adressen* von `x` und `y`, denn dann kennt die Funktion die zu bearbeitenden Speicherplätze. Hier ist das Beispiel:

```
void tausche(double* xa, double* ya)
{
  double z;
    z = *xa;
  *xa = *ya;
  *ya = z;
}

main()
{
  double x=5., y=10.;
  tausche(&x, &y);
  printf(" x=%.1f, y=%.1f \n", x, y);
}
```

In `main` werden beim Aufruf von `tausche` die Adressen von `x` und `y`, also `&x` und `&y` übergeben. Die Argumente `xa` und `ya` von `tausche` sind Adressen auf Variablen vom Typ `double`. Der Wert bei der Adresse `xa` wird mit `*xa` bezeichnet und wird in der Variablen `z` zwischengespeichert. Danach wird der Wert bei der Adresse `ya` an den Speicherplatz mit der Adresse `xa` geschrieben, das geschieht mit der üblichen Zuweisung der Werte `*xa = *ya.` Schließlich wird der Zwischenspeicher an den Platz mit der Adresse `ya` zugewiesen. Hier sehen Sie noch einmal die Bedeutung der x-Variablen:

`x`	Wert der Variablen x
`&x`	Adresse der Variablen x
`xa`	Adresse der Variablen x
`*xa`	Wert der Variablen x