Backen mit C....3
Geschichte....4
BCPL 4
B....4
C....4
C++ 4
Precompiler, Compiler und Linker....5
Ade, schnöde Welt....5
Precompiler (Präprozessor)....5
Compiler....5
Keywords....6
C89 & C90 Keywords....6
C95 Keywords....6
C99 Keywords....6
C++ Keywords....6
C89 & C90 header files....6
C95 header files:....6
C99 header files:....6
Begriffsdefinitionen....7
Funktion (function)....7
Zahlensysteme....10
dezimale Zahlensystem....10
duale Zahlensystem....10
Byte Order....10
Binäre Kodierung von Integer-Zahlen....11
big endian....11
little endian....11
Binäre Kodierung von Fließkommazahlen 12
Binäre Kodierung von Fließkommazahlen nach ANSI 754....12
Binäre Kodierung von Fließkommazahlen als Tempreal:....13
Escape Sequenzen....14
Variablen & Datentypen....15
Begriff »Variable«....15
Variblename....15
Deklarierung einer Variable....15
Unterschied zwischen Deklaration und Definition....16
Initialsierung einer Variable....16
Nummerische Variablen im Vergleich zu Stringvariablen....16
Lokale und globale Variablen....18
Ungarische Notation (Hungarian Notation)....18
Datentypen....19
Begriff »Datentyp«....19
Grund-Datentypen....19
Qualifier....19
Typumwandlung (type conversion)....20
Explizite Typumwandlung....20
Operatoren nach Anzahl ihrer Operanden....22
Position des Operators....22
Arithmetische Operatoren -, +, *, / und %....22
Vergleichsoperatoren ==, >, <, <= und !=....23
logische Operatoren !, && und | | 23
logische Variablen 23
konditionale (bedingte) Operatoren ? : 24
Sequenzoperator (Kommaoperator) 25
sizeof Operator....25
Indirektionsoperator (Verweisoperator) *....25
Adressoperator &....26
Schiebeoperatoren << >> 27
left-shift Operator <<....27
right-shift Operator >>....28
Setzen und löschen eines Bits....28
Assoziativgesetz....29
Basic I/O Funktionen....31
getchar()....31
putchar()....31
gets()....31
puts()....31
fputs()....32

printf()....32
scanf()....33
atoi()....33
Integer-Konstante....34
Fließkomma-Konstante....34
Zeichenkonstante....35
Zeichenkettekonstante....35

Kontrollstrukturen....36

Verzweigung....36
If Anweisung....36
If else Anweisung....37
switch Anweisung....39
for-Schleife....40
while-Schleife....40
do-Schleife....40

Felder (Arrays)....41

Array-Initialsierung und Zugriff auf Arraywerte....41

Zeiger (Pointer)....43

Near Pointer & far pointer....43
Zeiger-Operationen....43
Deklaration eines Zeigers mit dem Indirektionsoperator....43
Initialisierung des Zeigers....44
Wild Pointer & Null Pointer....44
Array als Zeiger....44
Zeiger-Arithmetik....45
Array an eine Funktion übergeben....45

Backen mit C

C-Manual in german written by
Jerboa 2007-06-24
v0.0.86
jerboa@aussiemail.com.au

Geschichte

BCPL

BCPL steht für Basic Combined Programming Language und wurde Mitte der 60er von Martin Richard an den Universitäten von Cambridge und London entwickelt.

B

Ken Thompson, der zusammen mit Dennis Ritchie für Bell Laboratories arbeitete, war zur gleichen Zeit damit beschäftigt, ein neues Betriebssystem für den DEC-PDP-7 Computer zu entwickeln. (Dieses Betriebssystem wurde 1970 von Brian Kernighan auf den Namen Unix getauft.)
Bis dahin war Unix in Assembler programmiert. Um die Programmierung zu vereinfachen, suchte Thompson nach einer wartungsfreundlichen Sprache. Da er keine passende fand, nahm er BCPL und reduzierte es zu B (B für Bell). Der Interpreter B war so sehr an die Hardware des DEC-PDP-7 angepasst, dass Thompson, als 1970 der PDP-11 herauskam, Unix wieder in Assembler aufrüstete.
Danach portierte Dennis Ritchie B auf die neue Maschine. Da der PDP-11 mit mehr Speicher als der PDP-7 ausgestattet war, konnte Ritchie B als Compiler implementieren und wurde »new B« genant. Dieses neue B wurde bald als C bezeichnet.
1972 wurde Unix zum ersten mal in C implementiert.

C

1978 veröffentlichten Ritchie Kernighan und Brian Kernighan »The C Programming Language« (aka “the White Book” und »K&R C«).
1983 rief das American National Standard Institute (ANSI) ein Komitee (namens X3J11) aus Software- und Hardwareentwicklern zusammen, um einen C Standard festzulegen: C89 (also 6 Jahre später; aka »ANSI X3.159-1989 "Programming Language C"« und »ANSI C«)
Dieser Standard wurde mit geringfügigen Änderungen ein Jahr später von der International Standards Organisation (ISO) als »ISO/IEC 9899:1990« übernommen. Teil des ANSI C Standards sind auch die Bibliotheken namens »ANSI C standard library«.

Eine kleine Revision von C89 (ISO-646 Zeichen und die Unterstützung erweiterter Zeichensätze) erschien 1993 unter der Bezeichnung »Amendment 1«, »AM1« und »C93«. Wurde auch veröffentlicht als ISO/IEC 9899-1:1994.

1995 kamen die Headerdateien <iso646.h>, <wchar.h> und <wctype.h> hinzu (bekannt als »Normative Amendment1«, kurz NA1).

1999 veröffentlichte die International Standards Organisation »C99« (ISO/IEC 9899). C99 beinhaltet etwa die C++ Möglichkeit Variablen an jeder Stelle in einem Block zu definieren (und nicht nur am Blockanfang). Neue Headerdateien namens <complex.h>, <fenv.ht>, <inttypes.h>, <stdbool.h>, <stdint.h> und <tgmath.h>

- Macros mit einer variablen Anzahl von Argumenten
- variable-length arrays
- official support for one-line comments beginning with //, borrowed from C++
- several new library functions, such as snprintf()
- several new header files, such as stdint.h
- inline functions
- addition of several new data types, including long long int (to reduce the pain of the looming 32-bit to 64-bit transition), an explicit boolean data type, and a complex type representing complex numbers

C++

C++ wurde von Bjarne Stroustroup Anfang bis Mitte der 80er entwickelt. Zuerst entwickelte er einen Pre-Compiler, um Objekt-Orientierte C-Programme in "normale" C Programme zu übersetzen. Diesen nannte er »C with classes«. Danach entwickelte er eine neue, aber weitgehend mit C kompatible Sprache, nämlich C++. Auf einem PC war C++ ab 1988 verfügbar.
1998 veröffentlichte ANSI einen C++ Standard.

Precompiler, Compiler und Linker

Ade, schnöde Welt

```
#include "stdio.h"
int main (void)
  {
    puts ("ade, schnoede welt!");
    return 0;
  }
```

Die erste Anweisung ist `#include`. An dieser Stelle liest der Compiler die Datei `stdio.h`. Sie ist ein Header-File (hat meist die Dateiendung .h) und enthält Informationen zur Standard-Ein- und Ausgabe (standard input output).
Das # leitet eine Anweisung für den Präprozessor ein. Der Präprozessor ist der erste Teil eines Compilers. Er übersetzt nicht, sondern nimmt einfache Textbearbeitungen vor. Nach dem # kommt der Name der einzulesenden Datei. Sie steht in Anführungszeichen oder in spitzen Klammern:

```
#include "foo.h"
#include <foo.h>
```

C Compiler arbeiten im in drei Schritten, welche in der Regel durch 3 Teilprogramme ausgeführt werden:
1) dem Precompiler, 2) dem Compiler und dem 3) Linker.

Precompiler (Präprozessor)

Der Precompiler ist ein Programm, das den Sourcecode (Endung ».c«, bei C++ ».cpp«) durchgeht und für ihn vorgesehene Befehle (Präprozessor-Direktive) ausführt. Diese Befehle beginnen meist mit #. Erkennt der Precompiler zum Bespiel #include <xy.h>, so ersetzt er an dieser Stelle den Code von #include <xy.h> durch den Text des Files xy.h . So wächst der Sourcecode entsprechend, bis am Ende ein Quelltext steht, der an den Compiler übergeben wird.
Die Datei, die dabei eventuel angelegt wird, hat meist die Endung »*.i«.

Compiler

Der Compiler übersetzt nun den Quelltext in ein sogenanntes Objektformat. Diese Files haben in der Regel die Endung ».o« (Unix) oder ».obj« (DOS).
Diese Objektdatei enthält den ausführbaren Maschinencode und Referenzen zu Code-Teilen, die erst noch durch den Linker aufgelöst werden müssen. Diese binären Dateien sind maschinenabhängig und lasssen sich (für gewöhnlich) nicht auf anderen Rechnertypen ausführen.

Linker

Der Linker (Binder) kann nun ein oder mehrere dieser Objektdateien zusammenfügen und ein startbares (executable) Programm (Zb »foo.out« oder unter Windows »foo.exe«) generieren.
Beim gcc läss sich mit dem Parameter –o eine frei wählbare Endung bestimmen:

```
gcc -o foo foo.c
gcc -o foo.exe foo.c
```

Angeblich dürfen Präprozessorbefehle nicht nach rechts eingerückt werden, sondern müssen in der ersten Spalte stehen.

Wenn in einer Fehlermeldung neben der Fehlernummer ein C steht und/oder die Zeilennummer, kommt sie vom Compiler. Steht neben der Fehlernummer ein L und/oder ein Unterstrich beim beanstandeten Wort, kommt die Fehlermeldung meist vom Linker (bei fehlenden oder fehlehaften Modulen).

Source Code (Quelltext)

Der Quelltext (auch Quellcode genannt) ist der vom Programmierer geschriebene Programmtext. Der Compiler erstellt aus diesem den Objektcode (Maschinenanweisungen) und übergibt diesen dem Linker (der aus dem Objektcode ein ausführbares Programm erstellt).
Im Quelltext sind diese Zeichen erlaubt:

- die zehn Dezimalziffern: 1 2 3 4 5 6 7 8 9 0
- die Groß- und Kleinbuchstaben des englischen Alphabets:
  a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- die folgenden Grafiksymbole: ! " % & / ( ) [ ] { } \ ? = ' # + * ~ - _ . : ; , | < > ^
- die Whitespace-Zeichen: Leerzeichen, Tabulatorzeichen, neue Zeile, neue Seite
- bei manchen Compilern (nicht ANSI C) kann auch das Dollarzeichen $ verwendet werden.

Keywords

Keywords (auch »reserved words« genannt) sind Anweisungen für den Compiler und können vom User nicht verändert oder als Variablen verwendet werden.

C89 & C90 Keywords

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

C95 Keywords

C++ bietet folgende Keywords als Synonyme für punctuation tokens:

```
and      &&        compl    ~         or_eq    |=
and_eq   &=        not      !         or_eq    |=
bitand   &         not_eq   ! =       xor      ^
bitor    |         or       ||        xor_eq   ^=
```

Die in C95 eingeführte Headerdatei <iso646.h> enthält gleichnamige Definitionen für Macros (die sich wie die oben genannten Keywords verhalten).C-Code die nicht diese Headerdatei inkludieren, können somit diese Bezeichnungen frei verwenden.

C99 Keywords

Mit C99 wurden folgende (für C++ unbekannte) Keywords definiert:

```
_Bool         Inline
_Complex      _Pragma
_Imaginary    Restrict
```

C++ Keywords

Folgende C++ Keywords sind C99 unbekannt:

```
asm            friend              template
bool           mutable             this
catch          namespace           throw
class          new                 true
const_cast     operator            try
delete         private             typeid
dynamic_cast   protected           typename
explicit       public              using
export         reinterpret_cast    virtual
false          static_cast         wchar_t
```

Header-Dateien

C89 & C90 header files

```
assert.h    locale.h    stddef.h
errno.h     math.h      stdio.h
fenv.h      setjmp.h    stdlib.h
float.h     signal.h    time.h
limits.h    stdarg.h
```

C++ deckt alle C89 Headerdateien ab. Da sie deprecated sind, sollten ihre jeweiligen neuen Entsprechungen verwendet werden, die mit c beginnen <cmath>, <cstddef>, <cstdio>, <cstdlib>, …

C95 header files:

```
iso646.h    wchar.h    wctype.h
```

C99 header files:

```
complex.h    inttypes.h    stdint.h
fenv.ht      stdbool.h     tgmath.h
```

Begriffsdefinitionen

case-sensitive

C ist case-sensitive (unterscheidet zwischen Groß- und Kleinschreibung).
Beispiel: Variable namens »foo« ist nicht ident mit Variable namens »Foo«. Bei Keywords ist die Art der Schreibung irrelevant.

Funktion (function)

Eine Funktion ist ein Unterprogramm (sub-routine), das mit der Angabe seines Namens aufgerufen wird.
Bei C++-Objekten werden Funktionen auch »Methoden« genannt.
Beginn und Ende der Funktion werden mit dem Keyword function und zwei geschweiften Klammern definiert.

```
function Funktionsname (Parameter1, Parameter1, ...)
  {
    Anweisungen
  }
```

POSIX

Steht für »Portable Operating System Interface for Unix«. Ist ein von der IEEE entwickeltes standardisiertes Interface, das die Schnittstelle zwischen Applikationen und dem Betriebssystem Unix darstellt. Somit sollte sich alle Unix-Derivate an en IEEE1003.1 von 1990 und IEEE1003.2 von 1992 halten. Wurde übernommen als ISO/IEC 9945. Ihre Wichtigkeit ergibt sich aus der Voraussetzung für die Ausschreibung von militärischen und öffentlichen IT-Projekten in den USA..

Einige Standardkomponenten:

P1003.1	Betriebssystemkern und C-Bibliotheken
P1003.2	Shell und Kommandos
P1003.3	POSIX Test Suite
P1003.4	Realzeiterweiterungen
P1003.5	Sprachanbindung an ADA
P1003.6	Systemsicherheit
P1003.x	Systemadministration

Anweisungsende

Das Semikolon kennzeichnet das Anweisungsende. Pro Zeile sind mehrere Anweisungen möglich. Jede Anweisung wird mit einem Semikolon abgeschlossen (auch bei nur einem Befehl pro Zeile).
C ist formatfrei (verlangt zum Beispiel keine Einrückungen oder Leerzeilen) und beinhaltet keine Zeilennummern. Einige Compiler verlangen aber bei Zeichenketten, dass das öffnende und das schließende Anführungszeichen in der selben Zeile stehen:

```
printf ("blub");
```

Das Beispiel

```
print
("blub");
```

erzeugt bei den meisten Compilern zumindest eine Warnung. Wenn die Zeile nicht mit einem Semikolon endet, muss sie mit einem \ beendet werden:

```
print\
("blub");
```

Whitespace

Whitespace ist der Sammelbegriff für Leerzeichen, Tabulator, Seitenvorschub und ähnliches.

Block (compound statement)

Ein Block ist ein zusammengehörenden Quelltextabschnitt.
Der Beginn und das Ende eines Blocks wird mit geschweiften Klammern festgelegt:

```
{
  Anweisungen
}
```

Bei der letzten Anweisung in einem Block ist der Semikolon NICHT wie in anderen Sprachen optional.
Blöcke, die nicht an eine Bedingung oder Schleife gebunden sind, werden »bare blocks« genannt.

File-Extension

File-Extensions sind Dateiendungen.
Bei C Files meist ».c«, bei C++ Files meist ».cpp« und bei Header-Files ».h«.

Prozedur

Eine Prozedur ist ein Unterprogramm (ein Block von Anweisungen), die mit einem Namen aufgerufen werden kann.
An eine Prozedur können Parameter übergeben werden. Eine Prozedur die sich selbst aufruft, wird als rekursiv bezeichnet. Eine Prozedur wird auch Funktion und Subroutine genannt.

Hochsprache

Hochsprachen stehen über maschinennahen Programmiersprachen wie Assembler. Duch ihre geringere Hardwarenähe lassen sie sich leichter auf andere Systeme portieren.

Interpreter

Ein Interpreter übersetzt den Quelltext jedesmal neu zur Laufzeit in Maschinensprache. Ein Compiler hingegeb übersetzt den Quelltext nur einmal in Maschinensprache (und legt den Maschinencode als Datei ab),

Ausdruck

Definition Ausdruck nach Monadjemi/Winkler: »Ein Ausdruck ist eine Rechenvorschrift, das heißt eine sinnvolle Aneinanderreihung von Zahlen, Strings oder sonstiges Datenelementen. Sowie Operatoren, die sich stets zu einem einzelnen Wert auflösen läßt.«

Bug

Als Bug wird ein Fehler im Programm bezeichnet. Fehler, die während dem Betrieb des Programmes auftreten, werden »*Laufzeit-Fehler*« genannt. Das Aufspüren und Entfernen von Bugs wird als »*debug*« bezeichnet. Wird eine Software dafür verwendet, wird diese »*debugger*« genannt. Wird das Programm vom debugger nicht nur auf maschinensprache Ebene inspiziert, sondern auch auf Quellcode Ebene, wird dies als »*source level debugging*« bezeichnet. Der Debugger ermöglicht das schrittweise ablaufen von einzelnen Programmzeilen, die laufende Überwachung von Variablen und das Setzen von Breakpoints (an denen das Programm gestoppt wird).

Remark

Kommentare (Remarks) werden mit einem /* begonnen und mit einem */ beendet. Der Kommentar kann mehrere Zeilen lang sein und kann nicht verschachtelt werden (in einem Kommentar kann nicht noch ein Kommentar sein)
C99 erlaubt auch den C++ Remark //:

```
/* das ist fuer nichts */

// das ist fuer nichts
```

Beispiel:

```
#include <stdio.h>

int main()
{
   //Programm ist im Pfad C:\programme\
   printf("Hallo Welt\n");
   return 0;
}
```

☠ aufgrund des letzten Slash wird print NICHT ausgeführt

Ablaufsteuerung (Kontrollstrukturen)

Auch »flow control« und »control structures« genannt.
Der lineare Programmfluß kann verändert werden mit:

- Loops (Schleifen: eine Gruppe von Anweisungen wird mehrmals ausgeführt)
- bedingte Befehlsausführung (if-Abfragen)
- Aufrufen von Funktionen (Unterprogramme)

Kontrollstrukturen werden optisch durch Struktogramme und Flussdiagramme dargestellt.

I/O (E/A)

Steht für »Input/Output«. Also für E/A (Eingabe/Ausgabe).

stdin (Standard-Input)

Die Standardeingabe (stdin) ist für gewöhnlich die Tastatur.

stdout (Standard-Out)

Die Standardausgabe (stdout) geht für gewöhnlich an den Bildschirm.

stderr (Standard-Error)

Die Ausgabe der Fehlermeldungen (stderr) geht für gewöhnlich an den Bildschirm. Wird auch Fehlerkanal genannt.

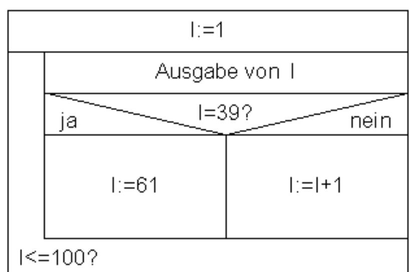
String

Ein String ist eine Zeichenkette. Zeichenketten stehen in Anführungszeichen und beinhalten beliebige Zeichen wie Ziffern, Buchstaben, Sonder- und Steuerzeichen.

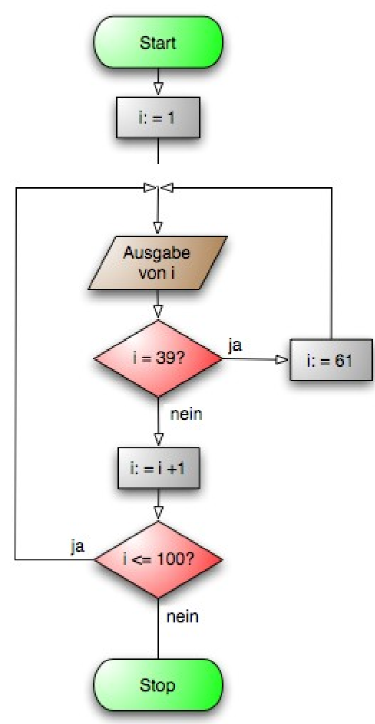
Struktogramm und Flussdiagramm

Sie dienen zur Visualisierung von Kontrollstrukturen.
Als Beispiel: Die Ausgabe der Zahlen von 1 bis 100, jedoch ohne den Bereich 40 bis 60.

Struktogramm:



Flussdiagramm:



Einrückung-Stile (Indent Styles)

K&R-Stil (Kernel Style; 1TBS)

Benannt nach Kernighan and Ritchie. Auch »kernel style« (dank Unix-Kernel-Quelltext) und »One True Brace Style« (1TBS) genannt.
In C besteht die Einrückung meist aus 8 Leerzeichen (oder einem Tab dieser Länge) und in C++/Java meist aus 4 Leerzeichen.

```
if (<cond>) {
        <body>
}
```

Whitesmith-Stil

Benannt nach einem C-Compiler namens Whitesmith.
Die Einrückung besteht meist aus 8 Leerzeichen (oder einem Tab dieser Länge) und gelegentlich aus 4 Leerzeichen.

```
if (<cond>)
        {
        <body>
        }
```

Allman-Stil (BSD Style)

Benannt nach einem Berkeley Entwickler namens Eric Allman. Deshalb auch »BSD Style« genannt.
In Pascal und Algol besteht die Einrückung meist aus 8 Leerzeichen (oder einem Tab dieser Länge) und in C++/Java meist aus 4 (selten 3) Leerzeichen.

```
if (<cond>)
{
        <body>
}
```

GNU EMACS-Stil

Benannt nach GNU EMACS (Free Software Foundation).
Indents are always four spaces per level, with `{` and `}` halfway between the outer and inner indent levels.

```
if (<cond>)
  {
    <body>
  }
```

Zahlensysteme

Zahlensysteme werden auch Positions- und Stellenwertsystem genannt. Stellenwertsysteme, weil in diesen Systemen jedem Zahlenwert außerdem ein Stellenwert zugeordnet ist. Bei der Dezimalzahl 3752 gibt z.B. die 5 durch ihre Stellung an, daß es sich um 5 Zehner handelt.

dezimale Zahlensystem

Das dezimale Zahlensystem verwendet als Basis (base) 10.
Somit gibt es 10 Ziffern (Nennwerte): 0,1,2,3,4,5,6,7,8 und 9.
Die Babylonier verwendeten als Basis 60. Die Stunde hat 60 Minuten und eine Minute hat 60 Sekunden.

$$4536.472 = (4*10^{3}) + (5*10^{2}) + (3*10^{1}) + (6*10^{0}) + (4*10^{-1}) + (7*10^{-2}) + (2*10^{-3})$$

$$4536.472 = 4*1000 + 5*100 + 3*10 + 6*1 + 4*0.1 + 7*0.01 + 2*0.001$$

duale Zahlensystem

Computer verwenden als Basis 2, also das binäre Zahlensystem (auch Dualsystem genannt).
Somit gibt es nur 2 Ziffern (Nennwerte): 0 und 1.

$$100101 = (1*2^{5}) + (0*2^{4}) + (0*2^{3}) + (1*2^{2}) + (0*2^{1}) + (1*2^{0})$$

100101 ist im dezimalen System 37.
Eine binäre Ziffer wird Bit genannt.

4 bit	=	1 nibble	24 = 16
2 nibble	=	1 byte	28 = 256
2 byte	=	1 word	216 = 65536
2 word	=	1 double word (dword)	232 = 4 294 967 296
2 dword	=	1 quad word (qword)	264 = 18 446 744 073 709 551 616

8 Bits bezeichnet man als 1 Byte. Byte kürzt man mit einem großen B ab, und Bit mit einem kleinen b:
Beispiel:
3 B = 24 b
5 bps = 5 Bit pro Sekunde

1 Kilobyte (kB) sind 1024 Bytes. Manchmal sind aber nur 1000 Bytes gemeint.
1 MB sind 1 048 576 Bytes (1024 * 1024) oder 1 000 000 Bytes (1000 * 1000) oder 1 024 000 Bytes (1024 * 1000).

Byte Order

Wird auch »byte sex« genannt.
Die Bezeichnungen »little endian« und »big endian« stammen aus "Gulivers Reisen" von Jonathan Swift.
Bei den Lilliputanern gibt es 2 Gruppen: die einen (Bigendians -> Großender) schlagen ihr Ei am größeren Ende auf, die andern beim Kleineren (Litte Endians -> Kleinender).

big endian (logical order)

Auch »network order« genannt. Verwendung bei: Motorola, IBM 370 family, the PDP-10, RISC
Beispiel bei einem double word (1-2-3-4):

```
00000000 00000111 00000100 00000001
```

little endian

Ist die inverse Darstellung von »Big Endian«
Verwendung bei: x86, DEC Alphas
Beispiel bei einem double word (4-3-2-1):

```
00000001 00000100 00000111 00000000
```

middle endian
auch »PDP-Endian« genannt.
systems save the most significant word first, with each word having the least significant byte first.
Beispiel: 2-1-4-3
Non-US hackers use this term to describe the American mm/dd/yy style of writing dates (Europeans write little-endian dd/mm/yy, and Japanese use big-endian yy/mm/dd for Western dates).
NUXI Problem: The string `UNIX' might look like `NUXI' on a machine with a different 'byte sex'

Binäre Kodierung

Binäre Kodierung von Integer-Zahlen

Man unterscheidet bei Interger-Zahlen zwischen »signed« und »unsigned« .
Bei einer »unsigned Kodierung« sind nur positive Zahlen möglich. Bei einer »signed Kodierung« sind positive und negative Zahlen möglich.

Es gibt 2 Möglichkeiten eine negative Zahl binär zu kodieren:
1. Die Verwendung eines Vorzeichens das mit einem einzelnen Bit dargestellt wird: 0: positiv, 1: negativ.
Diese Kodierung wird bei Gleitkommazahlen verwenden. Für negative ganze Zahlen wird eine andere Darstellung verwendet. Diese Darstellung ergibt sich aus der Absicht, die Arithmetik mit binären Zahlen so einfach wie möglich zu gestalten und im wesentlichen alle Operationen auf die Addition zurückzuführen.

big endian

Verwendung bei: Motorola, IBM 370 family, PDP-10, RISC, MIPS
Das erste Bit (sprich ganz rechts) wird »least significant bit« (LSB) und »low bit« genannt.
Bei einem double word werden die ersten 8 Bit (0 bis 7) »**low byte**« und die ersten 16 Bit (0 bis 15) »**low word**« genannt.
Die letzten 8 Bit (24 bis 31) werden »**high byte**« und die letzten 16 Bit (16 bis 31) »**high word**« genannt.
Das letzte Bit (31) wird »most significant bit« genannt (MSB).

LSB (least significant bit; niedrigstwertiges Bit) ⇨ das erste Bit von rechts
MSB (most significant bit; höchstwertiges Bit) ⇨ das letzte Bit von rechts

little endian

Verwendung bei: x86, DEC Alphas
Bei little endian wird die Reihenfolge der Bytes vertauscht:
Aus 4 3 2 1 wird 1 2 3 4.
Die einzelnen Bits in einem Byte bleiben aber unberührt.

Aus big endian:
```
31 30 29 28 27 26 25 24   23 22 21 20 19 18 17 16   15 14 13 12 11 10 09 08   07 05 04 03 02 01 00
```
wird little endian:
```
07 05 04 03 02 01 00   15 14 13 12 11 10 09 08   23 22 21 20 19 18 17 16   31 30 29 28 27 26 25 24
```

Binäre Kodierung von Fließkommazahlen

Binäre Kodierung von Fließkommazahlen nach ANSI 754

Bei 32 Bit Länge:
    1 Bit für das Vorzeichen (Bit 31), 8 Bits für den Exponenten (Bit 23 bis 30), 23 Bits für Mantisse (Bit 0 bis 22)

Bei 64 Bit Länge:
    1 Bit für das Vorzeichen (Bit 63), 11 Bits für den Exponenten (Bit 52 bis 62), 52 Bits für Mantisse (Bit 0 bis 51)

Das Vorzeichenbit hat für negative Zahlen den Wert 1, für positive Zahlen den Wert 0.
Die Mantisse ist normalisiert (das heisst die erste Ziffer vor dem Komma ist immer 1).
Somit gewinnt man 1 Bit (7 statt 6 Dezimalstellen).

Zahl = Vorzeichen * Mantisse * 2 Exponent

Die Basis ist stets 2. Der Exponent wird mit einer Verschiebung (Bias) gespeichert. Bei 32 Bit 127 und bei 64 Bit 1024.
Beispiel:

```
  1    10000000   10000000000000000000000
```

`10000000 = 128` Verschiebung: 128-127= 1

```
10000000000000000000000 = 4194304
```

-1 * 4194304 * 2^1

Eine Gleitkommazahl hat somit die Form

(-1)^ S * M * 2^E

S = Vorzeichen   0 für positive, 1 für negative Zahlen
M = Mantisse (Wert zwischen 1 und 2)
E = Exponent (ganzzahlig)

Bei einer 32 Bit Fließkommazahl belegt die Mantisse die Bits 0-22. Die Bits 23-30 belegt der Exponent und Bit 31 ist das Vorzeichen. Das Mantissenbit unmittelbar hinter dem Komma wird M-Bit genannt. Das Bit vor dem Komma, J-Bit genannt, wird nicht gespeichert, sondern implizit als gesetzt betrachtet. Nur wenn alle Bits der Mantisse und des Exponenten gelöscht sind, wird es als gelöscht betrachtet und die Zahl ist Null. Wegen des Vorzeichenbits kann man zwischen positiver und negativer Null unterscheiden.

Der Exponent belegt 8 Bits und wird in der Form E + 127 abgespeichert.
Der Maximale Exponent ist somit 128 (255 - 127). Dieser Wert des Exponenten wird aber nicht zur Darstellung von Fließkommazahlen verwendet.Die Werte des Exponenten liegen also zwischen -127 und +127
Beträgt der Exponent 128 (alle Bits des Exponenten gesetzt), so ist die Zahl »not a number« (NaN).
Ist die Mantisse gleich 1 (alle Bits der Mantisse gelöscht) und sind alle Bits des Exponenten gesetzt, ist die Zahl unedlich (Infinity). Ist der Wert der Mantisse gleich 1.5 (höchstes Bit der Mantisse gesetzt, alle anderen Bits der Mantisse 0) und E = 128, so gilt die Zahl als undefiniert. Wenn auch das Vorzeichenbit gesetzt ist, spricht man von der Real Indefinite. Diese spezielle NaN wird bei einer »Invalid Operation Exception« übergeben.

Ähnlich ist es bei 64 Bit Fließkommazahlen. Hier belegt die Mantisse die Bits 0-51, Bits 52-62 belegt der Exponent und Bit 63 ist das Vorzeichenbit. Der Exponent wird als E + 1024 abgespeichert. Er kann Werte zwischen -1023 und +1023 annehmen. Bei E = 1024 liegt wieder eine NaN vor.

Ansi 754 64 Bit

S	M	E	
0	1.0 < M < 2.0	1024	positive NaN
0	1.0	1024	positive Unendlichkeit
0	1.0 < M < 2.0	-1023 <= E <= 1023	positive normale Zahl
0	0.0	-1023	positive Null
1	0.0	-1023	negative Null
1	1.0 < M < 2.0	-1023 <= E <= 1023	negative normale Zahl
1	1.0	1024	negative Unendlichkeit
1	1.0 < M < 2.0	1024	negative NaN
1	1.5	1024	Real Indefinite

Ansi 754 32 Bit

```
S    M                E

0    1.0  < M < 2.0     128                     positive NaN
0    1.0                128                     positive Unendlichkeit
0    1.0  < M < 2.0     -127 <= E <= 127        positive normale Zahl
0    0.0                -127                    positive Null
1    0.0                -127                    negative Null
1    1.0  < M < 2.0     -127 <= E <= 127        negative normale Zahl
1    1.0                128                     negative Unendlichkeit
1    1.0  < M < 2.0     128                     negative NaN
1    1.5                128                     Real Indefinite
```

Beispiele in IEEE 743 - 32 Bit:

0 00000000 00000000000000000000000 = 0

1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinity

1 11111111 00000000000000000000000 = -Infinity

0 11111111 00000100000000000000000 = NaN

1 11111111 00100010001001010101010 = NaN

0 10000000 00000000000000000000000 = +1 * 2(128-127) * 1.0 = 2**

0 10000001 10100000000000000000000 = +1 * 2(129-127) * 1.101 = 6.5**

1 10000001 10100000000000000000000 = -1 * 2(129-127) * 1.101 = -6.5**

0 00000001 00000000000000000000000 = +1 * 2(1-127) * 1.0 = 2**(-126)**

0 00000000 10000000000000000000000 = +1 * 2(-126) * 0.1 = 2**(-127)**

0 00000000 00000000000000000000001 = +1 * 2(-126) * 0.00000000000000000000001 = 2**(-149) (Smallest positive value)**

Binäre Kodierung von Fließkommazahlen als Tempreal:

Bei den **80 Bit Tempreals** belegt die Mantisse die Bits 0 - 63.
Im Gegensatz zu den 32 Bit und 64 Bit Fließkommazahlen wird bei den Tempreals auch das J-Bit gespeichrt (Bit 63). Der Exponent belegt die Bits 64 - 78 und Bit 79 ist das Vorzeichenbit. Der Exponent wird als E + 16383 gespeichert. Er kann Werte zwischen -16383 und +16383 annehmen. Ist der Exponent gleich 16384, so liegt wieder eine NaN vor.

Bei normalen Zahlen liegt der Wert der Mantisse zwischen 1.0 und 2.0 (J-Bit gesetzt). Da das J-Bit bei Tempreals explizit vorhanden ist, und also auch gelöscht sein kann, kann die Mantisse auch kleiner als 1.0 sein.
Wenn die Mantisse kleiner als 1.0 ist, so ist die Zahl nicht normalisiert. Hat der Exponent den kleinst möglichen Wert (-16383), so liegt eine sogenannte denormalisierte zahl vor. Wenn die Mantisse 0.0 ist (alle Bits der Mantisse gelöscht), so ist die Zahl Null. Wegen des Vorzeichenbits kann man dabei noch zwischen positiver und negativer Null unterscheiden. Hat der Exponent dagegen den größten Wert (16384), so liegt eine sogenannte PseudoNaN vor.
Den übrigen Werten des Exponenten entsprechen die unnormalen Zahlen.

```
Tempreal80 (80 Bit)
===================

S              M                       E
======         =======                 ======

0              1.0  <    M    < 2.0    16384                      positive NaN
0              1.0            16384                               positive Unendlichkeit
0              1.0  <=   M    < 2.0    -16383 <=   E    <= 16383  positive normale Zahl
0              0.0  <    M    < 1.0    -16383                     positive denormale Zahl
0              0.0            -16383                              positive Null
0              0.0  <=   M    < 1.0    -16383 <    E    <= 16383  positive unnormale Zahl
0              0.0  <=   M    < 1.0    16384                      positive pseudo NaN
1              0.0  <=   M    < 1.0    16384                      negative pseudo NaN
1              0.0  <=   M    < 1.0    -16383 <    E    <= 16383  negative unnormale Zahl
1              0.0            -16383                              negative Null
1              0.0  <    M    < 1.0    -16383                     negative denormale Zahl
1              1.0  <=   M    < 2.0    -16383 <=   E    <= 16383  negative normale Zahl
1              1.0            16384                               negative Unendlichkeit
1              1.0  <    M    < 2.0    16384                      negative NaN
1              1.5            16384                               Real Indefinite
```

Escape Sequenzen

Escape Sequenzen sind nicht darstellbare Steuerzeichen zur Steuerung von Bildschirmausgaben.

Aus dem ASCII-Zeichensatz:

		ASCII-Wert	Umschreibung mit dem Escape-Zeichen	hex	oct
bell (Signalton)	bel	7	\a	7	7
backspace	bs	8	\b	8	8
horizontaler Tab	ht	9	\t	9	11
new line	nl	10	\n	A	12
vertikaler Tab	vt	11	\v	B	13
form feed (neue Seite)	ff	12	\f	C	14
carriage return	cr	13	\r	D	15

Sonstige:

String-Endmarkierung(NULL)	\0
Hexadezimaler Wert HH	\xHH
Oktaler Wert OO	\oOO

Da diese Zeichen zB bei printf() verwendet werden, müssen sie ebenfalls escaped werden:

\	fs	28	\\	1c	34
"		34	\"	22	42
'		39	\'	27	47
?		63	\?	3f	77

Beispiel1:

```
#include <stdio.h>

void main()
  {
  printf ("sie sagte: \"max, du bist cool!\""); // Funktion printf() wurde in stdio.h deklariert
  }
```

Beispiel2:

```
#include <stdio.h>

void main()
{
  printf ("sie sagte: \"martin, \
  du bist cool!!\""); // Wenn die Zeile nicht mit einem Semikolon endet, muss sie mit einem \ beendet werden
  getchar(); // wartet auf das Drücken der enter Taste
}
```

Beispiel3:

```
#include <stdio.h>

int main()
  {
    printf("neue zeile\n");
    printf("neue zeile hexadezimal : \xa \xa");
    printf("neue zeile octal   : \012 \012");
    printf("neue zeile dezimal     : 10 %c",10);
    printf("blub\n");
    return 0; // 0 wird zurückgegeben, wenn es keine Fehler oder Probleme gab
  }
```

⚠ In DOS/Windows wird für eine neue Zeile ein 0D 0A gebraucht.In Unix/Linux reicht ein 0A.

Variablen & Datentypen

Begriff »Variable«

Eine Variable ist eine vom Programmierer festgelegte Bezeichnung eines bestimmten Speicherplatzes. Der Inhalt des Speicherplatzes kann sich während des Programmablaufes ändern (ansonsten wäre es eine Konstante).
Über die Bezeichnung der Variable (Variablename) kann auf den Speicherinhalt zugegriffen werden.
Jede Variable muß von einem bestimmten Datentyp sein. Beim Definieren einer Variable reserviert der Compiler den Speicherplatz, den er für den Typ der Variable benötigt.
Anders gesagt: Eine Variable ist ein benannter Speicherabschnitt. Der Inhalt des Speicherabschnitts ist der Wert der Variable.

Varibلename

Für die Bezeichnung einer Variable und Funktion gelten diese Regeln:

1. das erste Zeichen ein Buchstabe oder ein Unterstrich sein.
2. gefolgt von Buchstaben, Ziffern und Unterstrichen
3. es wird unterschieden zwischen upper- and lowercase

Die Verwendung von Schlüsselwörtern als Variablename ist nicht möglich. Der Ansi-Standard verlangt, das ein C-Compiler bei intern verwendeten Namen die ersten 31 Zeichen berücksichtigt und bei externen (die in anderen C-Quelltexten definiert wurden sind) mindestens die ersten 6.
Da C »case sensitive« ist, unterscheider der Compiler zB. zwischen main() und MAIN() und betrachtet sie als zwei unterschiedliche Funktionen.
Manchmal werden den Variablen Präfixe und Suffixe angehängt:

foo_ptr	_ptr steht für pointer
foo_p	_p steht für pointer
foo_file	Variable vom Typ *FILE
foo_n	_n steht für nummer

Beispiele für gültive Variablenamen:
Summe
nimm_mich
_kuh

Beispiele für ungültive Variablenamen:

Summe$	Dollarzeichen ist nicht erlaubt
3mal	Variablename darf nicht mit einer Ziffer beginnen
int	int is a reserved word
biber%	Prozentzeichen nicht erlaubt
himmel&hoelle	Kaufmännisches Und nicht erlaubt

Deklarierung einer Variable

Die Deklarierung einer Variable besteht aus 2 bis 3 Teilen:

1. **Angabe des Datentyps** (kann durch Attribute genauer bestimmt werden)
   die optionalen Attribute steuern die Behandlung des Vorzeichens, die Speicherklasse (die die Lebensdauer und Sichtbarkeit bestimmt) und bei ganzzahligen Variablen die Größe
2. **Name der Variable**
3. **Initialisierungswert** (optional); er legt den Inhalt beim Start des Programm fest

Mögliche Attribute:

- Vorzeichen: signed, unsigned
- Speicherklasse: auto, extern, static, register
- Compiler-Steuerung: const, volatile
- Größe: short, long

Mögliche Typen:
int
char
float
double
char*
int*
float*
double*

Beispiel einer Definition:

```
#include <stdio.h>
void main()
  {
    int crap,hunger,blitz;
  }
```

Die Deklarierung muß gleich am Beginn von Funktions- und Anweisungsblöcken stehen (gleich nach der geschweiften Klammer {) und somit noch vor der ersten Anweisung im jeweiligen Block.

Unterschied zwischen Deklaration und Definition

Erst wenn bei einer Deklaration eine Initialisierung vorgenommen wird, handelt es sich um eine Definition.

Laut Peter Monadjemi und Eckart Winkler: »Bei der Deklaration wird der Name und der Typ eines Objekts eingeführt, aber noch ein Arbeitsspeicher zugewiesen. Eine Definition ist damit eine Deklaration, nur dass zusätzlich auch Speicherplatz bereitgestellt wird.«

Initialsierung einer Variable

Die Initialsierung ist eine Variabledefinition, bei der gleich ein Anfangswert zugewiesen wird. Dazu wird hinter dem Variablename ein Gleichheitszeichen (Zuweisungsoperator) angefügt, gefolgt vom Variablewert.
Beispiel:

```
#include <stdio.h>

void main()
{
  int crap=12;
  int fuck=7;
  char foo[]="blubber";    // Länge des Array wird automatisch ermittelt
                           // Stringlänge 7 ist somit gleich Arraygröße

  char foo2[30]="erdbeerkuchen";

                           //      char foo3[3]="erdbeerkuchen";
                           // Stringlänge überschreitet die maximale Arraygröße von 3 und sollte
                           // vom Compiler bestraft werden mit:
                           //  warning: initializer-string for array of chars is too long
                           // oder
                           //  Too many initializers in function 'main'

  printf ("wert1: %d wert2: %d wert3: %s wert4: %s \n",crap,fuck,foo,foo2);
}
```

Nummerische Variablen im Vergleich zu Stringvariablen

Nummerische Variable

Die Zuweisung eines Wertes erfolgt mit dem Zuweisungsoperator = (Gleichheitszeichen)

Beispiel:

```
#include <stdio.h>
void main()
 {
   int crap;
   crap=18;
 }
```

Einer Stringvariable kann auf diese Weise leider kein Wert zugewiesen werden:

```
#include <stdio.h>
void main()
  {
    int crap;
    char crap2[10];
    crap=18;
    crap2="blub";    // erzeugt compiler error
  }
```

Stringvariable
Der Datentyp char (character) kann nur ein einziges Zeichen aufnehmen. Trotzdem können Variablen zur Speicherung von Zeichenketten mit char deklariert werden: `char crap2[10];`
Dies entspricht einer Aneinanderreihung mehrerer Variablen des Typs char. Solch eine Aneinanderreihung eines einzelnen bestimmten Datentyps wird Array genannt.
Die einzelne Variable in der Reihe wird Feldelement bezeichnet. Die Anzahl der Feldelemente ergibt die Arraygröße. Die Zählung der Elemente beginnt bei Null. Das nullte Feldelement ist somit das erste wirkliche Element des Arrays.
Bei der Deklarierung muß die maximale Stringlänge angegeben werden. Im obigen Beispiel darf der String namens crap2 maximal 10 Zeichen beinhalten (von Feldpositon 0 bis 9).
Auf ein einzelnes Feldelement wird über den Arraynamen und die jeweilige Feldposition zugegriffen. In C sind assoziative Arrays nicht möglich (Feld-Index ist keine Zahl, sondern ein String).
Einer StringVariable kann nur durch die Funktion strcpy (string.h) ein Wert zugewiesen werden.

Eine Zuweisung wie bei einernumerischen Variable ist nicht möglich: `crap2="blub"` ☠

```
#include <stdio.h>
#include <string.h>
void main()
{
char crap[6];
strcpy(crap,"blub");// der Stringvariable crap wird der Wert blub zugewiesen
                      // es müssen zwingend doppelte Anführungszeichen (") sein (einfache sind nicht erlaubt: ')
printf("crap: %s\n",crap);
}
```

Der Compiler prüft nicht, ob die zugewiesene Stringkette in die Variable paßt (zugewiesener Stringwert ist größer als die Variable):

```
strcpy(crap,"123456789"); // 3 Zeichen zuviel; keine Compilerwarnung; führt zu Laufzeitfehlern
```

Beide Funktionsparameter sind Zeiger auf den Beginn einer Zeichenkette. Wie lange die Zeichenketten sind, ist der Funktion egal.

Jedes einzelne Zeichen eines Strings (jedes einzelne Arrayelement) wird über einen Index angesprochen. Die Durchnummerierung beginnt bei Null:

```
#include <stdio.h>
#include <string.h>
void main()
{
char crap[9];
char foo;
strcpy(crap,"blub");
foo=crap[2];                              // 3. Zeichen ist das Zeichen u
printf("crap: %s\nfoo: %c\n",crap,foo);
crap[2]='a';                           // dem 3. Feldelement (Indexwert 2) wird das Zeichen a zugewiesen
printf("crap: %s\n",crap);              // crap hat nun den wert blab
}
```

Einem Feldelement wird ein einzelnes Zeichen zugeordnet: `crap[2]='a';`
Es muß zwingend ein einfaches Anführungszeichen verwendet werden (doppelte Anführungszeichen erzeugen einen Compilerfehler).

Wird bei einer Variable-Deklaration eine Initialisierung (Wertzuweisung) vorgenommen wird, spricht man von einer Variable-Definition. Bei solch einer Variable-Definition kann der Compiler die maximale Feldgröße automatisch ermitteln:

```
char crap[]="blubber";  // maximal 7 Feldelemente (von 0 bis 6)
```

Beispiel:

```
#include <stdio.h>
void main()
{
char foo='A'; // Definierung der Variable foo vom Datentyp char. Der Variable wird der Wert A zugewiesen.
              // Entspricht der Zuweisung über dem ASCII-Wert von A:   char foo=65;

char crap[]="blubber";  // Definierung der Variable crap vom Datentyp char.
                        // Durch die eckigen Klammern wird ein Array (Aneinanderreihung
                        // von char-Elementen) definiert.
                        // Die maximale Feldanzahl wird automatisch ermittelt: sie wird durch die
                        // Länge der zugewiesenen Zeichenkette blubber bestimmt (also 7)

foo++;        // Erhöhung von 65 um 1. Der ASCII-Wert 66 ist der Buchstabe B

printf("crap: %s\nfoo: %c\n",crap,foo);

}
```

Lokale und globale Variablen

Lokale Variablen werden innerhalb einer Funktion deklariert. Sie leben nur innerhalb der jeweiligen Funktion und sind somit auch nur in dieser sichtbar. Die meisten Compiler legen diese Variablen nicht im Datenbereich ab, sondern am Stack. Nach Beenden der Funktion wird dieser Platz wieder freigegeben.

Globale Variablen werden außerhalb von Funktionen deklariert und sind programmglobal (alle Funktionen im Programm können diese Variablen sehen und verwenden). Der Compiler legt diese Variablen im Datenbereich ab (somit wird der dafür benötigte Speicher erst bei Programmende freigegeben).

Ungarische Notation (Hungarian Notation)

Die »ungarische Notation« wurde bei Microsoft bei der Entwicklung von Windows eingeführt und deshalb auch »Microsoft Notation« genannt. Es existieren mehrere, sich teilweise widersprechende, Beschreibungen (die gleichen Präfixe zur Bezeichnung verschiedener Datentypen).

c	char
by	BYTE (unsigned char)
uch	UCHAR (unsigned char)
s	short integer
n	short integer (steht im Widerspruch zu s)
us	unsigned short int
i	integer
ui	unsigned int
w	WORD (unsigned integer)
b	BOOL (unsigned short)
l	long
ul	unsigned long int
dw	DWORD (unsigned long int)
x	short int (für die x-Koordinate von Grafiken)
y	short int (für die y-Koordinate von Grafiken)
h	HANDLE (WORD)
fn	Funktion (oft bei Pointern auf Funktionen)
s	Zeichenketten (nicht notwendigerweise NULL terminiert)
sz	Zeichenketten (NULL terminiert)
p	Pointer
lp	long (oder far) pointer
np	short (oder near) pointer

Datentypen

Begriff »Datentyp«

C ist eine streng typisierte Sprache (auch »*strong typing*« und »*strongly typed*« genannt): Alle Variablen und Funktionen haben einen genau vom Programmierer festgelegten Datentyp.
Sprich: Die Variable muss vor ihrer Verwendung definiert werden.
Ist das Gegenteil von »loose typing« (also keine expliziete Angabe des Datentyps).

Ein Datentyp legt
1. den Aufbau der Daten
2. den gültigen Wertebereich
3. die möglichen Operationen fest

Grund-Datentypen

1.	char	Character und ein 8 Bit Integer
2.	int	ganzzahliger Wert (Integer)
3.	float	Gleitkommazahl mit einfacher Genauigkeit.
4.	double	Gleitkommazahl mit doppelter Genauigkeit.

Qualifier

Mit Ausnahme des Typs void kann die Größe der elementaren Datentype durch vier Qualifier näher bestimmt werden:Datentyps:
1. short (kann nur mit int verwendet werden; »kürzer« im Sinne von kleiner als ein normaler int)
2. long (kann nur mit int und double verwendet werden)
3. unsigned (ohne Vorzeichen)
4. signed (mit Vorzeichen)

Daraus ergeben sich folgende Datentypen:

Typ	**Länge**	**Wertebereich**	
char	1 Byte	je nach Compiler von -128 bis 127 oder von 0 bis 255	
unsigned char	1 Byte	0 bis 255	
signed char	1 Byte	-128 bis 127	
int (und signed int)	2 Byte	auf 16 Bit-Rechnern: -32 768 bis 32 767	%d, %i
	4 Byte	auf 32 Bit-Rechnern: -2 147 483 648 bis 2 147 483 647	
short int (signed short int und short)	2 Byte	-32 768 bis 32 767	
long int (und long)	4 Byte	-2 147 483 648 bis 2 147 483 647	%ld, %li
unsigned int	2 Byte	auf 16 Bit-Rechnern: 0 bis 65 535	
	4 Byte	auf 32 Bit-Rechnern: 0 bis 4 294 967 295	
unsigned short int	2 Byte	0 bis 65 535	
unsigned long int	4 Byte	0 bis 4 294 967 295	
float	4 Byte		%f, %g
double	8 Byte		%lf, %lg
long double	12	ab C99	%.llf, llg
long long		-9 223 372 036 854 775 808 bis 9 223 372 036 854 775 808 9 Trillionen; erst ab C99	%lld

Der **Wertebereich** und die mögliche Datentypen sind abhängig vom OS und Compiler. Der Ansi-Standard schreibt für jeden Datentyp nur den minimalen Wertebereich, nicht aber die Größe in Bits vor.

Es gibt keinen expliziten Variablentyp für Zeichenketten. Strings sind Zeiger auf mehrere characters [Datentyp char].

In C gibt es keinen Datentyp zum Repräsentieren boolescher Werte. Er wird deshalb fast immer in jedem C-Programm selbst definiert.
Demgegenüber gibt es in C++ den Datentyp boolean. Die zugehörige Wertemenge ist {true,false}.
Variablen vom Typ boolean benötigen im Prinzip nur 1 Bit als Speicherplatz für ihre beiden Zustände. Dennoch werden sie häufig in einem Byte oder sogar einem Maschinenwort abgelegt. Das erhöht zwar den Speicherplatzbedarf des Programms, verringert aber die Anzahl der notwendigen Maschinenbefehle zum Umgang mit dem Wert der booleschen Variablen.

Nur eines ist sicher: Ein short kann niemals länger sein als ein int, und ein long kann niemals kürzer als ein int sein:
char <= short <= int <= long
Im schlimmsten Falle sind also alle vier Typen gleich lang und trotzdem ANSI-konform. Nur mit dem sizeof Operator erfährt man die Größe eines Datentyps auf dem jeweiligen Rechner. Einige Compiler verfügen in ihrem include-verzeichnis über die Header-Datei limits.h, in der die Größen der Datentypen definiert werden.

Explizite long-Wertzuweisung durch ein nachgestelltes großes L:
foo = 873L;

```
#include <stdio.h>
void main()
  {
    double x;
    x=10.0/4.0;
    printf ("crap1: %lf \n",x);

    x=10/4;
    printf ("crap1: %lf \n",x);
  }
```

Typumwandlung (type conversion)

Eine Typkonvertierung ist eine Umwandlung des Datentyps (data type / data structure).
Operationen werden nur mit Operanden gleichen Typs ausgeführt.
Haben daher die Operanden eines Ausdrucks unterschiedliche Datentypen, werden sie für den betreffenden Ausdruck angeglichen und somit konvertiert.

Beteiligte Variablen des Ausdrucks behalten ihren Datentyp!

Während der Laufzeit wird ein Überlauf nicht gemeldet. Werden zum Beispiel zwei ganze Zahlen so subtrahiert, daß eine negative Zahl herauskommt und ist die Variable, der das Ergebnis zugewiesen wird, vom Typ unsigned, so wird die Bitrepräsentation der negativen Zahl übernommen.
Das ändern eines genaueren Datentyp in einen ungenaueren, zb float nach short, führt zu einem interessanten Ergebnis (abhängig von Compiler und OS):

```
#include <stdio.h>
int main(void)
  {
    short x;
    float z = 25.0;
    x = (short) z;
    printf("x3: %lf \n",x);    // Win Digital Mars: 805306368.000003
                             // Win GCC: 0.000000
                             // Win LCC: 2.64204e-308
                             // Win CC386:  2.64204e-308
    return 0;
}
```

Explizite Typumwandlung

Eine vom Programmierer veranlaßte Umwandlung (wirst casten genannt). Der gewünschte Datentyp (Cast) wird in Klammer vor die Variable gestellt:

Variable = (Datentyp) Variable;

Beispiel:
```
#include <stdio.h>
void main()
  {
    int i = 3, j = 4, k;
    double x = 2.5, y;

    y = (double) i + (double) j;  // 3.0 + 4.0 = 7.0
    y = (double) (i+j)            // 3+4 = 7                -> 7.0  (*)
    y = 3.5;
    i = (int) x + (int) y;        // 2 + 3 = 5              -> 5
    i = (int) (x+y)               // 2.5 + 3.5 = 6.0        -> 6  (*)
  }
```

Dabei behalten die ursprünglichen Variablen ihren Wert und Typ bei. Nur für die Zwischenrechnung wird der neue Wert angenommen. * = explizite Umwandlung unnötig, da sowieso implizit umgewandelt wird

Beispiel:
```
#include <stdio.h>
int main(void)
{
  int x = 5, y = 2;
  float z;

  z = x / y;                  // implizierter cast float
  printf("%f\n", z);          // 2.000000

  z = (float) x / (float) y;  // explizierter cast float
  printf("%f\n", z);          // 2.500000
                              // ein (statt zweit) explizierter cast float hätte genügt,
                              // denn bei mehreren unterschiedlichen Datentypen in einem Ausdruck,
                              //  werden alle Datentypen auf den genauesten (im Ausruck) vorhandenen
                              // Datentyp umgewandelt
  return 0;
}
```

Implizite Typumwandlung

Vom Compiler automatisch durchgeführte Umwandlung.
Operationen werden in dem Typ durchgeführt, der mehr Informationen (also genauer) speichern kann:

short ➤ int ➤ long ➤ float ➤ double

Die Addition einer ganzen Zahl mit einer Fließkommazahl wird also in Fließkommazahlen durchgeführt. Das Ergebnis wird dann in den Typ konvertiert, der auf der linken Seite der Zuweisung (der sogenannte *LValue*) steht.

Beispiel:
```
#include <stdio.h>
void main()
  {
    int i, j;
    double x, y;

    i = 3;
    x = 4.5;
    y = i + x;  //  double = int + double
    j = i + x;  //  int = int + double
  }
```

☠ Die automatische Typenumwandlung funktioniert nicht bei den logischen Operatoren && und ||.

Operatoren

Operatoren bilden mit ihren Operanden Ausdrücke (expressions):

x	=	y
⇪	⇪	⇪
Operand	Operator	Operand

Operatoren nach Anzahl ihrer Operanden

Unäre (unary) Operatoren

Unäre Operatoren werden mit einem Operanden ausgeführt (auch monadische Operatoren genannt).

+	positives Vorzeichen
-	negatives Vorzeichen/Negation
++	Inkrement
--	Dekrement

Binäre (binary) Operatoren

Binäre Operatoren werdem mit zwei Operanden ausgeführt (auch dyadische Operatoren genannt).
Diese Bezeichnung hat nichts mit dem binären Zahlensystem zu tun)

Trinäre Operatoren

Ternäre (ternary) Operationen werden mit drei Operanden ausgeführt (auch triadische Operatoren genannt).

Position des Operators

Infix	Operator steht zwischen den Operanden
Präfix	Operator steht vor dem Operand
Postfix	Operator steht nach dem Operand

Arithmetische Operatoren – , +, *, / und %

Nachdem eine Variable definiert und initialisiert wurde, kann mit ihr arithmetische Operationen durchgeführt werden.
Für arithmetische Operatoren gilt die Punkt-vor-Strich-Regelung: * und / gehen vor + und –

5 + 5 * 5 = 30 (nicht 50!)

50 wären:

(5 + 5) * 5 =50

Arithmetische Operatoren sind binäre Operatoren.

Subtraktionsoperator	–	`x-=5`	`x=x-5`	Subtraktion
Additionsoperator	+	`x+=5`	`x=x+5`	Addition
Multiplikationsoperator	*	`x*=5`	`x=x*5`	Multiplikation
Divisionsoperator	/	`x/=5`	`x=x/5`	Division
Modulusoperator	%	`x%=5`	`x=x%5`	Modulus (modulo)

Rest der bleibt, wenn der linke Operand durch den rechten dividiert wird. sprich: x modulo y

Beispiele:

```
100%2     // 0
100%3     // 1
20%7      // 6
9%6       // 3
```

Inkrementoperator	++	`x++`	Inkrement (Erhöhung) um eins (increment)
Dekrementoperator	--	`x--`	Dekrement (Erniedrigung) um eins (decrement)
		`x=y++`	Präinkrementierung; das Argument wird zuerst erhöht (prae=vor)
		`x=++y`	Postinkrementierung; das Argument wird zuerst ausgewertet und dann erhöht (post=nach)

Beispiele:

Angenommen wird: a=1; b=0;

`b=++a;`	a ist 2 und b ist 2	(Präfix-Inkrement; erhöhe erst a um 1, und weise erst dann, den Wert der Variable b zu)
`b=--a;`	a ist 0 und b ist 0	(Präfix-Dekrement)
`b=a++;`	a ist 2 und b ist 1	(Postfix-Inkrement; weise b den Wert von a zu, und erhöhe erst dann a um 1)
`b=a--;`	a ist 0 und b ist 1	(Postfix-Dekrement)

Vergleichsoperatoren ==, >, <, <= und !=

==	x==y	linker Operand gleich dem rechten
>	x>y	linker Operand größer gleich als der rechte
<	x<y	linker Operand kleiner als der rechte
<=	x<=y	linker Operand kleiner gleich als der rechte
!=	x!=y	linker Operand ungleich dem rechten

Nicht nur Vergleiche, sondern auch jeder Ausdruck alleine besitzt einen logischen Wert:
Jeder Ausdruck, dessen numerischer Wert nicht Null ist, gilt als true.
Jeder Ausdruck, dessen numerischer Wert Null ist, gilt als false.

Beispiele:

```
#include <stdio.h>
void main()
  {
    if (0) { printf("true\n");} else printf("0 \t\t\t-> false\n");
    if (!0) { printf("!0 \t\t\t-> true\n");}
    if (-1+1) { printf("true\n");} else printf("0 \t\t\t-> false\n");
    if (3+2) { printf("3+2 \t\t\t-> true\n");}
    if (3==1+2) { printf("3==1+2 \t\t\t-> true\n");}
    if (4!=1+2) { printf("4!=1+2 \t\t\t-> true\n");}
    if (4>3) { printf("4>3 \t\t\t-> true\n");}
    if (4<6) { printf("4<6 \t\t\t-> true\n");}
    if (6<=6) { printf("6<=6 \t\t\t-> true\n");}
  }
```

☠ Das Prüfen auf Ungleichheit mit <> ist nicht möglich (außer beim Digital Mars C++ Compiler) und wird mit einem Compiler-Error bestraft: `if (4<>6) { printf("4<>6 \t\t\t-> true\n");}`

logische Operatoren !, && und ||

!	logisches NOT	(NICHT)	Negation (invertiert den Wert); aus true wird false und aus false wird true
&&	logisches AND	(UND)	ist true, wenn beide Werte true sind
||	logisches OR	(ODER)	ist true, wenn ein Wert true ist

Beispiele (angenommen wird: x=3; y=7):

```
#include <stdio.h>
void main()
  {

    /*
    3 || 7          3 ist true; 7 ist true;  der Rückgabewert ist true
    x-x || 7>3    links ist false; rechts ist true; der Rückgabewert ist true
    7+3 || x*0    links ist true; rechts ist false; der Rückgabewert ist true
    3>7 || 7<0    links ist false; rechts ist false; der Rückgabewert ist false
    !(3>7) || 7<0  links ist true; rechts ist false;   der Rückgabewert ist true
    !(3>7) && 7<0  links ist true; rechts ist false;   der Rückgabewert ist false
    */

    int x=3,y=7;
    printf ("x=3, y=7 \n\n");
    if (x || y) { printf("x || y \t\t\t-> true\n");}
    if (x-x || y>3) { printf("x-x || y>3  \t\t-> true\n");}
    if (7+3 || x*0) { printf("7+3 || x*0 \t\t-> true\n");}
    if (3>7 || 7<0) { printf("true\n");} else { printf("3>7 || 7<0 \t\t-> false\n"); }
    if (!(3>7) || 7<0) { printf("!(3>7) || 7<0 \t\t-> true\n");}
    if (!(3>7) && 7<0) { printf("true\n");} else { printf("!(3>7) && 7<0 \t\t-> false\n"); }
  }
```

logische Variablen

Logische Variablen werden vom Programmierer als Flag verwendet. Ein Flag kann zwei unterschiedliche Zustände haben:
1. Flag gesetzt und 2. Flag nicht gesetzt.
Somit müssen diese Variablen nur 2 Zustände beschreiben: 1. wahr und 2. falsch
Für true wird die Variable auf 1 gesetzt. Für false auf 0. Vom Datentyp eignen sich zB short (geringster Speicherverbrauch), long und double.
In C99 wird dafür eigens ein neuer Datentyp eingeführt: _Bool
Bool, true und false werden in der Headerdatei stdbool.h definiert.

Zuweisungsoperator =

=

konditionale (bedingte) Operatoren ? :

Siehe if-Abfrage.

(Ausdruck/Bedingung) ? Zuweisung1 : Zuweisung2

⇩ ⇩

Bedingung trift zu Ausdruck ist true	Bedingung trifft nicht zu Ausdruck ist false

Sequenzoperator (Kommaoperator) ,

Mit dem Sequenzoperator werden mehrere Ausdrücke voneinander getrennt.

```
a++;
b=5;
```

ist identisch mit

```
a++,b=5;
```

Und

```
b=a;
a++;
c=5;
d=++c;
```

ist identisch mit

```
b=a,a++,c=5,d=++c;
```

Er wird auch dort verwendet, wo aus syntaktischen Gründen nur ein Ausdruck möglich ist, wie etwa in Schleifen.

Beispiel in C++ (die Reihenfolge der Elemente wird umgedreht):

```
#include<iostream.h>
void main()
  {
    int i;
    int j;
    int nTemp;
    int nArray[20];

    for (i=0; i++; i<11)
      {
        nArray[i]=i;
      }
    for (i=0, j=10; i<j; i++, j-- )
      {
        nTemp=nArray[i];
        nArray[i]=nArray[j];
        nArray[j]=nTemp;
        cout << i << " " << j << "\n";
      }
  }
```

sizeof Operator

Der unäre sizeof Operator gibt die Bytegröße der Variable (also des Datentyps) in Bytes zurück.
Beispiel:

```
#include <stdio.h>
int main(void) {
   printf("char     : %d Byte\n" , sizeof(char));        // zB 1
   printf("int      : %d Bytes\n", sizeof(int));        // zB 4
   printf("long     : %d Bytes\n\n", sizeof(long int));  // zB 4

   printf("float    : %d Bytes\n", sizeof(float));       // zB 4
   printf("double   : %d Bytes\n", sizeof(double));        // zB 8
   printf("66       : %d Bytes\n\n", sizeof(66));          // zB 4

   printf("Hallo    : %d Bytes\n", sizeof("Hallo"));   // zB 6
   printf("A        : %d Bytes\n", sizeof((char)'A')); // zB 1
   printf("A        : %d Bytes\n", sizeof('A'));       // zB 4
   printf("34343434 : %d Bytes\n\n", sizeof(34343434)); // zB 4
   return 0;
}
```

Indirektionsoperator (Verweisoperator) *

Die Variable-Deklaration besteht aus dem

1. Datentyp
2. dem Indirektionsoperator *
3. der Variablebezeichnung:

Datentyp *Zeigervariable

☠ Der Datentyp des Zeigers muß vom selben Datentyp sein, auf den er zeigt.

Der Stern ist der Indirektionsoperator (er kennzeichnet den Datentyp Zeiger) und befindet sich zwischen dem Datentyp und der Variablebezeichnung.
Das Anwenden des Indirektionsoperator auf einen Zeiger wird als "Dereferenzierung der Zeigervariable" bezeichnet.

Adressoperator &

Nach der Deklaration wird der Zeiger mit dem Adressoperator & initialisiert.

zeiger = &variable;

Beispiel:

```
#include <stdio.h>
int main(void)
  {
    int *blub,b;
    b=9;
    blub=&b;
    printf("\ndec address von b: %d ",blub);
    printf("\nhex address von b: %p \n",blub);
    printf("\nwert von b: %d ",*blub); // indirekter Zugriff auf Variable
    printf("\nwert von b: %d \n",b);   // direkter Zugriff auf Variable
    return 0;
  }
```

Den Zugriff auf den Inhalt einer Variablen über den Variablennamen nennt man direkten Zugriff. Und den Zugriff auf den Inhalt einer Variablen über einen Zeiger auf diese Variable nennt man indirekten Zugriff (Indirektion).

Bit-Operatoren & | ^ ~

&	bitweises AND	UND	gibt eine 1 zurück, wenn beide Bits 1 sind
			`x&=y`
			`x=x&y`

|	bitweises OR	ODER	gibt eine 1 zurück, wenn eines der beiden Bits 1 ist
			`x|=y`
			`x=x|y`

^	bitweises XOR		gibt eine 1 zurück, wenn ein Bit, aber nicht beide Bits 1 sind
			`x^=y`
			`x=^y`

~	bitweises NOT	NICHT	invertiert den Bit
			`x~=y`
			`x=x~`

<<			Linksverschiebung um y Stellen (left shift)
			`x<<=y`
			`x=x<<y`

>>			Rechtsverschiebung um y Stellen (right shift).
			Dazu gehört auch das Bit, indem das Vorzeichen gespeichert ist.
			`x>>=y`
			`x=x>>y`

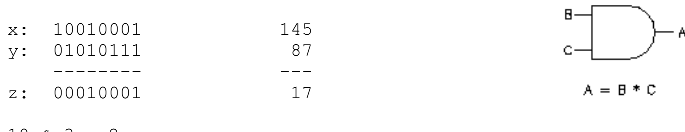
>>>			Rechtsverschiebung um y Stellen;
			`x>>>=y`
			`x=x>>>y`

AND Bitoperator &

Mit &(bitweises UND) werden die einzelnen Bits zweier Ganzzahlen miteinander verglichen.
Steht an einer bestimmten Bitposition in beiden Zahlen eine 1, dann bekommt
das Ergebnis der Operation derselben Stelle eine 1. Ansonsten 0.

z = x & y

Beispiel:




```
10 & 3 = 2
```

Damit der Output true ist, müssen beide Inputs in das AND gate true sein.
AND wird durch einen Punkt repräsentiert: B * C = A
False bedeutet 0 Volt (genannt VSS) und true zwischen 3 und 5 Volt (genannt VCC oder VDD)

OR Bitoperator |

Mit | (bitweises ODER) wird eine 1 gesetzt, wenn mindestens ein Bit eine 1 ist.

```
z = x | y
```

Beispiel:

```
x:  100111010          314
y:  010110100          180
    ---------          ---
z:  110111110          446

10 | 3 = 11
```

D = E + F

Mindestens ein Input muss true sein, damit der Output true ist.
OR wird durch ein plus repräsentiert: E + F = D

XOR Bitoperator ^

Mit ^ (bitweises XODER) wird eine 1 gesetzt, wenn nur eine 1 gesetzt ist.

```
z = x ^ y
```

Beispiel:

```
x:  1011011101         733
y:  0111100111         487
    ----------         ---
    1100111010         826

10 ^ 3 = 9
```

NOT Bitoperator ~

Mit ~ (bitweises NICHT / Negationsoperator / Inverter)
Vergleicht nicht 2 Integer, sondern invertiert die Bits. Aus 1 wird 0 und aus 0 wird 1.

```
z=~x
```

Beispiel:

```
x:  1001011100         604
z:  0110100011         419
```

$G = \overline{H}$

Damit der Output true ist, muss der Input false sein
NOT wird durch einen oberen Strich repräsentiert: $B = \overline{A}$

Bei 32 Bit ist 604 intern:	00000000000000000000001001011100
604 invertiert ist -605:	11111111111111111111110110100011
Bei unsigned:	4294966691

Schiebeoperatoren << >>

left-shift Operator <<

Mit dem << Operator wird das Argument auf der linken Seite des Operators, um die auf der rechten Seite angegebene Anzahl von Bits nach links verschoben. Die rechte Seite wird mit Null-Bits aufgefüllt.

x << 1 entspricht einer Multiplikation mit 2 (binär: 10)
Beispiel: 0110110 << 1 = 110110
Die 1. Null (bzw. die erste Ziffer) wird ans Ende verschoben

x << 2 entspricht einer Multiplikation mit 4 (binär: 100)
Beispiel: 00110110 << 2 = 11011000
Die ersten 2 Nullen (bzw. die ersten 2 Ziffern) werden ans Ende verschoben.

x << 3 entspricht einer Multiplikation mit 8 (binär: 1000)
Beispiel: 000110110 << 3 = 110110000
Die ersten 3 Nullen (bzw. die ersten 3 Ziffern) werden ans Ende verschoben.

Mit Verschiebe-Operatoren kann man nur Multiplikationen und Divisionen mit Potenzen der Zahl 2 durchführen (und dies nur mit Ganzzahlen). Sie sind aber schneller als arithmetische Operatoren.

right-shift Operator >>

Mit dem >> Operator wird das bitweise Nach-rechts-schieben unter Berücksichtigung des Vorzeichens implementiert. Das Vorderzeichen-Bit wird dupliziert und links wieder eingesetzt.

Der zweite Nach-rechts-schiebe-Operator ist >>>.
Hier wird ohne Berücksichtung des Vorzeichen-Bits die linke Seite mit Null-Buits aufgefüllt.

x >>> 1 entspricht einer Division durch 2 (binär: 10)
Beispiel: 110110 >>> 1 = 11011
Die letzte Ziffer links fällt weg (in diesem Fall 0).

x >>> 2 entspricht einer Division durch 4 (binär: 100)
Beispiel: 110110 >>> 2 = 1101
Die letzten 2 Ziffern links fallen weg (in diesem Fall 10).

x >>> 3 entspricht einer Division durch 8 (binär: 1000)
Beispiel: 110110 >>> 3 = 110
Die letzten 3 Ziffern links fallen weg (in diesem Fall 110).

Setzen und löschen eines Bits

»Setzen des Bits« bedeutet, daß das Bit den Wert 1 annimmt.
»Löschen des Bits« bedeutet, daß das Bit den Wert 0 annimmt.

Flag setzen:	1	oder	true
Flag löschen bzw. nicht gesetzt:	0	oder	false

(M)ontag bis (S)onntag:

```
M D M D F S S
0 0 0 0 0 0 0 0
```

Es hat am Donnerstag geregnet. Bei D soll ein Bit gesetzt werden.
Es soll so aussehen:

```
0 0 0 1 0 0 0 0
```

Um dies zu erreichen:

```
  0 0 0 0 0 0 0 0
| 0 0 0 1 0 0 0 0
-----------------
  0 0 0 1 0 0 0 0
```

Statt Donnerstag hat es am Mittwoch und Freitag geregnet.
Es soll so aussehen:

```
0 0 1 0 1 0 0 0
```

Um dies zu erreichen:

```
  0 0 0 1 0 0 0 0
| 0 0 1 0 1 0 0 0
-----------------
  0 0 1 1 1 0 0 0
& 0 0 1 0 1 0 0 0
-----------------
  0 0 1 0 1 0 0 0
```

main()

Jedes C-Programm muß aus mindestens einer Funktion bestehen. Und zwar aus der Funktion »main«. Diese wird als erste Funktion aufgerufen (alle anderen Funktionen werden direkt oder indirekt von main() aus aufgerufen).

```
#include <stdio.h>
int main (void)
  {
      putchar('X');                /* grosses X ausgeben */
      return 0;
  }
```

Mit der include-Anweisung wird der Preprozessor angewiesen, die Header-Datei stdio.h einzufügen.
Das Schlüsselwort int bestimmt den Datentyp des Rückgabewert (von der Funktion main).
Die runden Klammern nach dem Funktionsnamen umschließt die Argumente, die der Funktion übergeben werden.

☠ Im Beispiel wird der Funktion kein Parameter übergeben. Während »Kernighan & Richie« für diesen Fall nur leere Klammern vorsahen, wird bei Ansi-C das Schlüsselwort »*void*« verwendet.

Die geschweiften Klammern markieren den zur Funktion gehörenden Programmblock.
Die Funktion putchar() gibt den Buchstaben X aus. Das abschließende return bewirkt die Rückkehr zur aufrufenden Funktion.
Bei main() ist dies üblicherweise das Betriebssystem.

Assoziativität

Mit Assoziativität wird die Richtung bezeichnet, in der Operatoren gleicher Priotitätsstufe in einem Ausdruck ausgewertet werden.
Linksassoziative Operatoren von links nach rechts, rechtsassoziative Operatoren von rechts nach links.
Bei einigen Operatoren (nichtassoziative Operatoren) ist die Reihenfolge nicht relevant, nicht garantiert oder gar nicht möglich.

Die meisten Operatoren in C sind linksassoziativ:

`foo + bar - tom`	(foo + bar) – tom foo und bar werden addiert. Von der Summe wird tom subtrahiert.
Obiges Beispiel rechtsassoziativen:	foo + (bar – tom) bar mit tom subtrahiert. Zur Differenz wird foo addiert.

Assoziativgesetz

Das Assoziativgesetz (lat. *associare* - vereinigen, verbinden, verknüpfen, vernetzen), auf Deutsch Verknüpfungsgesetz und Verbindungsgesetz, ist eine mathematische Regel:

> Eine (zweistellige) Verknüpfung ist assoziativ, wenn die Reihenfolge der Ausführung keine Rolle spielt.
> Anders gesagt: die Klammerung mehrerer assoziativer Verknüpfungen ist beliebig.

Operatorprioriät (Operator-Rangfolge)

Der Vorrang (precedence) der Operatoren legt fest, welche Operatoren in einem Ausdruck zuerst ausgewertet werden. Operatoren mit höherer Priorität werden zuerst ausgewertet.
Die Auswertungsreihenfolge (also die Verarbeitungsfolge) kann mit Klammern verändert werden (Ausdrücke in Klammern haben Vorrang).

Operator   Assoziativiät

01. Priorität
- Funktionsaufruf   ( )   L ⇨ R
- Array-Element   [ ]   L ⇨ R
- Zeiger auf Strukturelement   ->   L ⇨ R
- Punktoperator   .   L ⇨ R

02. Priorität
- Inkrement nach Auswertung   ++   R ⇨ L
- Dekrement nach Auswertung   --   R ⇨ L
- Inkrement vor Auswertung   ++   R ⇨ L
- Dekrement vor Auswertung   --   R ⇨ L

unär
- Logische Negation   !   R ⇨ L
- Einerkomplement   ~   R ⇨ L
- Unäre Minus (Vorzeichen)   -   R ⇨ L
- Unäres Plus (Vorzeichen)   +   R ⇨ L
- Adressoperator   &   R ⇨ L
- Indirektion   *   R ⇨ L
- Größe in Bytes   sizeof   R ⇨ L
- Typenumwandlung (cast)   (datatype)   R ⇨ L

03. Priorität
- Multiplikation   *   L ⇨ R
- Division   /   L ⇨ R
- Modulo   %   L ⇨ R

04. Priorität
- Addition   +   L ⇨ R
- Subtraktion   -   L ⇨ R

05. Priorität
- Bitweises Linksverschieben   <<   L ⇨ R
- Bitweises Rechtsverschieben   >>   L ⇨ R

06. Priorität
- kleiner als   <   L ⇨ R
- größer als   >   L ⇨ R
- kleiner gleich als   <=   L ⇨ R
- größer gleich als   >=   L ⇨ R

07. Priorität
- Gleichheit (ist gleich)   ==   L ⇨ R
- Ungleichheit (ist nicht gleich)   !=   L ⇨ R

08. Priorität
- Bitweises UND   &   L ⇨ R

09. Priorität
- Bitweises Exklusiv-ODER   ^   L ⇨ R

10. Priorität
- Bitweises ODER | L ⇨ R

11. Priorität
- Logisches UND   &&   L ⇨ R

12. Priorität
- Logisches ODER || L ⇨ R

13. Priorität
- bedingte (ternäre) Operatoren   ? :   R ⇨ L

14. Priorität
- Zuweisung   =   R ⇨ L
- Zusammengesetzte arithmetische Zuweisungen   *= /= %= += -=
- Zusammengesetzte bitweise Zuweisungen   <<= >>= &= ^= |=

15. Priorität
- Komma-Operator   ,   L ⇨ R

In C++ besitzt der scope resolution operator (Bereichszugriffsoperator) die höchste Priorität:
- scope resolution operator   ::   R ⇨ L   unär
- scope resolution operator   ::   R ⇨ L   binär

Basic I/O Funktionen

Für den Beginn die 7 wichtigsten I/O-Funktionen:

1. getchar() Einlesen eines einzelnen Zeichen von der Standardeingabe (Tastatur)
2. putchar() ein einzelnes Zeichen an die Standardausgabe (Bildschirm) schicken
3. gets() ermöglicht das Einlesen von Texteingaben über Tastatur; identisch mit *scanf("%s",foo)*
4. puts() Text-Ausgabe am Bildschirm
5. fputs() wie puts(), jedoch ohne Zeilenumbruch
6. printf() formatierte Text-Ausgabe am Bildschirm (mit Platzhaltern für Variablen)
7. scanf() ermöglicht das Einlesen von Texteingaben über Tastatur und gibt diese formatiert zurück

   atoi() wandelt einen String in einen Integer um

getchar()

Ermöglicht das Einlesen eines einzelnen Zeichens von der Standardeingabe (stdin). Die Standardeingabe ist per default die Tastatur.

Die Eingabe wird durch das Drücken der Eingabetaste (auch Enter- und Returntaste genannt) abgeschlossen. Die Rückgabewert der Funktion ist ein Integer. In C kann ein Zeichen als char oder als int abgelegt werden.

Beispiel:

```
#include <stdio.h>
void main()
  {
    char zeichen;       // Variable namens zeichen als character deklarieren
    zeichen=getchar();
    putchar(zeichen);
  }
```

putchar()

Ermöglicht das Ausgeben eines einzelnen Zeichen an die Standardausgabe (stdout). Das Zeichen kann ein beliebiges ASCII-Zeichen sein. Also Buchstaben, Ziffern und Steuerzeichen.

Beispiel:

```
#include <stdio.h>
void main()
  {
    char ausgabe;
    int ausgabe2;
    putchar('A');      // Zeichen A ausgeben
    putchar('\x41');     // Zeichen A mit Angabe des Hexwertes 41 ausgeben
    putchar('\xa');      // Sonderzeichen NewLine (Zeilenumbruch) mit Angabe des Hexwertes A ausgeben
    putchar('2');    // Zeichen 2 ausgeben
    putchar('\n');   // Sonderzeichen NewLine (Zeilenumbruch) mit Angabe der Escape-Sequenz ausgeben
    putchar('*');   // Zeichen * ausgeben
    putchar('\\'); // Zeichen \ ausgeben
    ausgabe=32;      // der Variable ausgabe den Wert 32 zuweisen
    putchar(ausgabe);   // Leerzeichen (ASCII-Wert 32) ausgeben
    ausgabe2=81;     // der Variable ausgabe2 den Wert 81 zuweisen
    putchar(ausgabe2); // Zeichen Q (ASCII-Wert 32) ausgeben
  }
```

gets()

Liest auf der Tastatur eingegebene Zeichen ein. Beendet wird die Eingabe mit dem Drücken der Enter-Taste.

Der Text wird in einer Stringvariable abgelegt.

☠ Im Unterschied zu scanf() fügt gets() am Schluß einen Zeilenumbruch hinzu.

Beispiel:

```
#include <stdio.h>
void main()
  {
    char blub[100];
    puts("gib text ein: ");
    gets (blub);
    puts("du hast eingegeben: ");
    puts(blub);
  }
```

puts()

Gibt eine Zeichenkette am Bildschirm aus. Nach der Ausgabe wird ein Zeilenumbruch durchgeführt

Beispiel:

```
#include <stdio.h>
void main()
  {
    char blub[100];
    puts("gib text ein: ");
    gets (blub);
    puts("du hast eingegeben: ");
    puts(blub);
  }
```

fputs()

Gibt eine Zeichenkette am Bildschirm aus. Der erste Parameter ist die Zeichenkette, der zweite der Output-Kanal (zB stdout).

☠ Im Unterschied zu puts() wird kein Zeilenumbruch durchgeführt.

Beispiel:

```
#include <stdio.h>
void main()
  {
    char blub[100];
    fputs("erste zeile",stdout);
    puts("\nzweite zeile");
    puts("dritte zeile");
    fputs("kein fehler",stderr);
  }
```

printf()

Ermöglicht die formatierte Printausgabe am Bildschirm. Voraussetzung für die Verwendung ist das Inkludieren der Headerdatei *stdio.h* (Standard Input/Output). Der erste Parameter der Funktion ist der Steuerstring. Dieser String enthält den Text und eventuelle Formatelemente (Platzhalter für Variablen). Die Formatelemente werden beim Programmdurchlauf durch die eigentlichen Werte ersetzt.

Beispiel:

```
#include <stdio.h>
int main()
 {
   printf("\nblubber\n");
   return 0;
 }
```

Der Bereich zwischen den 2 Anführungszeichen ist der Text der ausgegeben wird. Die Escape-Sequenz \n erzeugt einen Zeilenumbruch.
Im Text können Platzhalter für Variablen stehen. Platzhalter bestehen aus dem Prozentzeichen % und einem Formatierungszeichen: c, d, e, f, g, h, i, o, s, u, x.
Platzhalter treten immer paarweise mit ihrem Wert auf. Die Anzahl der Platzhalter muß also identisch mit der Anzahl der Werte sein.

print ("\nwert von x ist %d (hexadezimal).\n", 15);   // wert von x ist 15.

Platzhalter (↑ %d) Wert (↑ 15)

Beispiel:

```
#include <stdio.h>
int main()
 {
   int x=15;
   printf("%d ist hex %x\n",x,x); // 15 ist hex f
   return 0;
 }
```

☠ Anführungszeichen, Tabulatoren, Zeilenumbrüche (\n) und ähnliches müssen escaped werden. Siehe Escape-Sequenzen.

Mogliche printf-Konvertierungszeichen:

%c	einzelnes Zeichen (character)
%d	Dezimalzahl mit Vorzeichen (int)
%i	Dezimalzahl (int) mit Vorzeichen; wie %d
%e	Fließkommazahl mit Vorzeichen in Exponential-Schreibweise
%f	Fließkommazahl mit Vorzeichen (float)
%g	Fließkommazahl, die als %e oder als %f dargestellt wird (je nachdem, was kürzer ist)
%o	Oktalzahl (zur Basis 8) ohne Vorzeichen (int)
%s	String
%u	Dezimalzahl (int) ohne Vorzeichen
%x	Hexadezimalzahl (zur Basis 16) ohne Vorzeichen (int)

Mit printf Bündigkeit erreichen:

```
#include <stdio.h>
void main()
 {
   printf("%10s","blub");   // erzeugt einen String mit der Länge von 10 Zeichen;
                            // der String wird links mit 6 Leerzeichen aufgefüllt
   printf("*\n");
   printf("%-10s","blubber");   // der String wird rechts mit 3 Leerzeichen aufgefüllt
   printf("*\n");
 }
```

Das Beispiel sollte diesen Output liefern:

```
      blub*
blubber   *
```

scanf()

Liest die Tastatureingabe und speichert sie in eine Variable. Der erste Parameter ist ein Steuerstring, das nur aus einem einzigen Steuerelement besteht (Text oder weitere Steuerelemente sind nicht möglich).
Beispiel:

```
#include <stdio.h>
void main()
 {
   char blub[20];
   printf ("\ngib text ein: ");
   scanf("%s",blub);                     // alternativ: scanf("%s",&blub);  beachte unteres Beispiel!
   printf("\ndeine eingabe: %s\n ",blub);
 }
```

Eingabe von Zahlen:

```
#include <stdio.h>
void main()
  {
    double blub;
    printf ("\ngib text ein: ");
    scanf("%d",&blub);         // hier ist die Angabe des Adressoperator zwingend notwendig (siehe oberes Beispiel)
    printf("\ndeine eingabe: %d\n ",blub);
  }
```

Eingabe auf bestimmte Anzahl von Zeichen beschränken:

```
#include <stdio.h>
void main()
  {
    char blub[20];
    printf ("\ngib text ein: ");
    scanf("%5s",blub);       // String auf die ersten 5 Zeichen beschränken
                             // alles ab dem 6. Zeichen der Eingabe wird gelöscht)
    printf("\ndeine eingabe: %s\n ",blub);
  }
```

atoi()

Wandelt einen String in einen Integer. Voraussetzung für die Verwendung ist das Inkludieren der Headerdatei *stdlib.h* (standard library).
Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
void main()
  {
    char eingabe[100];
    int geburtsjahr,alter,jahr;
    jahr=2063;
    puts("Gib Geburtsjahr ein: ");
    gets (eingabe);
    alter=2063-atoi(eingabe);
    printf ("Du bist im Jahr 2063 %d bereits Jahre alt!\n",alter);
  }
```

☠ Mit String-Variablen läßt sich nicht rechnen. Auch dann nicht, wenn der String nur aus Ziffern besteht.
Anders gesagt: mit Stringvariablen lassen sich keine mathematischen Operationen durchführen.
Folgende Zeile erzeugt einen Compiler-Error: `alter=2063-eingabe;`

Konstante

Man unterscheidet 4 Arten von Konstanten:

1. Integer-Konstante
2. Fließkomma-Konstante
3. Zeichenkonstante
4. Zeichenkette-Konstante

Im Gegensatz zu Variablen ist der Wert einer Konstante während der Laufzeit unveränderbar.

Integer-Konstante

Eine Integer-Konstante ist eine ganze Zahl, die positiv, negativ oder null sein kann.
Es gibt 3 Schreibweisen

1. dezimale Schreibweise
   Eine dezimale Integer-Konstante ist ein Kombination aus Ziffern von 0 bis 9
   Die erste Ziffer darf keine Null sein.
   Beispiele; 9, 7, 847, 0

2. oktale Schreibweise
   Eine oktale Konstante besteht aus den oktalen Ziffern 0 bis 7.
   Das erste Zeichen muß eine Null sein (sie steht vor dem eigentlichen Wert).
   Beispiele: 09, 007, 0847, 00

3. hexadezimale Schreibweise
   Eine hexadezimale Konstante besteht aus den hexadezimalen Ziffern von 0 bis F.
   Die ersten 2 Zeichen sind 0 und x (sie stehen vor dem eigentlichen Wert).
   Beispiele: 0x9, 0x7, 0x847, 0x0

Meist ordnet der Compiler einer numerischen , ganzzahligen Konstante den kleinstmöglichen Datentyp (meist integer) zu.
Sobald ihr Wert den möglichen Bereich überschreitet, ändert der Compiler von selbst den Datentyp (zb von integer auf lont int).
Der Programmierer kann dies verhindern und einen bestimmten Datentyp durch das Hinzufügen eines Suffix (L, U und UL) erzwingen (können auch klein geschrieben werden):

L durch das Suffix L wird **long** erzwungen.
Beispiel: 0L

U durch das Suffix U wird **unsigned** erzwungen.
Beispiel: 0x2aU

UL durch das Suffix LU wird **long unsigned** erzwungen.
Beispiel: 007UL

Bei Fließkommakonstanten stehen die Suffixe F und L zur Verfügung.

Fließkomma-Konstante

Eine Fließkoma-Konstante ist eine ganze Zahl mit Nachkommastellen.
Es gibt 2 Schreibweisen:

1. dezimale Ziffern mit einem Punkt als Kommazeichen
   Beispiele:
   8 acht
   0.8 null komma 8
   .8 null komma 8
   13.0 dreizehn komma 0
   13 dreizehn

2. Exponentialschreibweise
   Diese Form kombiniert eine ganze Zahl oder eine Fließkommazahl mit einer Potenz von 10
   Beispiele:
   5737.9E-4
   0.13E2
   0e0
   0.21E-2

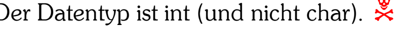
Meist ordnet der Compiler einer Fließkoma-Konstante als default den Datentyp double zu. Wie bei Integerkonstanten kann der Programmierer auch bei Fließkommakonstanten diese Default-Zuweisung durch das Hinzufügen von Suffixen verändern:

L durch das Suffix L wird **long double** erzwungen.

F durch das Suffix F wird **float** erzwungen.

Zeichenkonstante

Eine Zeichenkonstante ist ein einzelnes Zeichen das innerhalb einfacher Anführungszeichen steht.

Der Datentyp ist int (und nicht char). 

Zeichenkonstanten werden intern also als ganzzahliger Wert abgelegt (ASCII-Wert) (und schließen nicht wie Zeichenkette-Konstanten mit dem ASCII-Wert 0 ab).

Beispiele:

'x'
'2'
' '
'*'

```
#include <stdio.h>

void main()
  {
    printf ("Zeichenkonstante 'x' belegt %d Byte",sizeof('x'));  // zB 4 Byte (bei openwatcom 1 Byte)
  }
```

Zeichenkettekonstante

Eine Zeichenkette-Konstante ist eine Folge von beliebigen Zeichen, die innerhalb von doppelten Anführungszeichen stehen.

Im Unterschied zur Zeichenkonstante schließt die Zeichenkette mit dem ASCII-Wert 0 ab.

```
#include <stdio.h>

void main()
{
printf ("himmel \
  arsch und zwirn.." );

}
```

Zwei benachbarte Zeichenkettekonstanten werden vom Compiler zu einer zusammengefügt:

```
#include <stdio.h>

void main()
{
  printf ("himmel \
  arsch " "arsch und zwirn!" );
}
```

Kontrollstrukturen

Der lineare Programmablauf führt einzelne Anweisungen der Rehe nach aus.
Mit Kontrollstrukturen kann der ansonsten lineare Programmablauf geändert werden. Dies geschicht durch:

1. **Verzweigung** (Alternative)
   Damit können Anweisungen in Abhängigkeit von bestimmten Bedingungen durchgeführt werden.
   Anders gesagt: Wählen Anweisungen unter bestimmten Voraussetzungen aus oder übergehen sie.
   Man unterscheidet zwischen einfachen und mehrfachen Entscheidungen:
   a) mit *if* wird eine einfache (einseitige) Entscheidung (Alternative) getroffen
   b) mit *if else* wird eine zweiseitige Entscheidung getroffen (eine einfache Entscheidungen und deren konträr Entscheidung)
   c) mit *switch* werden mehrfache Entscheidung getroffen.

2. **Wiederholung**
   Mit Schleifen (Loops) kann ein bestimmter Programmteil wiederholt werden.
   Es gibt 3 Arten:
   a) die *for*-Schleife (Schleife, in der eine bestimmte Variable einen vorgegebenen Wertebereich durchläuft)
   b) die *while*-Schleife (Schleife, die solange läuft, solange eine Bedingung erfüllt ist)
   c) die *do-while*-Schleife (Schleife, die solange läuft, solange eine Bedingung erfüllt ist)

3. **Sprunganweisungen**
   Mit *continue* wird zum Schleifenanfang gesprungen, mit *break* wird eine Schleife verlassen und mit *goto* wird zu einem Label gesprungen.

Weitere Möglichkeiten, den Programmablauf zu steuern:

)a Nur C++: mit Exceptions werden Fehler abgefangen und zu anderen Programmteilen gesprungen: *try*, *throw* und *catch*.
)b mit Funktionen werden Programmteile zu Unterprogrammen (zum diesem Unterprogramm wird mit Angabe des Funktionsnamen gesprungen)
)c Zusicherungen (asserts)
ermöglicht die automatische Überwachung von Voraussetzungen und Bedingungen an bestimmten Punkten im Programmfluß.

Verzweigung

Eine Entscheidung ist ein Befehl, der eine Bedingung testet und in Abhängigkeit dieser bestimmte Anweisungen durchführt.
Dies ermöglicht, daß ein Programmteil nur in einem speziellen Fall (wenn die Bedingung erfüllt ist) durchlaufen wird. Tritt dieser Fall nicht ein (Bedingung ist nicht erfüllt), wird dieser Programmteil (Programm-Block) übersprungen.
Soll ein Programmteil nur dann durchlaufen werden, wenn die Bedingung nicht erfüllt ist, spricht man von einem Alternativ-Block (else Block). Er ist also die Alternative, wenn der spezielle Fall nicht eintritt.

If Anweisung

Die Ausführung von Anweisungen wird von einer Bedingung abhängig gemacht:
if (*logischer Ausdruck*) *Anweisung*;

Folgen mehrere Aweisungen, müssen diese in geschweifte Klammern gesetzt werden:
if (*logischer Ausdruck*) { *Anweisung1*; *Anweisung2*; }
Die Anweisung1 und Anweisung2 werden nur dann durchgeführt, wenn der logische Ausdruck (also die Bedingung) wahr ist (Bedingung trifft zu).

Anders gesagt: die **if** Anweisung ist eine bedingte Programmverzweigung

Beispiel:

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

void main()
{
  float x,y = -1;          // Variable x wird als Datentyp float deklariert
                           // Variable y wird mit dem Wert -1 definiert

  printf("zahl: ");            // text zahl: ausgeben

  scanf ("%g",&x);             // Variable X wird die von der Tastatur eingegebene Zahl zugewiesen

  if (x >= 0) y=sqrt(x);               // ist x größer-gleich 0, wird y das Ergebnis der Funktion sqrt zugewiesen

  printf ("quadratwurzel: %g\n",y);    // Wert der Variable y ausgeben
}
```

Der Ausdruck x>=0 ist die Bedingung. Sie wird Vergleichsausdruck und logischer Ausdruck genannt.
Die boolesche Logik besitzt nur 2 Zustände: 0 für falsch (false) und 1 für wahr (true).
Ist der Ausdruck wahr, wird die if-Anweisung ausgeführt. Ist sie falsch, wird die if-Anweisung übersprungen und somit nicht ausgeführt.

Beispiel Signum-Funktion: berechnet Vorzeichen einer Zahl (+1 für positive Zahlen, -1 für negative und 0 für Null):

```
#include <stdio.h>
void main()
{
  int x = -34 ;
  int signum=2;

  if (x > 0) signum = 1;
  if (x < 0) signum = -1;
  if (x == 0) signum = 0;

  printf("signum: %i \n",signum);
}
```

If else Anweisung

Ist der Ausdruck false, wird der else-Block der If-else-Anweisung ausgeführt.

if (*logischer Ausdruck*) Anweisung1;
else Anweisung2;

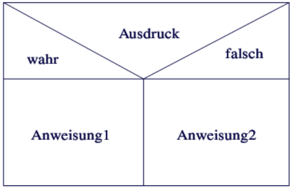
Folgen mehrere Aweisungen, müssen diese in geschweifte Klammern gesetzt werden:
if (*logischer Ausdruck*) { Anweisung1; Anweisung2; }
else {Anweisung3; Anweisung4;}

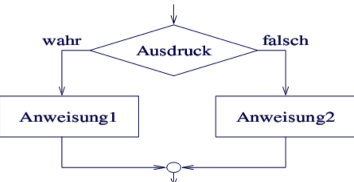
Anders gesagt: die *if else* Anweisung ist eine bedingte Programmverzweigung mit einem zusätzliche Alternativzweig.

Beispiel:

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

void main()
{
  float x,y = -1; // Variable x wird als Datentyp float deklariert
                  // Variable y wird mit dem Wert -1 definiert

  printf("zahl: "); // text zahl: ausgeben

  scanf ("%g",&x);  // Variable X wird die von der Tastatur eingegebene Zahl zugewiesen

  if (x >= 0) y=sqrt(x);  // ist x größer-gleich 0, wird y das Ergebnis der Funktion sqrt zugewiesen
    else printf ("zahl ist negativ\n");   // wenn x kleiner 0, dann text "zahl ist negativ" ausgeben

  printf ("quadratwurzel: %g\n",y);  // Wert der Variable y ausgeben
}
```

Links das Struktogramm, rechts das Flussdiagramm einer *if else* Anweisung:





Beispiel Signum-Funktion: berechnet Vorzeichen einer Zahl (+1 für positive Zahlen, -1 für negative und 0 für Null):

```
#include <stdio.h>
void main()
{
  int x = 0 ;
  int signum=2;

  if (x > 0)
    signum = 1;
  else
    if (x < 0)
      signum = -1;
    else
      signum = 0;

  printf("signum: %i \n",signum);
}
```

Ergibt mit gesetzten Klammern diese Blöcke:

```
#include <stdio.h>
void main()
{
  int x = -3 ;
  int signum=2;

  if (x > 0)  { signum = 1; }
  else
    {
      if (x < 0) { signum = -1; }
      else
        {
          signum = 0;
        }
    }
  printf("signum: %i \n",signum);
}
```

Im ersten else Block werden alle weiteren if else Anweisungen geschachelt (kaskadiert).
Anders wie in diesem PHP-Beispiel:

```
  if ($x<2) {echo "drin1";}
  elseif ($x<1) {echo "drin2";}
  elseif ($x<0) {echo "drin3";}
  else {echo "drin4";}
```

Das letze *else* schließt das erste *if* von der ersten Zeile. Somit steht jeder *else* Block für sich allein.
In C müßte es so geklammert werden:

```
if (x < 2) { puts("drin1");}
  else
    {
      if (x < 1) {puts("drin2");}
      else
        {
          if (x < 0) {puts("drin3"); }
            else
              {
                puts("drin4");
              }
        }
    }
```

Dangling Else Problem
Im folgenden Beispiel bezieht sich das else auf das zweite if. B ist somit 0 (und nicht 42)

```
  #include <stdio.h>
  void main()
  {
     int b=0, a = 0;

     if (a == 1)
       if (b == 1)
         a = 42;
     else
       b = 42;

      printf ("b ist %i \n",b); // b ist 0
  }
```

Geklammert ergibt das diese Blockstruktur:

```
    #include <stdio.h>
    void main()
      {
        int b=0, a = 0;

        if (a == 1)
        {
          if (b == 1)
            {
              a = 42;
            }
            else
              {
                b = 42;
              }
        }

        printf ("b ist %i \n",b); // b ist 0
      }
```

switch Anweisung

Ist der Ausdruck sdf sdf

Schleifen (Loops)

Schleifen (loops) sind Blöcke, die die darin enthaltenen Befehle öfters ausführen.
Die Anzahl der Ausführung hängt von einer Bedingung ab.
C kennt drei Arten von Programmschleifen:

1. for statement
2. while statement
3. do statement

for-Schleife

Die Schleife besteht aus einem Kopf und einem Rumpf. Der Kopf bestimmt, wie oft der Programmblock im Rumpf ausgeführt wird.
Der Schleifenkopf besteht aus 3 Teilen (die durch ein Semikolon getrennt werden):

1. init expression: die Zählvariable (counter) initialisieren (is used to set before the loop begins)
2. loop condition: die Schleifen-Bedingung festlegen (looping continues as long as this condition is satisfied)
3. loop expression: die Veränderung (meist De-oder Inkrementierung) der Zählvariable definieren

Kopf-Rumpf-Struktur der for-Schleife:

for (*Initialisierung* ; *Bedingung* ; *De-Inkrementierung*)
{
Anweisungen (statements)
}

Solange die Bedingung (condition) erfüllt ist, wird die Schleife (loop) ausgeführt.

Beispiel:

```
#include <stdio.h>

int main(void)
  {
    int n, summe;
    summe=0;

    for (n=1;n<=10;n++)
      {
        printf("\nschleifenzaehler: %i",n);
        summe=summe+n;

      }

    printf("\n\ntriangular number: %i \n\n",summe);  // 55

    // the sum of the integers from 1 through n is known as a triangular number

    return 0;
  }
```

while-Schleife

Solange die Bedingung (condition) erfüllt ist, wird die Schleife (loop) ausgeführt.
Kopf-Rumpf-Struktur der for-Schleife:

while (*Bedingung*)
{
Anweisungen (statements)
}

do-Schleife

Die do-while-Schleife wird mindestens einmal durchlaufen, während die while-Schleife eventuell gar nie durchlaufen wird!

do
{
Anweisungen
}
while (*Bedingung*) ;

Felder (Arrays)

Felder sind eine Zusammenfassung mehrer Variablen vom gleichen Datentyp unter einem einzigen Namen.
Jedes Element eines Arrays kann eindeutig über die Kombination von Name und Index angesprochen werden.
Arrays werden auch »Vektoren«, »Felder« und »Reihungen« genannt.
Die Variablendefinition besteht aus:

Datentyp Arrayname [Anzahl der Elemente]

Die Größe des Feldes (Anzahl der Elemente) wird duch eine Integer-Konstante bestimmt
Beispiele:

```
int iWoche[52];          /* Feld mit 52 Elementen vom Typ int */
char szMsg[100];         /* Zeichenkette mit maximal 100 Zeichen */
float fBetrag[10];       /* 10 Elemente vom Typ float */
```

Die einzelnen Elemente eines Array werden durch den Arraynamen und dem jeweiligen Indexwert (der in eckigen Klammern steht) angesprochen. Der Indexwert fängt bei Null an.
Die Elemente eines Arrays belegen in C einen zusammenhängenden Speicherbereich (sie befinden sich der Reihe nach an aufeinanderfolgenden Adressen).

Beispiel1:

```
#include <stdio.h>

void main()
{
  int crap[10]; // 0 bis 9

  printf ("crap belegt %d byte\n\n",sizeof(crap));

  printf("Adresse des ersten Elements: %lu\nAdresse des letzten Elements: %lu\n",&crap[0],&crap[9]);

  printf("\n\nzwischen dem ersten und dem letzten Element sollten 36 bytes liegen ");

}
```

Beispiel2:

```
#include <stdio.h>

void main()
{
  int crap[5]; // 0 bis 4; durch diese Deklaration wird Speicherplatz für 5 Arraywerte reserviert (zB 20 bytes)
}
```

☠ Die Größe eines Arrays kann während der Laufzeit nicht geändert werden und muß somit bereits vor dem Compilieren feststellen.

Array-Initialsierung und Zugriff auf Arraywerte

Beispiel1:

```
#include <stdio.h>

void main()
{
  int i [5];  // Array namens i wird mit 5 Arraywerten deklariert

  i[0]=5;     // Arrayinitialiserung: dem ersten Arraywert (mit Index 0) wird der Wert 5 zugewiesen
  i[1]=33;  // Arrayinitialiserung: dem zweiten Arraywert (mit Index 1) wird der Wert 33 zugewiesen
  i[2]=85;    // Arrayinitialiserung: dem dritten Arraywert (mit Index 2) wird der Wert 85 zugewiesen
  i[3]=421;
  i[4]=93;
  printf ("wert 0 = %d\n",i[0]);
  printf ("wert 1 = %d\n",i[1]);
  printf ("wert 2 = %d\n",i[2]);
  printf ("wert 3 = %d\n",i[3]);
  printf ("wert 4 = %d\n",i[4]);
}
```

Beispiel2:

```
#include <stdio.h>

void main()
{
  int c[5]; // Array namens i wird mit 5 Arraywerten deklariert
  int i;
  for (i=0;i<=8 ;i++ )
  {
    c[i]=i*10;       // Wertebereich von Array namens c wird bei Indexwert 5 überschritten ☠
                     // c[5] wird den Wert 50 nur solange behalten, bis sein Speicherbereich
                     // vom eigentlichen Besitzer überschrieben wird (zB eine andere Variable)

    printf("wert %d = %d \n",i,c[i]);
  }
}
```

☠ Das Über- und Unterschreiten eines Arrays wird vom Compiler nicht überprüft (weder beim Compilieren noch zur Laufzeit) und somit nicht beanstandet.
Erst durch die Compiler-Option range-checking wird das Über- und Unterschreiten zumindest während der Laufzeit überprüft. Da diese Option das Laufzeitverhalten verschlechtert, sollte sie nur zum Debuggen verwendet werden.

Schleifenzähler mit einem #define als Konstante definieren:

Beispiel:

```
#include <stdio.h>

# define TAGE 7   // Konstante namens TAGE

void main()
{
  int c[TAGE];
  int i;
  for (i=0;i<TAGE ;i++ )
  {
    c[i]=i*100;
    printf("Tag %d = %d \n",i,c[i]);
  }
}
```

Array gleich bei der Deklaration initialisieren:

Beispiel:

```
#include <stdio.h>

void main()
{
  int c[]= {1,2,5};
  int i;
  printf("Array belegt %d Bytes",sizeof(c));
}
```

Zeiger (Pointer)

Bei der Deklaration einer Variable wird vom Computer Speicher reserviert. Der Computer merkt sich die Startstelle des Speicherbereiches. Und diese Startadresse wird Zeiger genannt. Die Endadresse hängt vom Datentyp ab.
Somit bestimmt der Variabletyp die Größe des reservierten Speicher. Bei *long int* wird mehr Speicher benötigt als bei *int*.

Anders gesagt: Ein Pointer ist ein Zeiger auf eine Speicheradresse (Speicherstelle). Somit ist der Pointer-Inhalt eine Speicheradresse (und nicht der eigentliche Variablewert).

Um an den Variablewert zu kommen, muss man expliziert dereferenzieren (Adressaufösung). Für die korrekte Dereferenzierung muss man den Datentyp der Speicheradresse wissen (denn die Adresse ist für jeden Datentyp die selbe). Erst durch den Datentyp kennt der Computer die Endadresse.

Eigentlich sollte die Angabe des Datentyps unnötig sein, da jede Speicheradresse gleich gross ist und somit gleich viel Speicher beansprucht.
Die Notwendigkeit ergibt sich durch die Möglichkeit, Zeiger durch Addition (Inkrementierung)
und Subtraktion (Dekrementierung) verändern zu können. Wird zB ein Pointer inkrementiert (nächste Speicheradresse), muß der Computer wissen, um wieviel erhöht werden muß. Die Erhöhung ist abhängig vom Platzbedarf des Datentyps. Bei int werden *sizeof(int)* benötigt (zB 4 Byte). Bei long wird um *sizeof(long)* erhöht (zB 8 Byte)

Near Pointer & far pointer

Unter MS-DOS ist der Speicher in Segmente von je 64 KByte aufgeteilt. Man unterscheidet zwischen einem »*near pointer*« und einem »*far pointer*«:

- der *near pointer* enthält nur den Offset; ein Adreßwert, der sich auf eine Speicherzelle innerhalb des eigenen Segments bezieht
- der *far pointer* beinhaltet zusätzlich die Segmentadresse; deshalb benötigt dieser 4 Byte und der *near pointer* nur 2 Byte

Beispiel:

```
#include <stdio.h>

int main(void)
{
   int *piToInt, far *lpiToInt;
   printf ("\nnear pointer: %d, far pointer: %d\n",sizeof(piToInt),sizeof(lpiToInt));
   return 0;
}
```

Bei einem 32-Bit-OS herrscht eine lineare Speicherverwaltung, bei der jeder Pointer auf jede x-beliebige Speicherzelle zeigen kann. Somit entfallen die für DOS-Compiler üblichen Schlüsselwörter near und far pointer.

Zeiger-Operationen

Mögliche Zeigeroperationen:
1. Indirektionsoperator * (Zeiger-Deklaration)
2. Adressoperator & (Zeiger-Initialisierung)
3. Inkremetierung
4. Dekrementierung
5. Differenz (Abstand) von 2 Zeigern als Anzahl der Feldelemente
6. Vergleich von 2 Zeigern

Deklaration eines Zeigers mit dem Indirektionsoperator

Die Deklaration besteht aus dem

4. Datentyp
5. dem Indirektionsoperator *
6. der Variablebezeichnung:

Datentyp   *Zeigervariable

Der Datentyp des Zeigers muß vom selben Datentyp sein, auf den er zeigt.

Der Stern ist der Indirektionsoperator (er kennzeichnet den Datentyp Zeiger) und befindet sich zwischen dem Datentyp und der Variablebezeichnung.
Das Anwenden des Indirektionsoperator auf einen Zeiger wird als "Dereferenzierung der Zeigervariable" bezeichnet.

Beispiel:

```
#include <stdio.h>
int main()
{
  int *zeiger1;
  int* zeiger2;

  char *zeiger3;
  char* zeiger4;

  float *zeiger5;
  float* zeiger6;

  int* zeiger7, zeiger8;     /* ☠ Nur zeiger7 ist ein Pointer (zeiger8 ist ein gewöhnlicher integer) */

  int *zeiger9, zeiger10;    /* ☠ Nur zeiger9 ist ein Pointer (zeiger10 ist ein gewöhnlicher integer) */

  char *ch1, *ch2;             /* ch1 und ch2 sind Zeiger auf den Typ char */

  return 0;
}
```

Initialisierung des Zeigers

Nach der Deklaration wird der Zeiger mit dem Adressoperator & initialisiert.

zeiger = &variable;

Beispiel:

```
#include <stdio.h>
int main(void)
  {
    int *blub,b;
    b=9;
    blub=&b;
    printf("\ndec address von b: %d ",blub);
    printf("\nhex address von b: %p \n",blub);
    printf("\nwert von b: %d ",*blub); // indirekter Zugriff auf Variable
    printf("\nwert von b: %d \n",b);   // direkter Zugriff auf Variable
    return 0;
  }
```

Den Zugriff auf den Inhalt einer Variablen über den Variablennamen nennt man direkten Zugriff. Und den Zugriff auf den Inhalt einer Variablen über einen Zeiger auf diese Variable nennt man indirekten Zugriff (Indirektion).

Wild Pointer & Null Pointer

Die Verwendung von nicht initialisierten Zeigern (*wild pointer*) führt zu lustigen Ergebnissen (da sie auf irgendeinen Speicherbereich zeigen). Um wilde Zeiger zu vermeiden, können Zeiger gleich als Null-Pointer initialisert werden:

long *zeiger=NULL;

NULL ist eine in *stdio.h* definierte Konstante (wie ein NULL-Pointer intern abgebildet wird, hängt somit vom jeweiligen Compiler ab).
Wild pointer können auch entstehen, wenn die Pointer auf Speicherbereiche zeigen, die nicht mehr verwendet werden.

Array als Zeiger

Der Array-Name ohne eckige Klammern ist ein Zeiger auf das erste Element und ist eine Zeigerkonstante.

☠ Auf eine Zeigerkonstante kann keine Zeiger-Arithmetik angewendet werden.

Beispiel:

```
#include <stdio.h>

int main(void)
  {
    int feld[2], *zeiger;
    feld[0]=91;
    feld[1]=93;

    printf("\n\naddress: %d ",feld); //  0012FF7C
    printf("\n\naddress: %d \n",&feld);  //  0012FF7C
    printf("\nArraywert[0]: %d \n\n",*feld);  //  91

    return 0;
  }
```

Zeiger-Arithmetik

Die Elemente eines Arrays werden in sequentieller Reihenfolge im Speicher abgelegt. Das erste Element [0] befindet sich also an der niedrigsten Adresse. Das zweite Element befindet sich an der Startadresse plus Datentyplänge. Bei int zB 4 Byte: die Adresse jedes Array-Element ist um vier größer als die Adresse des Vorgängers (331245052, 1245056, 1235060, …)
Wird ein Zeiger um 1 erhöht (inkrementiert), weist der Zeiger ab nun auf das nächste Feld-Element. Der Computer erhöht also die alte Adresse um die Größe des Datentyps.

Beispiel:

```
#include <stdio.h>

int main(void)
  {
    long int  feld[3], *zeiger;
    feld[0]=91; feld[1]=93; feld[2]=95;

    zeiger=feld;
    printf("\n\nArraywert [0]: %d ",*zeiger);  //  Array [0]: 91
    printf("\nArray [0] address: %d",zeiger);  //  Array [0] address: 1245040
    printf("\nAbstand: %d",zeiger-feld);  //  Abstand: 0                                  Differenz
    if (zeiger>feld) {printf("\nArray [0] > Startelement");}
    if (zeiger==feld) {printf("\nArray [0] == Startelement");} // Array [0] == Startelement    Vergleich
    if (zeiger<feld) {printf("\nArray [0] == Startelement");}



    zeiger++;                                   // Inkrementierung
    printf("\n\nArraywert [1]: %d ",*zeiger);  //  Array [1]: 91
    printf("\nArray [1] address: %d",zeiger);  //  Array [1] address: 1245040
    printf("\nAbstand: %d",zeiger-feld);  //  Abstand: 0                                  Differenz
    if (zeiger>feld) {printf("\nArray [1] > Startelement");} // Array [1] > Startelement      Vergleich
    if (zeiger==feld) {printf("\nArray [1] == Startelement");}
    if (zeiger<feld) {printf("\nArray [1] == Startelement");}

    zeiger++;                                 // Inkrementierung
    printf("\n\nArraywert [2]: %d ",*zeiger);  //  Array [2]: 91
    printf("\nArray [2] address: %d",zeiger);  //  Array [2] address: 1245040
    printf("\nAbstand: %d",zeiger-feld);  //  Abstand: 0                                  Differenz
    if (zeiger>feld) {printf("\nArray [2] > Startelement");} // Array [2] > Startelement      Vergleich
    if (zeiger==feld) {printf("\nArray [2] == Startelement");}
    if (zeiger<feld) {printf("\nArray [2] == Startelement");}

    printf("\n\n");
    return 0;
  }
```

Array an eine Funktion übergeben

Eine direkte Übergabe des Arrays an die Funktion ist nicht möglich. Darum wird lediglich die Startadresse als Zeiger übergeben.
Es gibt 3 Möglichkeiten wie die Funktion an die Länge des Arrays kommt:

1. das Ende des Arrays kann durch einen bestimmten Feldwert gekenntzeichnet sein; zB -666
2. die Länge des Arrays ist ein weiterer Übergabewert
3. die Länge ist eine globale Variable und somit auch innerhalb der Funktion verfügbar

8. Dynamische Speicherverwaltung

Die

• <u>Ansi754 / 32 Bit in Fließkommazahl konvertieren:</u>

Im Big Endian 32 Bit:

0 10011101 00111001100000000001101

ist hexadezimal: 4E 9C C0 0D

Aus
00111001100000000001101
mach ich:
1.00111001100000000001101

Ich habe eine 1 und ein Komma vorne hinzugefügt (also die normalisierte Anzeige).
Dezimal: 1.224610924720764

Das höchstwertige Bit, das ganz rechts ist (Also das 31. Bit, Zählung bei Null beginnend) ist für das Vorzeichen.
0 bedeutet positiv und 1 negativ.

Die nächsten acht Bit sind der Exponent (Bit 23 bis Bit 30).
10011101 ist dezimal 157.
Von den 157 ziehe ich 127 ab und erhalte 30 (also den Exponenten).

1,224610924720764 * 2 hoch 30

• <u>Ansi754 / 64 Bit in Fließkommazahl konvertieren:</u>

Im Big Endian 64 Bit:

11.001000111101011100001010001111010111000010100011110

Normalisieren:

1.1001000111101011100001010001111010111000010100011110

1001000111101011100001010001111010111000010100011110

Der Exponent ist 1 (nur eine Verschiebung) und wird mit 1023 addiert: 1024 (10000000000)

0 10000000000 1001000111101011100001010001111010111000010100011110

Zur Mantisse füge ich die Ziffer 1 und das Komma hinzu:
1.1001000111101011100001010001111010111000010100011110
und das ist dezimal: 1.57
Der Exponent ist 1024-1023 = 1

1.57 * 2 hoch 1 = 3.14

Binäre Darstellung byteweise gruppiert:

01000000 00001001 00011110 10111000 01010001 11101011 10000101 00011110

• Fließkommazahl in Ansi754 konvertieren - Beispiel 1:

Gegeben sei die dezimale Fließkommazahl 3.14.
Ich wandle die Zahl in die Basis 2 um: Der Vorkommateil 3 ist binär: 11
Dann wandle ich den Nachkommteil 0.14 um (das Basis 2 ist, stets mit 2 multiplizieren):

ich multipliziere 0,14 mit 2: 0,28 die Ziffer vor dem Komma ist eine Null
diese Null ist die erste binäre Ziffer hinter dem Komma: 11.0

das Ergebnis 0,28 mal 2: 0,56 die Ziffer vor dem Komma ist eine Null
diese Null ist die zweite binäre Ziffer hinter dem Komma: 11.00

das Ergebnis 0,56 mal 2: 1,12 die Ziffer vor dem Komma ist eine Eins. Diese Ziffer ist die dritte binäre
Ziffer hinter dem Komma: 11.001

und so fort und erhalte: 11.00100011110101110000010

Zu Beginn ist der Exponent Null.
Jetzt muss die Zahl normalisiert werden. Das bedeutet, dass es nur eine Ziffer (und zwar die Eins) vor
dem Komma gibt. Diese Normalisierung erfolgt durch das Verschieben des Kommas:
aus
11.00100011110101110000010
wird
1.100100011110101110000010
Das Komma wurde um eine Stelle nach links verschoben.
Der Exponent wird um 1 erhöht und hat nun den Wert 1.
Die 1 vor dem Komma wird NICHT für die Mantisse verwendet.

Ein Beispiel für ein Rechsverschieben:
Aus
0.00100100000111111011010111110
wird
1.00100000111111011010111110
Der Exponent wird um 3 erniedrigt und hat nun den Wert -3.

Natürlich kann auch ein Fall eintreten, indem die Mantisse nicht mehr normalisiert werden muss, da bereits nur eine Ziffer (und zwar der Einser) vor dem Komma steht.

Der Exponent mit dem Wert 1 wird um 127 erhöht: 128 (binär 10000000).

Das Vorzeichenbit ist Null, da die Zahl positiv ist (ist die Zahl negativ, ist das Vorzeichenbit eine 1).

Vorzeichen	Exponent	Mantisse
0	10000000	10010010000111111011010

Zur Mantisse füge ich die Ziffer 1 und das Komma hinzu:
1.10010010000111111011010
und rechne das in das dezimale System um: 1.5699999
Exponent dezimal 128

1,5699999 * 2 hoch (128-127) = 3,1399998

Binäre Darstellung byteweise gruppiert:

01000000	01001001	00001111	11011010
40	49	0F	DA

• Fließkommazahl in Ansi754 konvertieren - Beispiel 2:

*P*ut 0.085 in single-precision format

The first step is to look at the sign of the number.
Because 0.085 is positive, the sign bit =0.

$(-1)^0 = 1$.

Write 0.085 in base-2 scientific notation.
This means that we must factor it into a number in the range [1 <= n < 2] and a power of 2.

$0.085 = (-1)^0 * (1+\text{fraction}) * 2^{\text{power}}$, or:
$0.085 / 2^{\text{power}} = (1+\text{fraction})$.

So we can divide 0.085 by a power of 2 to get the (1 + fraction).

$0.085 / 2^{-1} = 0.17$
$0.085 / 2^{-2} = 0.34$
$0.085 / 2^{-3} = 0.68$
$0.085 / 2^{-4} = 1.36$

Therefore, $0.085 = 1.36 * 2^{-4}$

Find the exponent.
The power of 2 is -4, and the bias for the single-precision format is 127. This means that the exponent = 123_{ten}, or 01111011_{bin}

Write the fraction in binary form
The fraction = 0.36 . Unfortunately, this is not a "pretty" number, like those shown in the book. The best we can do is to approximate the value. Single-precision format allows 23 bits for the fraction.

Binary fractions look like this:

$0.1 = (1/2) = 2^{-1}$
$0.01 = (1/4) = 2^{-2}$
$0.001 = (1/8) = 2^{-3}$

To approximate 0.36, we can say:

0.36 = (0/2) + **(1/4)** + (0/8) + **(1/16)** + **(1/32)** +...
$0.36 = 2^{-2} + 2^{-4} + 2^{-5}+...$

$0.36_{ten} \sim 0.01011100001010001111011_{bin}$.

The binary string we need is: 01011100001010001111011.

It's important to notice that you will not get 0.36 exactly. This is why floating-point numbers have error when you put them in IEEE 754 format.

Now put the binary strings in the correct order -
1 bit for the sign, followed by 8 for the exponent, and 23 for the fraction. The answer is:

	Sign	Exponent	Fraction
Decimal	0	123	0.36
Binary	0	01111011	01011100001010001111011

• Ansi754 in Fließkommazahl konvertieren - Beispiel 1:

Convert the following single-precision IEEE 754 number into a floating-point decimal value.

1 10000001 10110011001100110011010

First, put the bits in three groups.
Bit **31** (the leftmost bit) show the sign of the number.
Bits **23-30** (the next 8 bits) are the exponent.
Bits **0-22** (on the right) give the fraction

Now, look at the sign bit.
If this bit is a 1, the number is negative.
If it is 0, the number is positive.

This bit is 1, so the number is negative.

Get the exponent and the correct bias.
The exponent is simply a positive binary number.
$10000001_{bin} = 129_{ten}$

Remember that we will have to subtract a bias from this exponent to find the power of 2. Since this is a single-precision number, the bias is 127.

Convert the fraction string into base ten.
This is the trickiest step. The binary string represents a fraction, so conversion is a little different.

Binary fractions look like this:

$0.1 = (1/2) = 2^{-1}$
$0.01 = (1/4) = 2^{-2}$
$0.001 = (1/8) = 2^{-3}$

So, for this example, we multiply each digit by the corresponding power of 2:

$0.10110011001100110011010_{bin} = \mathbf{1*2^{-1}} + 0*2^{-2} + \mathbf{1*2^{-3}} + \mathbf{1*2^{-4}} + 0*2^{-5} + 0 * 2^{-6} + \ldots$
$0.10110011001100110011010_{bin} = 1/2 + 1/8 + 1/16 + \ldots$

Note that this number is just an approximation on some decimal number. There will most likely be some error. In this case, the fraction is about 0.7000000476837158.

This is all the information we need. We can put these numbers in the expression:

$(-1)^{\text{sign bit}} * (1+\text{fraction}) * 2^{\text{exponent - bias}}$

$= (-1)^{1} * (1.7000000476837158) * 2^{129-127}$
= -6.8

The answer is approximately -6.8.

• <u>Bitweise addieren</u>

Es gelten diese 4 Regeln:

a) 0+1=1
b) 0+0=0
c) 1+0=1

Wird 1 und 1 addiert, gibt es einen Übertrag von 1:

d) 1+1=0 (Übertrag ist 1)

```
   10100111        167
 + 01110010      + 114
  ---------       ----
  100011001       281
```

• <u>Bitweise subtrahieren</u>

Es gelten diese 4 Regeln:

a) 1-0=1
b) 0-0=0
c) 1-1=0

Wird 0 und 1 subtrahiert, gibt es einen Übertrag von 1:

d) 0-1=1 1. Übertrag ist 1
2. subtrahiere
3. überprüfen, ob für die nächste Subtraktion wieder ein Übertrag ist
4. mit dem Ergebnis der Subtraktion subtrahiere ich den Übertrag

Beispiel:

```
   10010011101     1181
 - 01101110100    - 884
  -----------     -----
  00100101001       297
```

Beispiel:

```
   10011100110     1254
 - 01101101101    - 877
  -----------     -----
  00101111001       377
```

• Bitweise multiplizieren

Es gelten diese 4 Regeln:
a) 1*1=1
b) 1*0=0
c) 0*1=0
d) 0*0=0

Beispiel:
```
1101 * 1001              13*9=117
----
1101
 0000
   0000
    1101
-------
1110101
```

Beispiel:
```
111 * 10                 7*2=14
---
111
 000
----
1110
```

Beispiel:
```
101 * 110                5*6=30
---
101
 101
  000
-----
11110
```

• bitweise dividieren

Beispiel:

```
  10:100=1
   0R

 11:10=1.1
   10
    0R

 10:11=0.1
  100
 -11
  ---
  001Rest

  11:1=11
  01
   0Rest

   1:11=0.01
   100
  - 11
   ---
    001R

  1:10=0.1
  10
    0R

  10100:101=100
   00
  0Rest


  111001:11=10011
    100
   - 11
    ---
   0011

  10111:10=1011.1
    11
     11
      10
       0R

  11011101:10=1101110.1
  10
   011
     11
      10
       010
```

- vom dezimalen Zahlensystem ins binäre

Beispiel:

```
256

1. 256:2=128
     5
     16
      0R

2.128:2=64
      0R

3.64:2=32
     4
     0R

4.32:2=16
    12
     0R

5.16:2=8
     0R

6.8:2=4
    0R

7.4:2=2
    0R

8.2:2=1
    0R

9.1:2=0
    1R
```

Beginne bei 9. aufwärts bis 1. die Reste nebeneinander von rechts nach links aufzuschreiben:
100000000

• <u>vom binären Zahlensystem ins dezimale</u>

Beispiel:

210
101 = $(1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 5$
$4 + 0 + 1 = 5$

101 ergibt im dezimalen System 5

Beispiel:

6543210
1101100 = $(1*2^6) + (1*2^5) + (0*2^4) + (1*2^3) + (1*2^2) + (0*2^1) +(0*2^0)$

1101100 ergibt im dezimalen System 108

• vom dezimalen System ins hexadezimale System

Beispiel:

Gegeben ist 547 im dezimalen System:

1. 547 : 16 = 34
   67
    3R

2. 34 : 16 = 2
   2R

3. 2 : 16 = 0
   2R

Beginne bei 3. aufwärts die Reste nebeneinander bis 1. aufzuschreiben.

547 ist im hexadezimalen System 223.

• vom hexadezimalen System in dezimale System

Beispiel:
Gegeben ist 223 im hexadezimalen System:

210
223 = $(2 * 16^2) + (2 * 16^1) + (3 * 16^0) = 547$

223 ist im dezimalen System 547.

Beispiel:
Gegeben ist AA im hexadezimalen System:

10
AA = $(10 * 16^1) + (10 * 16^0) = 170$

AA ist im dezimalen System 170.

• vom hexadezimalen System ins binäre System

Beispiel:
4 binäre Ziffern ergeben 1 hexadezimale Ziffer.
Darum sind 8Bit (1Byte) 2 hexadezimale Ziffer.

1011 0110
B 6 10110110 im binären System sind B6 im hexadezimalen System.

A C
1010 1100 AC im hexadezimalen System sind 10101100 im binären System.