

# Einführung in die Programmiersprache C

Prof. Dr.-Ing. Kai Bruns

12. September 2005

Bei meinen Kollegen Herrn Prof. Beck, Herrn Roeper, Herrn Rüger, Herrn Prof. Schäfer und Herrn Prof. Wöhner möchte ich mich ausdrücklich für die vielen Hinweise bei der Erstellung dieses Lehrheftes bedanken.

Lehrheft für die Ausbildung an der HTW Dresden (FH)

# Inhaltsverzeichnis

0.1	Vorbemerkungen . . . . .	6
<b>1</b>	<b>Lexikale Grundlagen</b>	<b>7</b>
1.1	Zeichenvorrat . . . . .	7
1.2	Trennzeichen und Kommentare . . . . .	7
1.3	Identifikatoren . . . . .	8
1.4	Schlüsselworte . . . . .	8
1.5	Konstanten und Zeichenketten . . . . .	9
1.5.1	Integerkonstanten . . . . .	9
1.5.2	Gleitkommakonstanten . . . . .	9
1.5.3	Zeichenkonstanten . . . . .	10
1.5.4	Zeichenkettenkonstanten . . . . .	10
1.5.5	Programmstruktur . . . . .	11
<b>2</b>	<b>Datentypen, Variablendefinition und Speicherklassen</b>	<b>12</b>
2.1	Grunddatentypen . . . . .	12
2.2	Strukturierte Datentypen . . . . .	13
2.3	Deklaration neuer (eigener) Typbezeichner . . . . .	13
2.4	Variablen . . . . .	14
2.4.1	Initialisierung von Variablen . . . . .	14
2.4.2	Speicherklassen . . . . .	14
2.4.2.1	Die Speicherklasse <code>auto</code> . . . . .	15
2.4.2.2	Die Speicherklasse <code>static</code> . . . . .	15
2.4.2.3	Die Speicherklasse <code>extern</code> . . . . .	16

2.4.2.4	Die Speicherklasse <code>register</code>	16
2.4.3	Typkonvertierung	16
2.4.3.1	automatische Typkonvertierung	16
2.4.3.2	erzwungene Typkonvertierung ( <code>cast-Operator</code> )	17
<b>3</b>	<b>Ausdrücke</b>	<b>18</b>
<b>4</b>	<b>Operatoren</b>	<b>19</b>
4.1	Monadische Operatoren	19
4.1.1	Referenzoperatoren	19
4.1.2	Arithmetische Negation	20
4.1.3	Logische Negation	20
4.1.4	Inkrement- und Dekrementoperator	20
4.1.5	<code>sizeof</code> -Operator	21
4.2	Multiple Operatoren	21
4.2.1	Arithmetische Operatoren	21
4.2.2	Verschiebeoperatoren	22
4.2.3	Relationale Operatoren	22
4.2.4	Gleichheitsoperatoren	22
4.2.5	Bitorientierte Operatoren	23
4.2.6	Logische Operatoren	23
4.2.7	Bedingter Operator	24
4.3	Zuweisungsoperatoren	24
4.3.1	Mehrfachzuweisung	25
4.4	Strukturoperatoren	25
4.5	Kommaoperator	25
4.6	Prioritäten und Abarbeitungsreihenfolgen	26
<b>5</b>	<b>Anweisungen</b>	<b>27</b>
5.1	Marken	27
5.2	Ausdrucksanweisung	27
5.3	Blöcke	28

5.4	Leere Anweisung . . . . .	28
5.5	Steuerstrukturen . . . . .	28
5.5.1	Bedingte Anweisung (if-Anweisung) . . . . .	29
5.5.2	mehrfach bedingte Auswahl (switch-Anweisung) . . . . .	30
5.5.3	Zyklusanweisungen . . . . .	31
5.5.3.1	While-Anweisung . . . . .	31
5.5.3.2	Do-Anweisung . . . . .	32
5.5.3.3	For-Anweisung . . . . .	32
5.5.4	Modifikatoren . . . . .	33
5.5.4.1	Fortsetzungsanweisung . . . . .	34
5.5.4.2	Abbruchanweisung . . . . .	34
5.5.5	Sprunganweisung . . . . .	34
<b>6</b>	<b>Zeiger</b>	<b>35</b>
6.1	Zeigervariable . . . . .	35
6.2	Adressarithmetik . . . . .	36
6.3	Initialisierung von Zeigern . . . . .	37
<b>7</b>	<b>Vektoren</b>	<b>38</b>
7.1	Definition und Handhabung von Vektoren . . . . .	38
7.2	Äquivalenz: Zeiger-Vektoren . . . . .	39
7.3	Äquivalenz: Zeiger-Zeichenketten . . . . .	40
7.4	Vektoren von Zeigern und Zeichenketten . . . . .	40
<b>8</b>	<b>Strukturen</b>	<b>42</b>
8.1	Deklaration von Strukturen . . . . .	42
8.2	Definition von Strukturvariablen . . . . .	43
8.3	Nutzung von Strukturen . . . . .	43
8.4	Initialisierung von Strukturen . . . . .	44
8.5	Rekursive und dynamische Strukturen . . . . .	45
8.6	Bitfelder . . . . .	45
8.7	Aufzählungstypen . . . . .	46

<b>9 Vereinigungen</b>	<b>47</b>
<b>10 Funktionen</b>	<b>49</b>
10.1 Definition von Funktionen . . . . .	49
10.2 Deklaration von Funktionen (Funktionsprototypen) . . . . .	50
10.3 Funktionsaufruf und Parameterübermittlung . . . . .	50
10.4 Funktionsergebnis . . . . .	52
10.5 Zeiger auf Funktionen . . . . .	52
10.6 Argumente aus der Kommandozeile . . . . .	53
10.7 Rekursive Funktionsaufrufe . . . . .	54
<b>11 Präprozessoranweisungen</b>	<b>55</b>
11.1 include-Anweisung . . . . .	55
11.2 define-Anweisung . . . . .	56
11.3 Bedingte Übersetzung . . . . .	57
<b>12 ANSI-C Bibliotheksfunktionen</b>	<b>58</b>
12.1 stdio.h – Standard E/A-Funktionen . . . . .	58
12.1.1 Dateioperationen . . . . .	58
12.1.2 Formatierte Ausgabe . . . . .	60
12.1.3 Formatierte Eingabe . . . . .	62
12.1.4 Ein- und Ausgabe von Zeichen . . . . .	64
12.1.5 Direkte Ein- und Ausgabe . . . . .	65
12.1.6 Positionieren in Dateien . . . . .	65
12.1.7 Fehlerbehandlung . . . . .	66
12.2 stdlib.h – Allgemeine Hilfsfunktionen . . . . .	66
12.3 string.h – Funktionen für Zeichenketten . . . . .	68
12.4 ctype.h – Testen und Umwandeln von Zeichen . . . . .	70
12.5 math.h – Mathematische Funktionen . . . . .	71
12.6 time.h – Datums- und Zeitfunktionen . . . . .	73
<b>A Zugaben</b>	<b>76</b>
A.1 Literatur- und Internetverweise . . . . .	76

## 0.1 Vorbemerkungen

Die Programmiersprache C wurde 1972 von Brian W. Kernighan und Dennis B. Ritchie entwickelt. C fand zunächst über das Betriebssystem UNIX eine schnelle Verbreitung, da es für dessen Systemprogrammierung eine wesentliche Rolle spielte.

Heute gibt es praktisch für fast jede Computerplattform entsprechende C- Compiler. Zwar wurde C immer wieder wegen seiner „theoretischen Mängel“ kritisiert, aber das konnte seinen Erfolg kaum schmälern. Vor allem aus der Systemprogrammierung ist C heute kaum noch wegzudenken. Hinzu kommen eine Vielzahl von Bibliotheken, die C für eine breite Palette von Anwendungen empfehlen. Die Programmierung verteilter Anwendungen in lokalen und globalen Computernetzen, Datenbank Anwendungen, Programme für betriebswirtschaftliche und statistische Berechnungen, die Programmierung von Compilern selbst, wissenschaftliche Simulationen bis hin zu komplexen Multimedia-Anwendungen sind nur Beispiele hierfür.

Das vorliegende Lehrheft versteht sich weder als vollständige Sprachreferenz noch als umfassendes Lehrbuch; für beides gäbe es auch keinen Bedarf mehr, da bereits viele gute Bücher auf dem Markt sind. Das Anliegen des Lehrheftes ist es, in kompakter überschaubarer Form die Grundlagen der Programmiersprache C zusammenzufassen und somit dem Studenten als Kurzreferenz dienen zu können. Durch eine Vielzahl von Beispielen soll es das praktische Nacharbeiten der Vorlesung unterstützen. Die verwendeten Syntaxdiagramme sollen anschaulich in die Problematik einführen. Der Anspruch auch Vollständigkeit wird hier nicht erhoben.

# Kapitel 1

## Lexikale Grundlagen

### 1.1 Zeichenvorrat

Das Alphabet der Programmiersprache C setzt sich aus Buchstaben, Ziffern und Sonderzeichen zusammen.

Die folgenden Zeichen gelten als Buchstaben:

```
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 _
```

als Ziffern:

```
0 1 2 3 4 5 6 7 8 9
```

als Sonderzeichen:

```
! ? " ' ( ) [ ] { } + - < > \ / .
, : ; * & # % ^ ~ |
```

und alle nicht druckbaren Zeichen der ASCII-Codetabelle.

Zwischen Groß- und Kleinbuchstaben besteht ein signifikanter Unterschied. Aus dem Alphabet der Sprache werden Identifikatoren, Schlüsselworte, Konstanten, Zeichenketten, Operatoren, Trennzeichen und Kommentare gebildet.

### 1.2 Trennzeichen und Kommentare

Trennzeichen dienen zur Trennung einzelner Morpheme (Schlüsselworte, Bezeichner, ...) und werden bei der Syntaxanalyse ignoriert. Als Trennzeichen gelten Leerzeichen, Tabu-

latoren, Zeilumbrüche und Kommentare.

Kommentare werden durch die Zeichenkette `/*` eingeleitet und durch `*/` beendet. Eine Schachtelung von Kommentaren ist i.a. nicht zulässig.

```
/* das ist ein kurzer Kommentar */
a=4711;          /* Zuweisung */
```

## 1.3 Identifikatoren

Ein Identifikator dient z.B. zur Bezeichnung von Datentypen, Variablen und Funktionen (oder sogar von Konstanten). Er ist definiert als eine Folge von Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muss. Man beachte aber, dass Identifikatoren in ihrer signifikanten Länge manchmal eingeschränkt sind.

```
tmp, i, access_counter,
umfang, Umfang, file_ptr
```

## 1.4 Schlüsselworte

Schlüsselworte (Wortsymbole) sind reservierte Zeichenfolgen, denen bereits durch die Sprache C eine feste Bedeutung zugeordnet wurde. Sie dürfen also vom Programmierer nicht als Identifikatoren verwendet werden. Die Programmiersprache C kennt folgende Schlüsselworte:

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

Bei einigen Compilern gelten zusätzlich auch folgende Schlüsselworte:

```
pascal    fortran    NULL
```

Bemerkung: Wenn NULL nicht als Schlüsselwort behandelt wird, dann ist es im allgemeinen wie folgt definiert:

```
#define NULL ((void*)0)
```



## 1.5 Konstanten und Zeichenketten

Der Wert von Konstanten bleibt während der Abarbeitung eines Programms unverändert, wobei 4 Arten von Konstanten unterschieden werden: Integerkonstanten, Gleitkommakonstanten, Zeichenkonstanten und Zeichenkettenkonstanten.

Konstanten können einfach durch Angabe ihres Wertes verwendet werden.

Von besserem Programmierstil zeugt es jedoch, wenn Konstanten explizit definiert werden. Das geschieht entweder zentral in sogenannten Headerfiles oder am Anfang eines Programmes und hat den Vorteil, dass Änderungen von Programmkonstanten nur an einer Stelle vorgenommen werden müssen und dann für das ganze Programm gelten.

Zur Definition von Konstanten kann das Schlüsselwort `const` und zu deren Deklaration die Präcompilieranweisung `#define` verwendet werden.

```
#define pi    3.1415926
const    long colors_16bit = 65536;
```

### 1.5.1 Integerkonstanten

Integerkonstanten können dezimal, hexadezimal oder oktal angegeben werden, wobei standardmäßig vom Compiler eine Dezimalzahl angenommen wird. Oktalzahlen werden durch eine führende 0 und Hexadezimalzahlen mit 0x bzw. 0X gekennzeichnet.

In Oktalzahlen stehen die Ziffern 0 bis 7 zur Verfügung, während für Hexadezimalzahlen die folgenden Zeichen zulässig sind:

0	1	2	3	4	5	6	7	8	9		
a	b	c	d	e	f	A	B	C	D	E	F

```
4711, -509           /* dezimal    */
0715, 03             /* oktal    !    */
0x3fff, 0xab         /* hexadezimal */
```

Standardmäßig wird für Integerkonstanten der Typ `int` angenommen. Durch Nachstellen des Zeichens `l` bzw. `L` kann explizit der Typ `long` festgelegt werden.

```
4711l                /* dezimal vom Typ long */
```

### 1.5.2 Gleitkommakonstanten

Gleitkommazahlen sind syntaktisch komplizierter aufgebaut, aber dennoch für die meisten Nutzer intuitiv in ihrer Nutzung. Sie bestehen aus einem ganzzahligen Anteil, einem Dezimalpunkt, dem Bruchteil und dem davon durch `e` bzw. `E` getrennten Exponenten. Ein

Vorzeichen ist sowohl für die Gleitkommazahl selbst als auch für den Exponenten möglich. Wenn ein Exponent angegeben wird, kann der Dezimalpunkt entfallen.

```
.333333333
2.6e-4          /* 2.6 mal 10 hoch -4 */
3E12           /* 3 mal 10 hoch 12  */
```

### 1.5.3 Zeichenkonstanten

Zeichenkonstanten werden in Apostrophen eingeschlossen. Ihr numerischer Wert entspricht dem Wert des Zeichens in der ASCII-Kodetabelle (siehe Seite 77).

```
'f','e','i','n'
```

Einige wichtige nichtdruckbare Zeichen werden explizit wie folgt benannt:

'\n'	-	Newline (neue Zeile, Zeilenvorschub)
'\t'	-	horizontaler Tabulator
'\b'	-	Backspace (Rücksetzen um ein Zeichen)
'\r'	-	Wagenrücklauf (an den Anfang der Zeile)
'\''	-	Apostroph
'\"'	-	Anführungszeichen
'\\'	-	Backslash (Schrägstrich nach links)
'\0'	-	internes Endekennzeichen in C
'\a'	-	kurzes Tonsignal
'\nnn'	-	beliebiges anderes Zeichen, nnn wird als Oktalzahl interpretiert !!

### 1.5.4 Zeichenkettenkonstanten

Zeichenkettenkonstanten werden in Anführungszeichen eingeschlossen. Es sind Anführungszeichen und Steuerzeichen auch innerhalb der Zeichenkette zulässig, wenn sie wie oben beschrieben kodiert werden. Intern endet jede Zeichenkettenkonstante mit einem '\0'-Zeichen, das aber nicht explizit angegeben wird.

```
"Hi Medieninformatiker"          /* Zeichenkettenkonstante */
"Zeichenkettenkonstante mit Zeilenumbruch \n"
```

Werden im Programm Zeichenketten über mehrere Zeilen definiert, so ist jede Zeile mit einem Backslash (\) abzuschließen.

```
"Menu:          \
\n1- Eingeben   \
\n2- Löschen    \
\n3- Anzeigen   \
\n0- Ende"
```

### 1.5.5 Programmstruktur

Die grundlegenden Datenobjekte sind Variablen und Konstanten. Variablen (manchmal auch Konstanten) werden durch einen Identifikator repräsentiert. Es besteht Deklarationszwang vor der ersten Anwendung einer Variablen bzw. einer durch einen Identifikator bezeichneten Konstanten.

Mittels Operatoren werden Variablen und Konstanten zu Ausdrücken verknüpft, wobei ein Ausdruck immer einen Wert liefert. Ausdrücke können mit Operatoren zu neuen Ausdrücken kombiniert werden.

Ein C-Programm besteht aus einer Menge von Funktionen, die den eigentlichen Programmcode, Vereinbarungen und Anweisungen, enthalten. Anweisungen sind durch ein Semikolon abzuschließen und werden entsprechend den angegebenen Steuerstrukturen sequentiell abgearbeitet.

Ein Programm muss genau eine Funktion mit dem Namen `main` besitzen. Dieser Funktion wird bei der Ausführung des Programmes die Steuerung durch das System übergeben, d.h. die erste Anweisung der Funktion `main()` ist auch die erste Anweisung des Programms.

```
#include <stdio.h>

/* Ein einfaches Programm mit nur einer Ausgabe */
void main() {
    printf("Hallo Programmierer\n");
}
```

Im folgenden werden Variablen, Konstanten, berechenbare Ausdrücke und Funktionen oft als Objekte (Datenobjekte) oder, wenn sie als Einheiten der Speicherung betrachtet werden, auch als Speicherobjekte bezeichnet. Dabei werden jedoch in keinsten Weise Konzepte der objektorientierten Programmierung impliziert.

## Kapitel 2

# Datentypen, Variablendefinition und Speicherklassen

### 2.1 Grunddatentypen

Die Programmiersprache C kennt nur wenige Grunddatentypen, die dazu dienen, das Bitmuster auf dem Speicherplatz der Variablen richtig zu interpretieren, d.h. den entsprechenden Wert zu bilden. Die Grunddatentypen werden in C wie folgt bezeichnet:

- `char` - ein Zeichen aus der ASCII-Tabelle
- `int` - ein ganzzahliger Wert
- `float` - eine Gleitkommazahl
- `double` - eine Gleitkommazahl doppelter Genauigkeit
- `enum` - eine Aufzählung

Diese Grunddatentypen können zum Teil noch mit Modifikatoren kombiniert werden, um spezielle Wertebereiche abdecken zu können bzw. eine bestimmte Anzahl von Bytes zu reservieren. Als Modifikatoren sind zulässig: `signed`, `unsigned`, `short`, `long`.

Folgende Wertebereiche ergeben sich bei den möglichen Modifikatoren unter Verwendung des C-Compilers „Visual C++“ von Microsoft:

Datentyp	Bytes	alternativer Name	Wertebereich
int	2/4	signed, signed int	systemabhängig
unsigned int	2/4	unsigned	systemabhängig
char	1	signed char	-128 bis 127
unsigned char	1		0 bis 255
short	2	short int, signed short int	-32768 bis 32767
unsigned short	2	unsigned short int	0 bis 65,535
enum	2		-32768 to 32767
long	4	long int, signed long int	-2147483648 bis 2147483647
unsigned long	4	unsigned long int	0 bis 4294967295
float	4		3.4E +/- 38 (7 Zeichen)
double	8		1.7E +/- 308 (15 Zeichen)
long double	10		1.2E +/- 4932 (19 Zeichen)

Durch den ANSI-Standard ist festgeschrieben, dass die Anzahl der benötigten Bytes für den Datentyp `short` kleiner oder gleich der für den Datentyp `int` und kleiner oder gleich der für den Datentyp `long` zu sein hat.

In C gibt es keinen Datentyp `boolean`. Ein beliebiges Datenobjekt, das mindestens an einer Bitstelle eine 1 aufweist, stellt den Wert `TRUE` (wahr) dar. Ein Datenobjekt, das den Wert `FALSE` (falsch) repräsentiert, besteht auf Bitebene lediglich aus Nullen.

Bemerkung: Vorsicht ist geboten, wenn eine Variable des Typs `float` auf den Wert 0 (`FALSE`) geprüft wird!!!

Der Aufzählungstyp `enum` wird im Zusammenhang von Strukturen und Vereinigungen im Abschnitt 8 ab Seite 42 behandelt.

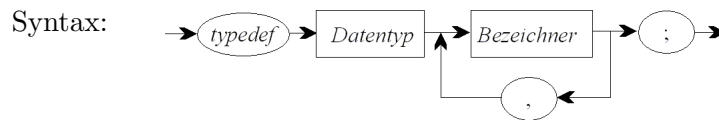
## 2.2 Strukturierte Datentypen

Die Grunddatentypen können zu „beliebig“ komplexen Strukturen zusammengesetzt werden. Da diesem Punkt mehr Aufmerksamkeit gewidmet werden muss, wird er in den Abschnitten über Vektoren (7), Strukturen (8) und Vereinigungen (9) ausführlich behandelt.

## 2.3 Deklaration neuer (eigener) Typbezeichner

Zusätzlich zu den Grunddatentypen kann man sich mit dem Schlüsselwort `typedef` eigene neue Typbezeichner (d.h. neue Namen für bekannte Typen) deklarieren, was insbesondere

für Strukturen sinnvoll ist.



```
typedef int INTEGER, bool;  
#define TRUE 1  
#define FALSE 0  
bool    flag=TRUE;
```

## 2.4 Variablen

Eine Variable ist eine Zuordnung zwischen einem Identifikator, einem Speicherplatz und einem Datentyp. Der Wert der Variable, d.h. der Inhalt des Speicherplatzes, kann beliebig verändert werden. Für Variablen besteht Definitionszwang. Bei einer Variablendefinition wird zunächst der Datentyp angegeben und dann eine Liste mit (durch Komma getrennte) Namen für die Variablen, die von diesem Typ benötigt werden.

```
int alter;  
long a,b,c;  
char eingabe;
```

### 2.4.1 Initialisierung von Variablen

In jeder Variablendefinition kann durch Anfügen einer einfachen Wertzuweisung eine Initialisierung mit einem Wert vorgenommen werden. Variablen ohne Initialisierung sind zunächst in ihrem Wert undefiniert.

```
char z = 'n';  
long summe = 4+5;  
int x = summe-1;
```

### 2.4.2 Speicherklassen

Speicherklassen dienen dazu, die Lebensdauer, die Verfügbarkeit und den Ort der Speicherung (die Adresse) eines Speicherobjektes festzulegen. Prinzipiell werden globale (im gesamten Programm gültige) und lokale (nur in einem Block verfügbare) Variablen unterschieden.

Die Speicherklasse einer Variable hat außerdem Einfluß auf den Zeitpunkt der Initialisierung. Der Speicherklassenbezeichner wird vor den Variablentyp geschrieben. Wird bei einer Variablendefinition kein Speicherklassenbezeichner angegeben, so gilt für eine Definition außerhalb jeglicher Funktion standardmäßig die Speicherklasse **extern** (globale Variablen) und innerhalb einer Funktion die Speicherklasse **auto** (lokale Variablen).

#### 2.4.2.1 Die Speicherklasse **auto**

Automatische Variable sind nur in einem Block bekannt und existent, d.h. sie sind lokal. Der Variable wird Speicherplatz bei Eintritt in den Block vom Laufzeitsystem zugewiesen (auf dem Programmstack). Beim Verlassen des Blocks wird dieser Speicherplatz automatisch wieder freigegeben, die Variable ist nicht mehr verfügbar. Lebensdauer und Gültigkeit von Variablen der Speicherklasse **auto** sind somit auf den Block beschränkt, in dem sie definiert wurden.

Jede innerhalb einer Funktion bzw. innerhalb eines Blockes definierte Variable ist standardmäßig eine „auto-Variable“.

```
void watch_event () {
    int event_zaeher;      /* Speicherklasse auto */
    ...
}
```

#### 2.4.2.2 Die Speicherklasse **static**

Es gibt lokale und globale Variablen dieser Speicherklasse. Lokale statische Variablen sind lokal gültige Variablen jeweils eines Blockes und werden beim ersten Eintritt in den sie umgebenden Block initialisiert. Im Unterschied zu automatischen Variablen wird Ihnen jedoch Speicherplatz bereits zur Übersetzungszeit zugeordnet (d.h. zum Programmstart auf dem Heap). Das hat zur Folge, dass der Wert der Variable nach Verlassen des Blocks erhalten bleibt und bei einem späteren Wiedereintritt in den Block wieder genutzt werden könnte. Die Lebensdauer einer statischen Variablen entspricht somit der Zeit der Programmausführung, die Gültigkeit ist auf den Block ihrer Definition begrenzt.

Globale statische Variablen werden außerhalb jeglicher Funktionen in einem Quelltextfile definiert und haben in allen Funktionen dieses Quelltextfiles Gültigkeit. Sie können in anderen Quelltextfiles nicht verwendet werden.

```
void watch_event () {
    static int event_zaeher;
    ...
}
```

### 2.4.2.3 Die Speicherklasse `extern`

Externe Variablen existieren während der gesamten Programmausführung global, da ihnen während der Übersetzungszeit Speicherplatz zugeordnet wird (zum Programmstart auf dem Heap).

Mit Hilfe externer Variablen können Daten zwischen Funktionen eines oder mehrerer Quelltextfiles ausgetauscht werden. Eine externe Variable wird außerhalb jeder Funktion ohne Speicherklassenbezeichner definiert und ist somit in allen Funktionen dieses Quelltextfiles benutzbar (gültig). Soll nun von einem anderen Quelltextfile auf diese Variable zugegriffen werden, so ist sie in diesem Quelltextfile nochmals mit dem Speicherklassenbezeichner `extern` zu deklarieren.

Quelltextfile n:	Quelltextfile m:
<code>extern long time;</code>	<code>long time;</code>

### 2.4.2.4 Die Speicherklasse `register`

Registervariablen dienen der Effizienzsteigerung in Programmen mit zeitkritischen Funktionen. Der Compiler versucht, Registervariablen in den „schnellen“ Registern des Prozessors zu speichern, soweit derartige frei zur Verfügung stehen. Ist dies für eine Variable nicht möglich, wird sie wie eine Variable der Speicherklasse `auto` behandelt.

Als Registervariablen sind nur die Typen `int`, `unsigned`, `char` und Zeiger zulässig.

```
register int zaehler;  
register char zeichen;
```

## 2.4.3 Typkonvertierung

Es gibt Typkonvertierungen, die der Compiler (bzw. das Laufzeitsystem) automatisch vornimmt. Manchmal ist es jedoch notwendig, eine Typkonvertierung im Programm explizit anzuweisen.

### 2.4.3.1 automatische Typkonvertierung

Kommen Operanden unterschiedlichen Typs in Ausdrücken vor, so erfolgt (wenn möglich) automatisch eine Konvertierung in einen einheitlichen Typ.

Enthält der Ausdruck Operanden verschiedenen Datentyps, so gilt prinzipiell, dass der „kleinere“ Datentyp in den „größeren“ umgewandelt wird, wobei folgende Hierarchie der Datentypen zugrundegelegt wird:

`int`  $\mapsto$  `unsigned int`  $\mapsto$  `long`  $\mapsto$  `unsigned long`  $\mapsto$  `float`  $\mapsto$  `double`  $\mapsto$  `long double`



Auf der Wertebereichsbasis dieses Datentyps wird der Ausdruck berechnet, und er ist auch der Typ des Ergebniswertes. Es gibt zwei weitere Fälle, in denen eine automatische Typkonvertierung durchgeführt wird:

1. Bei Wertzuweisungen wird nach (!) der Berechnung des Ausdrucks rechts vom Zuweisungsoperator in den Zieldatentyp konvertiert.
2. Bei Funktionsaufrufen wird der Datentyp des aktuellen Parameters automatisch in den des formalen Parameters konvertiert. Beachte, dass es bei Typkonvertierungen in „kleinere“ Datentypen zu Wertverlusten und Wertverfälschungen kommen kann.

```
double d=1.23456789012345;
float f;
f=d;                                /* im günstigsten Fall f=1.234568 */

float berechne(float fp); /* Deklaration einer float-Funktion */
f=berechne(d);            /* d wird als float-Wert übergeben */
```

#### 2.4.3.2 erzwungene Typkonvertierung (cast-Operator)

Manchmal ist wünschenswert und sinnvoll, eine Typkonvertierung explizit zu erzwingen. Dies erreicht man dadurch, dass man vor den zu konvertierenden Operanden den Zieldatentypen in Klammern als sogenannten cast-Operator schreibt.

```
int a,b;
float g;
a=1; b=3;
g=a/b;          /* g wird 0.000000 zugewiesen */
g=(float)a/b;   /* g wird 0.333333 zugewiesen */
```

## Kapitel 3

# Ausdrücke

Ausdrücke beschreiben in C die Bildung von Werten eines bestimmten Typs in Abhängigkeit und von Operanden und ggf. Operatoren. Jeder Ausdruck liefert somit einen getypten Wert und kann geklammert als Operand Bestandteil eines noch komplexeren Ausdrucks sein. Durch die direkte Weiterverwendung des Wertes eines Ausdrucks in einem komplexeren Ausdruck können in C sehr kompakte Ausdrücke geschrieben werden, die vom Compiler zumeist auch in sehr effektiven Programmcode übersetzt werden. Die in C übliche kompakte Schreibweise reduziert vor allem für ungeübte Programmierer die Verständlichkeit des Programmtextes.

In Abhängigkeit von den Operatoren enthalten Ausdrücke einen oder mehrere Operanden, die allerdings „beliebig“ komplex sein können. Die Wertbildung bei komplexen Ausdrücken erfolgt unter Beachtung der Priorität der enthaltenen Operatoren bzw. entsprechend der Klammerung von „innen nach außen“. Die Verwendung eines Ausdrucks als Operand bedingt die Übereinstimmung (bzw. Konvertierbarkeit) des Typs des durch den Ausdruck gelieferten Wertes mit dem darauf angewendeten Operator und ggf. anderer Operanden.

Konstantenausdrücke sind Ausdrücke, die nur Konstanten miteinander verknüpfen und somit bereits zur Übersetzungszeit berechnet werden können. Identifikatoren als Operanden repräsentieren zumeist Variablen, d.h. sie liefern als Ausdruck den Wert der Variablen.

<code>(4911-49)</code>	<code>/* Konstantenausdruck vom Typ int */</code>
<code>(a*b)+0.22</code>	<code>/* Ausdruck vom Typ double */</code>
<code>fak(23-a)/2.123</code>	<code>/* Ausdruck mit Funktionsaufruf */</code>

# Kapitel 4

## Operatoren

Operatoren werden auf Operanden angewendet, wobei grundsätzlich mehrere Klassen von Operatoren zu unterscheiden sind, die im folgenden näher erklärt werden sollen.

### 4.1 Monadische Operatoren

Monadische Operatoren sind einstellige (unäre) Operatoren und haben die höchste Verarbeitungspriorität in komplexen Ausdrücken. Man unterscheidet zwischen den Referenzoperatoren, der logischen Negation, dem Inkrementoperator, dem Dekrementoperator und dem `sizeof`-Operator.

#### 4.1.1 Referenzoperatoren

Mit Hilfe von Referenzoperatoren kann die Adresse einer Variable ermittelt oder mit Hilfe der Adresse auf den Speicherplatz einer Variable zugegriffen werden. Referenzoperatoren werden in der Präfix-Notation mit den syntaktischen Zeichen `&` und `*` verwendet.

Der Operator `&` dient als Adressoperator und ermittelt die Adresse einer Variable, die auch als Strukturvariable ausgeprägt sein kann, oder einer Funktion (siehe Abschnitt 10.5, Seite 52). Der Operator `*` dient als Dereferenzierungsoperator. Der Operand wird als Adresse auf ein anderes Speicherobjekt interpretiert und ermöglicht hierüber den Zugriff auf dessen Wert („Dereferenzieren“).

```
z=&a;                /* liefert die Adresse der Variable a */
z=&vektor[7][8];     /* Adresse des Elementes vektor[7][8]
                     in einem zweidimensionalen Array */
*z;                 /* liefert den Wert der Variablen,
                     auf den z zeigt (hier: Wert von a) */
*(&a);              /* liefert den Wert der Variablen a */
```

Auf die Verwendung der Referenzoperatoren wird noch ausführlich im Abschnitt 6 eingegangen.

### 4.1.2 Arithmetische Negation

Die arithmetische Negation bildet den negativen Wert des Operanden. Hierfür wird das `'-'`-Zeichen verwendet.

```
-5;                /* klar!                */
a=-b;             /* auch klar, oder?    */
```

### 4.1.3 Logische Negation

Die logische Negation wendet man auf einen Ausdruck im Präfix an, und negiert damit den Wahrheitswert des Ausdrucks. Für die logische Negation verwendet man das `'!'`-Zeichen.

```
a=1;
(!a);                /* liefert den Wahrheitswert FALSE */
!(!a);              /* liefert den Wahrheitswert TRUE  */
```

Erinnern Sie sich: Enthält der Ausdruck mindestens ein Bit mit dem Wert 1, dann ist sein Wahrheitswert TRUE, sonst FALSE.

### 4.1.4 Inkrement- und Dekrementoperator

Inkrement- und Dekrementoperatoren können auf Variablen vom Typ `int`, `char` und auf Zeiger (Speicherplatzadressen) beliebigen Typs angewendet werden. Es ist Präfix als auch Postfix-Notation möglich, wobei unterschiedliche Semantiken zu beachten sind. Die Verwendung der Präfixform impliziert das In-/Dekrement des Wertes *vor* (!) der Berechnung des einbettenden Ausdrucks. Die Verwendung der Postfixform bewirkt hingegen das In-/Dekrement *nach* (!) Berechnung des Ausdrucks.

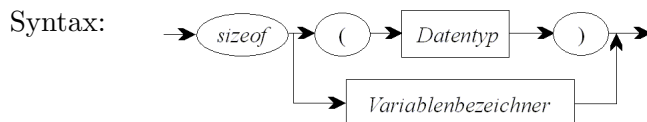
Als Inkrementoperator schreibt man `++`, als Dekrementoperator `--`.

```
++a; a++;           /* semantisch gleiche Operationen */
i=5; vektor[i++];   /* Zugriff auf Element 5 und i=i+1 */
i=5; vektor[++i];   /* i=i+1 und Zugriff auf Element 6 */
```

Die Anwendung der Operatoren auf Zeigervariablen in-/dekrementiert diese entsprechend der physischen Speicherlänge der referenzierten Variable (siehe auch Seite 36).

### 4.1.5 sizeof-Operator

Der sizeof-Operator unterstützt eine portable Programmierung, indem er den Zugriff auf eine implementations- d.h. plattformspezifische Eigenschaft von Variablen und Datentypen gewährt. Mit dem Operator `sizeof` kann die interne Größe des für einen Datentyp bzw. einer Variable benötigten Speichers in Bytes ermittelt werden.



```
float vektor[12];           /* ein float-Vektor */
a=sizeof (long);           /* a=4             */
b=sizeof vektor;           /* b=48            */
```

## 4.2 Multiple Operatoren

Multiple (binäre) Operatoren verbinden zwei Operanden zu einem neuen (komplexeren) Ausdruck und haben eine geringere Priorität als monadische Operatoren. Im folgenden sind die möglichen multiplen Operatoren nach fallender Priorität geordnet: arithmetische Operatoren, Verschiebeoperatoren, relationale Operatoren, Gleichheitsoperator, bitorientierte Operatoren, logische Operatoren und der bedingte Operator.

### 4.2.1 Arithmetische Operatoren

Mit Hilfe arithmetischer Operatoren können einfache Berechnungen durchgeführt werden.

*	Multiplikation der Operanden
/	Division der Operanden
%	Rest bei ganzzahliger Division (Modulo-Division)
+	Summe der Operanden
-	Differenz der Operanden

Beachte möglicherweise notwendige Typkonvertierungen bei der Anwendung der Operatoren.

```
c = 3*(5+a);           /* trivial */
```

Die Abarbeitung erfolgt von links nach rechts, wobei die Operatoren `*`, `/` und `%` eine höhere Priorität haben als die Summen- und Differenzbildung (die Klammern im vorangegangenen Beispiel sind also notwendig!). Zu beachten ist weiterhin, dass bei der Division ganzer

Zahlen der unter Umständen entstehende Rest abgeschnitten wird und keine Rundung stattfindet.

```
int i;  
i = 9/5;                /* ergibt 1 */
```

#### 4.2.2 Verschiebeoperatoren

Mit Hilfe des Verschiebeoperators werden die Bits des linken Operanden um die durch den zweiten Operanden angegebene Anzahl von Bitstellen verschoben. „Freiwerdende“ Bitstellen werden mit 0 aufgefüllt.

```
>>    Rechtsverschiebung  
<<    Linksverschiebung  
  
a = b >> 2;          /* Rechtsverschiebung um 2 Bit */  
c = b << n;          /* Linksverschiebung um n Bit  */
```

#### 4.2.3 Relationale Operatoren

Relationale Operatoren liefern einen Wahrheitswert, entsprechend der Erfüllung der bezeichneten Relation der Operanden.

```
<      kleiner  
>      größer  
<=     kleiner gleich  
>=     größer gleich  
  
a <= 5;  
b > c;          /* trivial */
```

Relationale Operatoren haben alle die gleiche Priorität und werden in komplexeren Ausdrücken in der Reihenfolge ihres Auftretens von links nach rechts ausgewertet.

#### 4.2.4 Gleichheitsoperatoren

Gleichheitsoperatoren liefern einen Wahrheitswert, entsprechend der Erfüllung der bezeichneten Gleichheitsrelation der Operanden.

```
==      gleich  
!=      ungleich
```

Verwende nie den Zuweisungsoperator, um die Gleichheit zweier Operanden zu testen!!!

```
if (n*(56-a) == x) ...
if (a != 4)         ...
if (a = 4)          ...      /* semantischer Fehler! */
```

#### 4.2.5 Bitorientierte Operatoren

Mit Hilfe bitorientierter Operatoren lassen sich die Operationen der boolschen Algebra auf Objekte vom Integertyp bzw. darin konvertierbare Objekte (z.B. char) umsetzen, wobei jeweils die Bits mit der gleichen Stellung im Objekt entsprechend der logischen Operation verknüpft werden.

<code>&amp;</code>	bitweises UND (Konjunktion)
<code>^</code>	bitweises Exklusiv-ODER (Anivalenz)
<code> </code>	bitweises ODER

In komplexeren Ausdrücken hat UND eine höhere Priorität als das Exklusiv- ODER, und dieses eine höhere Priorität als das ODER.

```
a = a & 0xFF00;      /* Rücksetzen des "unteren" Bytes */
a = a | b;           /* "Vereinigung" gesetzter Bits   */
```

#### 4.2.6 Logische Operatoren

Mit logischen Operatoren werden zwei Wahrheitswerte konjunktiv bzw. disjunktiv verknüpft. Das Ergebnis ist wiederum ein Wahrheitswert.

<code>&amp;&amp;</code>	logische Konjunktion (UND)
<code>  </code>	logische Disjunktion (ODER)

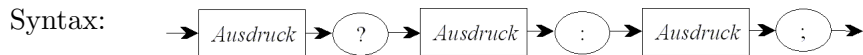
```
if ((a<=5) && (b>8)) ...
if ((a<3) || (a>10)) ...
```

Beachte den Unterschied zwischen bitorientierten und logischen Operatoren:

<code>2 &amp; 4</code>	<code>/* ergibt 0 */</code>
<code>2 &amp;&amp; 4</code>	<code>/* ergibt 1 */</code>

### 4.2.7 Bedingter Operator

Der bedingte Operator verknüpft 3 Operanden (Ausdrücke):



Wenn der erste Ausdruck den Wahrheitswert wahr ( $\neq 0$ ) liefert, wird als Ergebnis des bedingten Operators der zweite Ausdruck ausgewertet, ansonsten der dritte Ausdruck.

```
max = (a<b)? b:a;          /* Ermittlung des Maximums */
min = (a<b)? a:b;          /* Ermittlung des Minimums */
```

## 4.3 Zuweisungsoperatoren

Mit Hilfe von Zuweisungsoperatoren werden Variablen mit Werten belegt. Die Priorität von Zuweisungsoperatoren ist noch geringer als die des bedingten Operators. D.h. vor der Wertzuweisung werden die Operationen rechts vom Zuweisungsoperator ausgeführt. Folgende Zuweisungsoperatoren werden unterschieden:

=   +=   -=   \*=   /=   %=   >>=   <<=   &=   |=   ^=

Das Gleichheitszeichen ist der normale Zuweisungsoperator. In C werden zusätzliche Zuweisungsoperatoren für den (recht häufigen) Fall angeboten, dass der Ausdruck rechts vom Zuweisungsoperator aus zwei Operanden besteht, von denen ein Operand mit der Zielvariable identisch ist. In diesem Fall wird die jeweilige Operation mit dem Zuweisungsoperator ausgeführt.

```
a = b+c;          /* normale Zuweisung          */
a += b;           /* entspricht: a = a+b          */
a -= b;           /* entspricht: a = a-b          */
a *= 2;           /* entspricht: a = a*2          */
a /= 10;          /* entspricht: a = a/10         */
a %= b;           /* entspricht: a = a%b          */
a >>= 4;          /* entspricht: a = a<<4         */
a <<= 2;          /* entspricht: a = a>>2         */
a &= 0xFF00;      /* entspricht: a = a&0xFF00     */
a |= b;           /* entspricht: a = a|b          */
a ^= b;           /* entspricht: a = a^b          */
```



### 4.3.1 Mehrfachzuweisung

Die Zuweisung ist selbst ein Operator und liefert den Wert des Ausdrucks auf der rechten Seite. Daraus ergibt sich die Möglichkeit einer Mehrfachzuweisung, die dann von rechts nach links abgearbeitet wird.

```
z=a=7;          /* entspricht: a=7; z=a          */
a+=z*=2;        /* entspricht: z = z*2; a = a+z;
                  d.h. es sollte nun a==21 gelten */
```

## 4.4 Strukturoperatoren

Mit Hilfe von Strukturoperatoren kann man auf einzelne Elemente von Arrays und Strukturen zugreifen, wobei die Priorität von Strukturoperatoren noch höher ist, als die von monadischen Operatoren. Es werden 3 Strukturoperatoren unterschieden:

`[]`     `.`     `->`

Der Operator `[]` dient zur Definition von ein- bzw. mehrdimensionalen Vektoren, sowie zum Zugriff auf deren Elemente. Er wird im Abschnitt 7 auf Seite 38 noch genauer erklärt.

```
int vektor[20];    /* definiert einen Vektor mit
                    20 Zahlen vom Typ int          */
vektor[i];         /* Zugriff auf das i-te Element
                    das 1. Element hat Index 0 !!! */
```

Mit den Operatoren `.` und `->` wird der Zugriff auf Elemente von Strukturen und Vereinigungen realisiert (siehe Seite 43).

## 4.5 Kommaoperator

Der Kommaoperator dient zum Aufzählen mehrerer Ausdrücke und bildet gleichzeitig einen neuen (komplexeren) Ausdruck, dessen Wert der des letzten Ausdrucks ist. Die einzelnen Teilausdrücke werden von links nach rechts ausgewertet.

Anwendung findet der Kommaoperator vor allem bei der **for**-Anweisung, die im Abschnitt 5.5.3.3 auf Seite 32 beschrieben wird.

```
for (a=0, c=10; c<z; c++,a++) {
    ...
};
```

## 4.6 Prioritäten und Abarbeitungsreihenfolgen

Die Abarbeitungsreihenfolge (Assoziativität) von Operatoren in komplexeren Ausdrücken ist abhängig von deren Prioritäten untereinander. Die folgende Aufzählung ordnet die Operatoren noch einmal nach fallender Priorität:

- Strukturoperatoren:  
[] . ->
- Monadische Operatoren :  
\* & - ! ++ -- sizeof
- Multiple Operatoren:  
\* / % + - >> << < > <= >=  
== != & ^ | && ||
- Zuweisungsoperatoren:  
= += -= \*= /= %= >>= <<= &= ^= |=
- Kommaoperator :  
,

Stehen Operatoren gleicher Priorität nebeneinander, so gilt für Strukturoperatoren und multiple Operatoren eine Abarbeitungsreihenfolge von links nach rechts und für monadische und Zuweisungsoperatoren eine Abarbeitungsreihenfolge von rechts nach links.

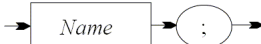
## Kapitel 5

# Anweisungen

Als Anweisung bezeichnet man jede ausführbare syntaktische Einheit eines C-Programms. Man unterscheidet einfache Anweisungen, strukturierte Anweisungen, Blöcke und sogar leere Anweisungen. Anweisungen können markiert werden.

### 5.1 Marken

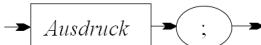
Vor jeder Anweisung können beliebig viele Marken stehen, auf die dann durch den angegebenen Identifikator bezug genommen werden kann. Marken sind jeweils nur in dem Funktionsblock gültig, in dem sie definiert wurden. Sie bestehen aus einem Identifikator, der die Marke eindeutig identifiziert, gefolgt von einem Doppelpunkt.

Syntax: 

Es zeugt von einem schlechten Programmierstil, Marken zu verwenden, und auf diese in Verbindung mit der Sprunganweisung `goto` zu verwenden (siehe auch Abschnitt [5.5.5](#))!!!

### 5.2 Ausdrucksanweisung

Jeder syntaktisch richtige Ausdruck wird zur Ausdrucksanweisung, wenn er durch ein Semikolon abgeschlossen ist. Beachte, dass das Semikolon nicht wie in Pascal als Trennzeichen zwischen jeweils 2 Anweisungen fungiert, sondern eine Anweisung abschließt.

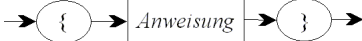
Syntax: 

```
z = 3*a+5;
einfuegen("Horst Mueller", 34, "Dresden");
```

### 5.3 Blöcke

Blöcke sind syntaktische Konstruktionen, die Vereinbarungen und Anweisungen enthalten. Ein Block wird nach außen syntaktisch als einzelne Anweisung betrachtet und kann an jeder Stelle stehen, wo vom Compiler eine Anweisung erwartet wird. Ein Block wird in geschweiften Klammern geschrieben und nicht (!) mit einem Semikolon beendet.

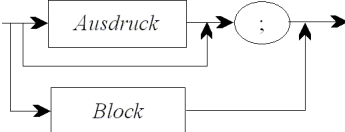
Variablendefinitionen innerhalb eines Blocks haben jeweils nur eine lokale Gültigkeit und gehören zur Speicherklasse `auto`.

Syntax: 

```
{z=3*a+5;
  einfuegen("Horst Mueller",34,"Dresden");
  z++;}
{int tmp;          /* lokale Variablendefinition */
  tmp=fak(i);}
```

### 5.4 Leere Anweisung

Die leere Anweisung hat keine Wirkung. Sie kann an Stellen verwendet werden, wo syntaktisch eine Anweisung verlangt, aber vom Programminhalt keine Aktion benötigt wird. Die Möglichkeit einer leeren Anweisung ergibt sich praktisch schon aus der Syntax von Anweisungen.

Syntax: 

```
for(i=12; i>0; printf("%d\n",i--))
    ;
```

### 5.5 Steuerstrukturen

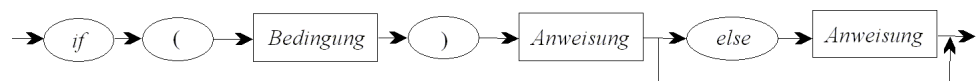
Mit Hilfe von Steuerstrukturen kann die ansonsten rein sequentielle Reihenfolge der Abarbeitung von Anweisungen in sinnvoller Weise erweitert (gesteuert) werden. Es werden

die bedingte Anweisung, die mehrfach bedingte Anweisung, die Zyklusanweisung, die Unterbrechungsanweisung und die Sprunganweisung unterschieden.

### 5.5.1 Bedingte Anweisung (if-Anweisung)

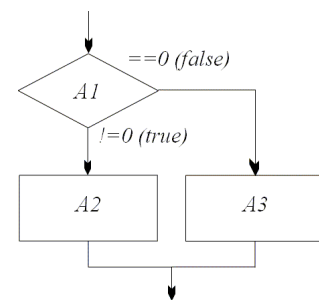
Mit Hilfe von bedingten Anweisungen wird eine Zweiwegentscheidung realisiert. Der in runden Klammern stehende Bedingungsteil (A1) wird als Ausdruck berechnet und als Wahrheitswert interpretiert. Ist dieser „wahr“ (d.h.  $\neq 0$ ) wird A2 ausgeführt, ansonsten (falls vorhanden) der Ausdruck A3.

Syntax:



```
if (A1)
    A2;
else
    A3;      /* optional */
```

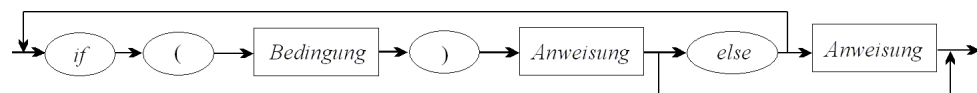
In A2 und A3 dürfen wiederum bedingte Anweisungen stehen. Wenn durch Klammerung (d.h. durch Bildung von Blöcken) nichts anderes definiert wird, gehört das `else` jeweils zum unmittelbar vorhergehenden `if`-Konstrukt. Der `else`-Zweig ist optional, d.h. er kann entfallen.



```
if (a<b)
    min=a;
else
    min=b;
```

Eine im `else`-Teil tiefgeschachtelte bedingte Anweisungen kann zum Beispiel zu Wertebereichstests verwendet werden.

Syntax:



```
if (gehalt<1000)
    gehalt+=50;
else if (gehalt<2000)
    gehalt+=40;
else if (gehalt<3000)
```

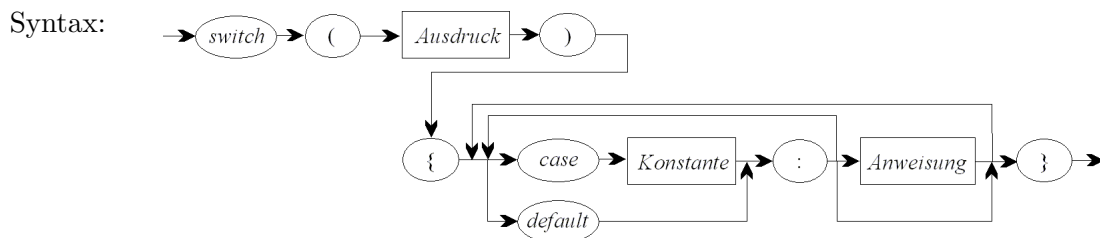
```

    gehalt+=30;
else if (gehalt<4000)
    gehalt+=20;
else
    gehalt+=10;

```

### 5.5.2 mehrfach bedingte Auswahl (switch-Anweisung)

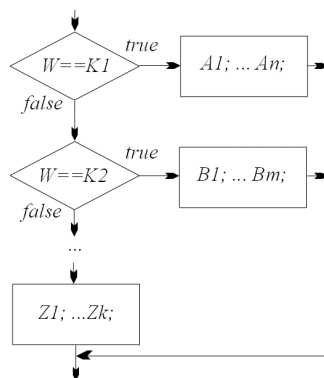
Zur Realisierung von Mehrwegentscheidungen in C steht die **switch**-Anweisung zur Verfügung, wobei nachfolgende Syntax einzuhalten ist:



```

switch (W) {
    case K1:
        A1; ...; An;           /* optional */
        break;                 /* optional */
    case K2:
        B1; ...; Bm;           /* optional */
    ...
    default:
        Z1; ...; Zk;           /* optional */
};

```



In Abhängigkeit von einem Ausdruck  $W$  wird eine von mehreren Anweisungenalternativen ausgewählt. Der Ausdruck  $W$  muss ein Integerwert sein (oder dahingehend konvertierbar). Jede Konstante  $K$  muss zur Übersetzungszeit berechenbar sein und einen anderen Wert repräsentieren. Es werden (falls vorhanden) die Anweisungen nach der **case**-Marke abgearbeitet, deren Konstantenausdruck dem Wert des Schalterausdrucks entspricht. Ist kein solcher Konstantenausdruck zu finden, wird falls vorhanden die Anweisung nach der **default**-Marke abgearbeitet oder die Abarbeitung nach dem **switch**-Ausdruck fortgesetzt.

Die Abarbeitung eines `case`-Zweiges kann mit einer Abbruchanweisung (`break`) beendet werden. Ist im `case`-Zweig keine Abbruchanweisung vorhanden, werden auch noch die Anweisungen der darauffolgenden `case`-Zweige (und des `default`-Zweiges) bis zum Auftreten einer Abbruchanweisung oder bis zum Ende der `switch`-Anweisung abgearbeitet.

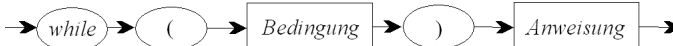
```
switch (c) {
    case 'A':
    case 'B':
    case 'C': puts("A, B oder C"); break;
    case '0':
    case '1': puts("0 oder 1"); break;
    default:  puts("unberücksichtigtes Zeichen");
}
```

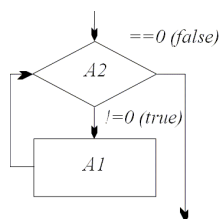
### 5.5.3 Zyklusanweisungen

In C gibt es 3 Sprachkonstruktionen für Zyklusanweisungen: die `while`-Anweisung, die `do`-Anweisung und die `for`-Anweisung.

#### 5.5.3.1 While-Anweisung

Mit Hilfe der `while`-Anweisung realisiert man sogenannte abweisende Zyklen. Das bedeutet, dass der Zyklustest `A2` jeweils vor der Abarbeitung der Zyklusoperationen durchgeführt wird. Ist dieser „falsch“ (d.h. `==0`) wird der Zyklus abgebrochen oder gar nicht erst begonnen. Ansonsten wird der Ausdruck `A1` solange berechnet (ausgeführt) bis `A2` den Wert 0 hat.

Syntax: 



```
while (A2)
    A1;
```

Sind mehrere Anweisungen in einem Zyklus auszuführen, so werden diese in einem Block geklammert, der dann für `A1` stehen kann.

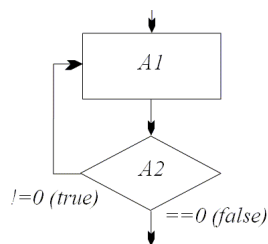
Beispiel zur iterativen Berechnung der Fakultät von 6:

```
long fak=1; int i=1;
while (i<=6) {
    fak*=i;
    i++;
}
```

### 5.5.3.2 Do-Anweisung

Die **do**-Anweisung realisiert einen nicht abweisenden (d.h. endgesteuerten) Zyklus, dessen Zyklusoperationen **A1** mindestens einmal ausgeführt werden. Der Zyklustest **A2** wird jeweils nach Abarbeitung der Zyklusoperationen durchgeführt. Ist dieser „falsch“ (d.h.  $==0$ ) wird der Zyklus abgebrochen, ansonsten wird er wiederholt.

Syntax:



```
do
    A1;
while (A2);
```

Sind mehrere Anweisungen in einem Zyklus auszuführen, so werden diese in einem Block geklammert, der dann für **A1** stehen kann.

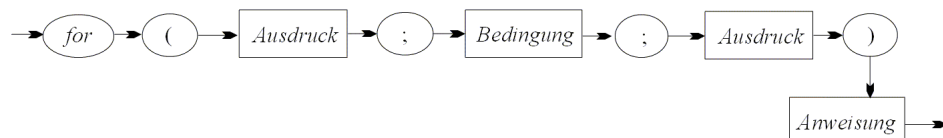
Beispiel zur iterativen Berechnung der Fakultät von 6:

```
long fak=1; int i=0;
do {
    ++i;
    fak*=i;
} while (i<6);
```

### 5.5.3.3 For-Anweisung

Die **for**-Anweisung dient wie die **while**-Anweisung zur Realisierung von „abweisenden Zyklen“ mit dem Zyklustest am Anfang. Sie wird relativ häufig verwendet, da sie die Lesbarkeit von Programmen dadurch erhöht, dass sie die wesentlichen zyklussteuernden Elemente (z.B. die Laufvariablen) in syntaktischen Zusammenhang mit dem Abbruchtest stellt.

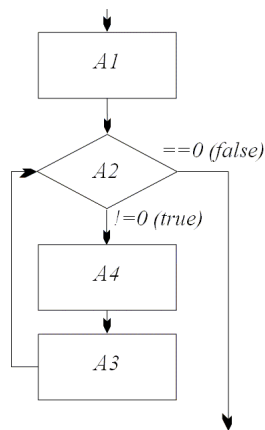
Syntax:



```
for(A1;A2;A3)
    A4;
```



Der Ausdruck **A1** dient der Initialisierung der Schleife. **A2** wird vor jeder Ausführung von **A4** berechnet, als Wahrheitswert interpretiert und bewirkt bei „wahr“ (d.h.  $\neq 0$ ) dessen Ausführung, sonst wird der Zyklus beendet bzw. gar nicht erst begonnen. Der Ausdruck **A3** dient i.a. zur Berechnung des nächsten Wertes der in **A1** initialisierten Variablen. Die Ausdrücke **A1**, **A2** und **A3** können jeweils weggelassen werden. Das entsprechende Semikolon muss jedoch geschrieben werden. Sind mehrere Anweisungen in einem Zyklus auszuführen, so werden diese in einem Block geklammert, der dann für Anweisung stehen kann.



```

for(A1;A2;A3) {
    A4; A5;
    ...
}
  
```

Die **for**-Anweisung ist in ihrem Steuerfluss äquivalent zu folgender **while**- Konstruktion:

```

A1;
while(A2) {
    A4; A5; A3;
}
  
```

Beispiel mit 3 semantisch gleichen **for**-Anweisungen zur Berechnung der Fakultät von 6:

```

fak=1;
for(i=1; i<=6; i++)          /* 1. Berechnung */
    fak*=i;

for(fak=1, i=1; i<=6; i++)    /* 2. Berechnung */
    fak*=i;


for(fak=i=1;;) {              /* 3. Berechnung */
    if (i>6) break;
    else fak*=i++;
}
  
```

#### 5.5.4 Modifikatoren

Es stehen zwei Anweisungen zur Verfügung, mit denen der Steuerfluss innerhalb von Zyklen bzw. in Auswahlanweisungen unter Beibehaltung der Grundstruktur geändert werden kann: die Fortsetzungsanweisung und die Abbruchanweisung.

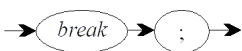
#### 5.5.4.1 Fortsetzungsanweisung

Diese Anweisung ist in **while**-, **do**- oder **for**-Schleifen erlaubt und bewirkt den vorzeitigen Übergang zum nächsten Zyklusdurchlauf, d.h. es wird der nächste Zyklustest ausgeführt und in Abhängigkeit davon der nächste Zyklus begonnen. In der **for**-Anweisung erfolgt vorher noch die Berechnung von **A3**.

Syntax: 

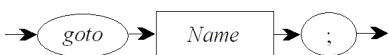
#### 5.5.4.2 Abbruchanweisung

Die Abbruchanweisung in einer **while**-, **do**- oder **for**- Schleife bewirkt den vorzeitigen und bedingungslosen Übergang zur nächsten Anweisung nach der Zyklusanweisung. In der **switch**-Anweisung ist sie notwendig zum Beenden eines **case**-Zweiges und bewirkt auch dort den bedingungslosen Übergang zur nächsten Anweisung nach der **switch**- Anweisung.

Syntax: 

#### 5.5.5 Sprunganweisung

Auf die Möglichkeit, jede Anweisung zu markieren, wurde bereits hingewiesen. Darauf aufbauend bietet C die Möglichkeit des unbedingten Sprunges zu einer mit einem Identifikator bezeichneten Marke des aktuellen Blocks.

Syntax: 

Im Unterschied zu den Unterbreuchungsanweisungen **break** und **continue**, die immer noch die algorithmischen Grundstrukturen beachten, kann mit **goto** die Strukturierung des Programmes derart willkürlich gestaltet werden, dass u.a. Lesbarkeit und Wartbarkeit der Quelltexte unmöglich werden. Sprunganweisungen sollten deshalb am besten überhaupt nicht verwendet werden.

# Kapitel 6

## Zeiger

Eines der wesentlichen Merkmale von C ist ein ausgezeichnetes und sehr effektives Zeigerkonzept. Als Zeiger bezeichnet man Adressvariablen, die die Adresse einer anderen Variable oder eines bestimmten Speicherbereiches als Wert beinhalten. In diesem Sinne „zeigt“ die Adressvariable dann auf die andere Variable bzw. auf den anderen Speicherbereich. Man sagt auch: die Adressvariable referenziert ein solches Speicherobjekt. Über die Adressvariable kann man dann auf das Speicherobjekt, auf das es zeigt, Bezug nehmen, d.h. darauf zugreifen. Diese Bezugnahme nennt man Dereferenzieren.

Die Besonderheit von C ist die Existenz einer Adressarithmetik, die in sinnvoller Weise das Rechnen mit Adressen erlaubt und eine sehr effiziente Programmierung für eine Vielzahl von Algorithmen ermöglicht. Besondere Bedeutung erlangen Zeiger bei der dynamischen Speicherverwaltung (siehe Vorlesung). Die hierfür notwendigen Bibliotheksfunktionen sind im Kapitel 12 ab Seite 58 beschrieben.

### 6.1 Zeigervariable

Eine Zeigervariable wird wie jede andere Variable typbezogen definiert, mit dem Unterschied, dass vor dem Namen ein `*` geschrieben wird.

```
int *zeigi;           /* Zeiger auf int-Variable */
long *zeigl;          /* Zeiger auf long-Variable */
char *zeigc;          /* Zeiger auf char-Variable */
int i; long l;
char c;               /* 3 normale Variablen      */
i=3; l=4711; c='x';
```

Zeigervariable bezeichnet den Zeiger selbst. Wird er in Ausdrücken verwendet, liefert er nur seinen Wert, d.h. die Adresse des referenzierten Speicherobjektes. Wird er hingegen mit dem Dereferenzierungsoperator `*` geschrieben (d.h. `*Zeigervariable`), dann liefert er

den Wert des referenzierten Speicherobjektes. Mit dem Adressoperator `&` lässt sich die Adresse eines Speicherobjektes ermitteln, die dann mit dem Zuweisungsoperator einer Zeigervariablen zugewiesen werden kann.

Fortsetzung des Beispiels:

```
zeigi=&i;           /* zeigi zeigt Variable i      */
zeigl=&l; zeigc=&c;  /* zeigl und zeigc auf l bzw. y */
zeigi=&l;           /* unzulässig, da Typkonflikt   */
l=*zeigi;          /* entspricht l=i              */
*zeigl=2*(*zeigi); /* entspricht l=2*i; d.h. 6    */
```

Es können Zeiger auf alle Variablentypen gebildet werden. Somit sind auch Zeiger auf Zeigervariablen erlaubt, wofür es durchaus sinnvollere Anwendungen gibt, als das folgende Beispiel glauben machen könnte:

Fortsetzung des Beispiels:

```
char **zeigz;      /* Zeiger auf Zeiger
                   für char-Variablen      */
zeigz=&zeigc;      /* zeigz zeigt auf Zeiger zeigc */
**zeigz=='x';      /* liefert "wahr", da c=='x'    */
*zeigc=**zeigz+1; /* entspricht c=c+1; d.h. c='y'  */
```

Beachte: Von Variablen der Speicherklasse `register` (siehe Abschnitt 2.4.2.4 auf Seite 16) kann keine Adresse und somit kein Zeiger auf eine solche Variable gebildet werden.

## 6.2 Adressarithmetik

Die Adressarithmetik gehört zu den Stärken von C und ermöglicht es, in sinnvoller Weise mit Adressen zu rechnen. Jede arithmetische Adressveränderung eines Zeigers basiert grundsätzlich auf dem bei der Definition der Adressvariablen angegebenen Typ. Über diesen Typ ist die Länge des referenzierten Speicherobjektes ermittelbar. Eine Addition bzw. Subtraktion eines Zeigers um einen Wert `n` hat eine Erhöhung bzw. Reduzierung der Adresse um das `n`-fache der Länge des referenzierten Speicherobjektes zur Folge.

```
long *zeigc, vektor[30];
zeigc=&vektor[0]; /* Initialisierung des Zeigers */
zeigc++;         /* zeigc zeigt auf nächstes Element
                  d.h. Erhöhung des Zeigers um 4 */
zeigc+=7;        /* zeigc zeigt auf das um 7 long-
                  Objekte entfernte Objekt */
zeigc-=8;        /* zeigc zeigt nun wieder auf das
                  ursprüngliche long-Objekt */
```

Zulässige Operatoren der Adressarithmetik sind:

- Addition einer ganzen Zahl zum Zeigerwert
- Subtraktion einer ganzen Zahl vom Zeigerwert
- Subtraktion zweier Zeiger (ergibt den Abstand im Hauptspeicher)
- Vergleich von zwei Zeigerwerten

Beispiel für die Subtraktion zweier Zeiger:

```
long *c,*x,y;  
c=&y;  
x=c+2;  
c-x;                                /* ergibt 2, und die Warnung,  
                                   dass der Wert nicht genutzt wird */
```

Adressarithmetik mit Zeigern bietet vielfältige Möglichkeiten, ist aber auch immer wieder eine Quelle von Programmierfehlern, da zum Übersetzungszeitpunkt i.a. nicht geprüft werden kann, ob das jeweils referenzierte Objekt auch dem in der Zeigerdefinition angegebenen Typ entspricht. Das erfordert vom Programmierer selbst hohe Sorgfalt bei der Verwendung von Zeigern.

Sinnvolle Adressarithmetik steht in engem Zusammenhang mit der Möglichkeit, Vektoren und Pointer gleichartig zu behandeln, was in einem eigenen Abschnitt (siehe Seite 36) noch ausführlich besprochen wird.

### 6.3 Initialisierung von Zeigern

Die Initialisierung von Zeigervariablen folgt prinzipiell den gleichen Regeln, wie die Initialisierung von Variablen allgemein. Allerdings stehen für die Initialisierung nur die Adressen bereits definierter Objekte zu Verfügung.

Eine Sonderstellung nimmt der für Zeigervariablen gültige Wert 0 (geschrieben auch NULL) ein. Da dieser Wert niemals die Adresse eines gültigen Speicherobjektes sein kann, gilt diese Wertbelegung als „leere Adresse“ und wird oft mit einer zusätzlichen Semantik belegt (z.B. „Funktion fehlgeschlagen“, „letztes Element einer dynamischen Liste erreicht“, ...).

```
int i;  
int *p1=&i;                /* p1 zeigt auf i          */  
int *p2=NULL;              /* p2 mit NULL initialisiert */  
int *p3=&f;                /* nicht zulässig, f unbekannt */  
int f;
```

# Kapitel 7

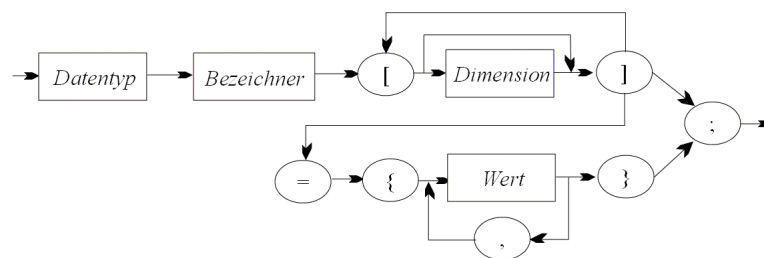
## Vektoren

Mit Vektoren fasst man mehrere Werte gleichen Typs zu einem einzigen Speicherobjekt zusammen.

### 7.1 Definition und Handhabung von Vektoren

Man definiert einen Vektor, indem man Typ und Anzahl der Komponenten, sowie einen Identifikator (d.h. Variablennamen) festlegt. Die Anzahl der Komponenten wird in eckigen Klammern durch Angabe eines Konstantenausdruckes von Typ `int` festgelegt oder vom Compiler implizit aus der Initialisierung entnommen, falls nur die leeren Klammern `[]` angegeben wurden. Die Anzahl der Vektorelemente ist statisch festgelegt und kann zur Programmlaufzeit nicht geändert werden. Die Initialisierung erfolgt durch eine in geschweiften Klammern anzugebende Werteliste.

Syntax:



```
int zahlen[100];           /* Vektor von 100 int-Werten    */
char b[20];                /* Vektor mit 20 char-Werten   */
int a[]={11,13,17,19};     /* Vektor mit 4 int-Werten incl.
                           Initialisierung */
int b[];                   /* nur ein Zeiger wird angelegt */
```

Im Gegensatz zu anderen Programmiersprachen sind Operationen mit einem Vektor als Ganzem nicht möglich, sondern nur mit den einzelnen Elementen. Der Zugriff auf einzelne Vektorelemente ist mit Hilfe des Operators `[]` möglich, wobei der anzugebende Ausdruck in den eckigen Klammern den Index des Vektorelementes bestimmt und vom Typ `int` sein muss. Zu beachten ist dabei, dass das 1. Vektorelement den Index 0 besitzt.

Beachte: Der Zugriff auf nicht vorhandene Vektorelemente (z.B. `zahlen[-1]`) oder `b[20]`) wird von der C-Laufzeitumgebung nicht verhindert und ist ein häufiger Fehler (nicht nur bei Anfängern).

Fortsetzung des Beispiels:

```
a[0]==11;          /* "wahr" */
a[2]==17;          /* "wahr" */
a[3]=256;          /* Wertzuweisung für 4. Element */
a[4]=257;          /* Fehler: nicht vorhandenes
                    Element! Keine Fehlermeldung! */
```

Neben eindimensionalen können in C auch mehrdimensionale Vektoren definiert und benutzt werden, wobei die oben angegebene Syntax gilt. Bei der Initialisierung mehrdimensionaler Vektoren ändert sich die letzte Dimension beim Übergang von einem Element zum nächsten am schnellsten.

```
float matrix[5][7];      /* Vektor mit 5x7 Elementen */
int i[][] = {{1,2,3},
             {4,5,6}};   /* Vektor mit 2x3 Elementen */
i[0][0]==1; i[0][1]==2;
i[0][2]==3; i[1][0]==4;
i[1][1]==5; i[1][2]==6; /* wahre Aussagen */
```

## 7.2 Äquivalenz: Zeiger-Vektoren

Es wurde bereits auf die Äquivalenz von Zeigern und Vektoren hingewiesen. Diese ist vor allem in der Organisationsform von Vektoren begründet. D.h. der Identifikator eines Vektors ist in C gleichzeitig eine konstante Zeigervariable, die auf das erste Vektorelement verweist. Sie kann als Zeigervariable verwendet werden, mit der Einschränkung, dass ihr kein neuer Wert zugeordnet werden kann. Sie wird immer auf den Vektor (d.h. dessen erstes Element) zeigen, für den sie deklariert wurde.

```
int zahlen[50];          /* ein Vektor mit 50 int-Werten */
int *z;                  /* Zeiger auf int-Wert */
z=zahlen;                /* ist identisch mit z=&zahlen[0] */
*z=5;                   /* ist nun identisch mit
                        *zahlen=5 oder zahlen[0]=5 */
```

Die Adressarithmetik wurde bereits besprochen. Auf ihr basiert die Äquivalenz von Zeigern und Vektoren. Da Vektorvariablen als Zeiger organisiert sind, kann der Programmierer jeden Zugriff auf ein Vektorelement `vektor[i]` in einen semantisch äquivalenten Adressausdruck `*(vektor+i)` umwandeln.

Fortsetzung des Beispiels:

```
int i;
i=*zahlen;      /* identisch mit i=zahlen[0] oder i=*z    */
i=(z+5);        /* identisch mit i=zahlen[5] oder i=z[5] */
*(z+3)=9;       /* identisch mit zahlen[3]=9 oder
                  (zahlen+3)=9                               */
```

Beachte die Kammersetzung im vorangegangenen Beispiel. Der Dereferenzierungsoperator muss vor der Klammer stehen, da er stärker bindet als die Adressoperationen!

## 7.3 Äquivalenz: Zeiger-Zeichenketten

Es existiert in C kein Datentyp für Zeichenketten (wie z.B. „string“ in Pascal). In C werden Zeichenketten als Vektoren von Zeichen organisiert, die durch das Standardendezeichen `'\0'` abgeschlossen werden, d.h. eine Zeichenkette der Länge  $n$  belegt  $n+1$  Vektorelemente vom Typ `char`. Der Zugriff auf Zeichenketten über Zeiger ist äquivalent zum Zugriff auf Vektoren, wenn ein Zeiger auf die Zeichenkette definiert wird. Das Endekennzeichen wird vom Compiler bei der Initialisierung des Speicherbereiches für die Zeichenkette automatisch mit generiert.

```
char *zk;        /* Zeiger auf char oder Zeichenkette */
zk="Vorlesung C"; /* Initialisierung des Zeigers                */
if (zk[0]=='V')   ... /* "wahr", identisch zu *zk=='V'                */
if (zk[11]=='\0') ... /* "wahr", identisch zu *(zk+11)=='\0'          */
```

## 7.4 Vektoren von Zeigern und Zeichenketten

Vektoren wurden definiert als Zusammenfassung mehrere Objekte gleichen Typs. Diese Definition schließt Vektoren von Zeigern ein, die in C wie folgt angelegt werden.

```
char *vektor[20]; /* uninitialisierter Vektor für
                  20 Zeiger auf char-Objekte          */
...
c=*vektor[3];     /* Zugriff auf das char-Objekt auf
                  das der 4. Vektoreintrag verweist  */
printf(vektor[1]); /* Ausgabe des zweiten Vektorelementes */
```



Vektoren von Zeigern werden in der Praxis vor allem für Verweise auf Zeichenketten verwendet, da diese i.a. unterschiedliche Länge haben. Ein Programm, das mehrsprachige Ausgaben unterstützt, könnte z.B. die auszugebenden Zeichenketten jeweils in Vektoren mit Verweisen auf Zeichenketten organisieren:

```
#define ENGLISCH      0
#define DEUTSCH       1
#define POLNISCH      2
...
char *text1[] = {
    "Please insert your credit card now!",
    "Bitte geben Sie jetzt Ihre Kreditkarte ein!",
    "Prosze wlozyc teraz karte kredytowa!",
    ...
}
...
sprache=DEUTSCH;          /* je nach Wunsch */
...
printf("%s\n",text1[sprache]); /* gibt text1[1] aus */
...
```

Die Zeichenketten selbst können mit keinem Operator als Objekt, d.h. als ein Ganzes, behandelt werden. Mit dem Zuweisungsoperator kopiert man nur den Zeigern auf die jeweilige Zeichenkette.

```
char *text="abcd",*poi;    /* ein initialisierter und
                           ein leerer Zeiger */
poi=text;                 /* nur der Zeiger wird kopiert,
                           d.h. poi und text zeigen auf
                           dieselbe Zeichenkette */
poi[0]='x';               /* Änderung des ersten Zeichens,
                           d.h. text zeigt jetzt auf die
                           Zeichenkette "xbcd" */
```

Natürlich stellt Ihnen die C-Bibliothek viele hilfreiche Funktionen für die Arbeit mit Zeichenketten zur Verfügung (siehe Abschnitt [12](#) ab Seite [58](#)). Mit diesen Funktionen ist dann auch das Kopieren von Zeichenketten möglich.

# Strukturen

Die Definition von Strukturen erfolgt unter Verwendung des Schlüsselwortes **struct**. Nach dem Schlüsselwort kann ein Name folgen, der Strukturtypname genannt wird und nachfolgend zum Definieren von Strukturvariablen dieses Typs benutzt werden kann. Die Elemente (auch Komponenten genannt) einer Struktur werden durch Variablenvereinbarungen angegeben und durch geschweifte Klammern zusammengefasst. Die den geschweiften Klammern folgenden Identifikatoren bezeichnen die Strukturvariablen.

The diagram illustrates the states and transitions for the grammar rules of the C language. The states are represented by ovals and the transitions by arrows. The grammar rules are as follows:

- $struct \{ Datentyp Name ; \}$
- $enum \{ Name ; \}$
- $union \{ Datentyp Name ; \}$
- $typedef Datentyp Name ;$

The states and transitions are as follows:

- States (Ovals):**  $struct$ ,  $\{$ ,  $\}$ ,  $;$ ,  $Datentyp$ ,  $;$ ,  $,$ .
- Transitions (Arrows):**
  - $struct \rightarrow \{$
  - $\{ \rightarrow Datentyp$
  - $Datentyp \rightarrow Name$
  - $Name \rightarrow ;$
  - $; \rightarrow \}$
  - $\} \rightarrow struct$
  - $\} \rightarrow enum$
  - $\} \rightarrow union$
  - $enum \rightarrow \{$
  - $union \rightarrow \{$
  - $typedef \rightarrow Datentyp$
  - $Datentyp \rightarrow Name$
  - $Name \rightarrow ;$
  - $enum \rightarrow Name$
  - $union \rightarrow Name$
  - $Name \rightarrow ,$
  - $,$  has a self-loop.

Von einer Strukturdeklaration spricht man, wenn keine Strukturvariablen angegeben werden. In einer Strukturdeklaration wird lediglich die Struktur mit ihren Elementen (Komponenten) unter einem Namen definiert, ohne Variablen anzugeben, für die der Compiler (und dann die Laufzeitumgebung) Speicherplatz reservieren müsste.

42

```
};
```

Unter dem Strukturtypnamen `datum` kann nun auf diese Strukturvereinbarung Bezug genommen werden.

## 8.2 Definition von Strukturvariablen

Um Strukturvariablen zu definieren, muss man nur die Identifikatoren der Strukturvariablen zur Strukturvereinbarung hinzufügen.

```
struct datum {                /* Def. zweier Strukturvariablen */
    int tag;
    char *monat;
    int jahr;
} dat1, dat2;
```

Ist die Struktur bereits vereinbart worden, kann später im Programm bei der Definition weiterer Strukturvariablen unter Nutzung des jeweiligen Identifikators darauf Bezug genommen werden.

Fortsetzung des Beispiels:

```
struct datum dat3, dat4;    /* Strukturvariablen */
```

Noch einfacher ist es, wenn man eine Struktur als neuen Datentyp deklariert. Hierzu wurde bereits das Schlüsselwort `typedef` eingeführt und dessen Verwendung demonstriert. Damit kann man sich im Programm komplexe Datentypen deklarieren, und dann wie gewohnt Variablen dieser neuen Typen definieren.

```
typedef struct {              /* neuer komplexer Datentyp */
    char *name;
    char *anschrift;
    struct datum geburtstag;
} person;
person x, y, z;               /* Variablen dieses neuen Typs */
```

## 8.3 Nutzung von Strukturen

In der gleichen Weise wie für Vektoren gilt auch für Strukturen, dass C nur wenige Operatoren bereitstellt, die mit Strukturen als ganzes Objekt umgehen können. Hierzu gehört der Zuweisungsoperator für Wertzuweisungen zwischen Strukturvariablen gleichen Typs, und der Adressoperator, mit dem man die Adresse einer Strukturvariable ermitteln kann.

```

person p,q,*z;          /* 2 Strukturvariablen und ein Zeiger */
...
z=&p;                   /* z zeigt auf die Strukturvariable  */
q=p;                   /* q hat nun die gleichen Werte wie p */

```

Operationen sind ansonsten nur auf einzelne Elemente (Komponenten) der Struktur zulässig, auf die über ihren Namen Bezug genommen wird. Eine solche Bezugnahme wird über die Strukturoperatoren `.` bzw. `->` realisiert, wobei der Punkt für Strukturvariablen und der Pfeil zur Auflösung von Zeigern auf Strukturen verwendet wird.

```

char *mo[] = { "Januar", "Februar", ... }
struct datum {
    int tag;
    char *monat;
    int jahr;
} dat, *poi;             /* Strukturvariable und Zeiger */

dat.tag=13;
dat.monat=m[8];
dat.jahr=1965;           /* Wertzuweisung für Elemente */
poi=&dat;
poi->tag=14;              /* identisch mit (*poi).tag=14 */
poi->monat=m[9];
poi->jahr=1966;           /* ändert die Wertebelegung!! */

```

Die Strukturoperatoren sind verkettbar, so dass auch auf komplexere Strukturen zugegriffen werden kann:

```

person p;                /* Strukturvariable */
p.geburtstag.jahr=1965;  /* Wertzuweisung */

```

## 8.4 Initialisierung von Strukturen

Die Initialisierung von Strukturen erfolgt analog zur Initialisierung von Vektoren durch eine in geschweiften Klammern eingeschlossene Liste von Werten. Bei geschachtelten Strukturen sind die geschweiften Klammern ebenfalls entsprechend zu schachteln.

```

struct datum dat=        /* Strukturvariable */
    {7,"Februar",1922};  /* Initialisierung */
person p[]={              /* Vektor mit 2 Personen */
    {"A. Beck", "HTW Dresden",
     {27,"Dezember", 1955}},

```

```

        {"K. Bruns", "HTW Dresden",
         {13, "September", 1965}}

};                                     /* Initialisierung      */

```

## 8.5 Rekursive und dynamische Strukturen

Wie aus den bereits angegebenen Beispielen sichtbar wird, können Strukturen wiederum Strukturen als Elemente beinhalten. Dies gilt jedoch nicht für Strukturen gleichen Typs. Es ist also (aus ersichtlichen Gründen) nicht möglich, dass eine Komponente der Struktur von gleichen Typ ist, wie die Struktur selbst. Ein solcher rekursiver Bezug ist nur über Zeiger möglich und spielt eine wesentliche Rolle beim Aufbau dynamischer Datenstrukturen wie Listen oder Bäumen.

```

struct person{
    char *name;
    char *anschrift;
    struct datum geburtstag;
    struct person *next;
} x, y, z;                               /* 3 Strukturvariablen */
x.next=&y;                                /* Verketteten der Personen */
y.next=&z;
z.next=&x;
(x.next)->geburtstag.tag=12;             /* Zugriff auf y über x */
y.geburtstag.tag=12;                     /* identischer Befehl   */

```

## 8.6 Bitfelder

Unter anderem Bitfelder haben C fälschlicherweise den Ruf eingebracht, eine „assembler-nahe“ Programmiersprache zu sein.

Im folgenden Beispiel wird eine Variable `b` definiert, die als Bitfeld 3 Komponenten enthält. Die Komponenten `flag` und `paritaet` sind jeweils 1 Bit groß und haben somit als gültige Zustände nur die Werte 0 und 1. Die Komponente `zust` umfasst 4 Bit und somit einen Wertebereich von 0 bis 15.

```

struct {
    unsigned flag      :1;
    unsigned paritaet  :1;
    unsigned zust      :4;
} b;

```

Auf Bitfelder wird in der für Strukturen üblichen Form mit dem Punktoperator zugegriffen. Der Wert eines Bitfeldes wird als vorzeichenlose ganze Zahl interpretiert und kann in Ausdrücken (z.B. als Testvariable) verwendet werden.

```

if (b.paritaet==1) {
    ...
} else {
    ...
};
b.zustand=8;

```

## 8.7 Aufzählungstypen

Der Aufzählungstyp lehnt sich an die syntaktische Konstruktion von Strukturen an und erlaubt (in Analogie zur Programmiersprache Pascal) die explizite Aufführung von Werten. Der Aufzählungstyp ist durch das Schlüsselwort `enum` gekennzeichnet. Die aufzuzählenden Werte werden in geschweifte Klammern geschrieben und durch Komma voneinander getrennt. Intern wird jeder Wert von 0 beginnend auf eine positive ganze Zahl abgebildet. Innerhalb des Gültigkeitsbereiches der Aufzählungsvariable dürfen die Werte der Aufzählungsvariable nicht anderweitig im Programm z.B. als Variablennamen verwendet werden.

```

enum Tag {
    Montag,           /* ist intern = 0 */
    Dienstag,        /* ist intern = 1 */
    Mittwoch,         /* ist intern = 2 */
    Donnerstag,       /* ist intern = 3 */
    Freitag,          /* ist intern = 4 */
    Sonnabend,        /* ist intern = 5 */
    Sonntag };        /* ist intern = 6 */
enum Tag t;          /* Variable von "Typ" Tag */
t=Mittwoch;          /* gültige Wertzuweisung */
if (t==Donnerstag) {;}; /* gültiger Test */
int Freitag;         /* Fehler: Neudefinition */

```

Der interne Wert der Elemente einer Aufzählung kann bei deren Vereinbarung durch explizite Wertzuweisung beeinflusst werden. Das Folgeelement in der Aufzählung enthält dann auch den unmittelbar folgenden internen int-Wert. Doppelbelegungen sind möglich.

```

enum Ampelfarbe {
    rot=1,            /* ist intern = 1 */
    gelb=4,           /* ist intern = 4 */
    gruen };          /* ist intern = 5 */

```

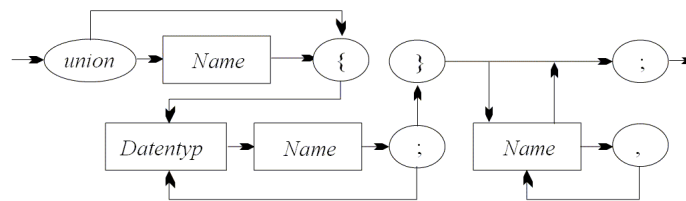
## Kapitel 9

# Vereinigungen

Mit Hilfe von Vereinigungen können ebenfalls mehrere Komponenten möglicherweise verschiedenen Datentyps zu einem einzigen Objekt zusammengefasst werden.

Zwischen Strukturen und Vereinigungen besteht syntaktisch eine sehr enge Verwandtschaft. Anstelle des Schlüsselwortes `struct` wird bei Vereinigungen das Schlüsselwort `union` verwendet.

Syntax:



```
union zahlen {          /* Definition einer Vereinigung */
    int      i;
    long     l;
    float    f;
    double   d;
} varu;
```

Semantisch besteht jedoch ein entscheidender Unterschied zwischen Strukturen und Vereinigungen. Im Gegensatz zu Strukturen wird allen Komponenten einer Vereinigung derselbe Speicherplatz (dieselbe Adresse) in der Größe der größten Komponente reserviert. Dadurch besteht die Möglichkeit, einen Speicherbereich mit Variablen verschiedenen Typs zu interpretieren. Da die interne Repräsentation von Variablen i.a. von Plattform zu Plattform variiert, können Vereinigungen die Portabilität von Programmen behindern.

Der Speicherplatz für Vereinigungen wird so bereitgestellt, dass die größte in ihr enthaltene Komponente aufgenommen werden kann.

Fortsetzung des Beispiels:

```
varu.i=67;      /* nutzt Variable als int-Objekt */
varu.f=234.56;  /* nutzt Variable als float-Objekt */
varu.i==67;     /* liefert "falsch", da der
                  int-Wert überschrieben wurde */
```



# Kapitel 10

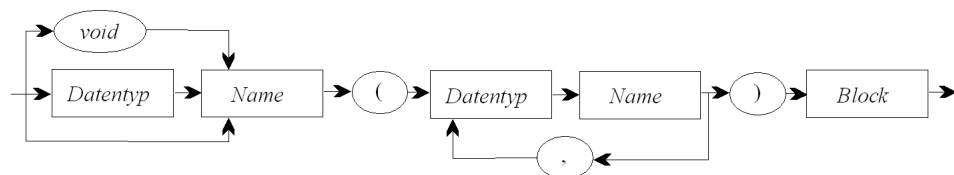
## Funktionen

Die Softwaretechnologie lehrt, dass komplexere Problemstellungen in kleinere überschaubare Teilaufgaben zu zerlegen sind. Solche Teilaufgaben werden in C als eigene Funktionen geschrieben. Das erhöht zum einen die Lesbarkeit und Überschaubarkeit von Programmen und ermöglicht zum anderen die Wiederverwendbarkeit von Programmteilen. Jede Funktion kann in Programmbibliotheken abgelegt und somit in anderen Programmen wieder eingebunden werden. Jedem C-Compiler werden heute in Form von Bibliotheken bereits eine Vielzahl von allgemein wiederverwendbaren Funktionen mitgegeben. Hierüber lassen sich ohne größeren Aufwand z.B. komplizierte numerisch mathematische Verfahren oder komplexe Operationen für Windows-Oberflächen realisieren.

### 10.1 Definition von Funktionen

Funktionen sollten zunächst definiert oder zumindest deklariert werden, bevor man sie verwendet. Zur Definition von Funktionen ist laut ANSI folgende Syntax einzuhalten, wobei ein Block bereits im Abschnitt 5.3 auf Seite 28 erklärt wurde:

Syntax:



Wenn kein Speicherklassenbezeichner vor dem Funktionstyp angegeben wird, sind Funktionen immer **extern**. Zulässig ist noch der Speicherklassenbezeichner **static**, der bewirkt, dass der Name der Funktion lediglich im aktuellen Quelltextfile bekannt ist.

Wird ein Funktionstyp angegeben, so spezifiziert er den Typ des Wertes, den die Funktion nach ihrer Ausführung als Wert zurückliefert. Ohne die Angabe des Funktionstyps wird **int**

als Rückgabewert angenommen. Soll von der Funktion kein Wert zurückgegeben werden, so ist für Funktionstyp `void` anzugeben. Die Wahl des Funktionsnamen kann bis auf eine Ausnahme frei erfolgen, wobei das gleiche Regelwerk wie bei Variablennamen gilt.

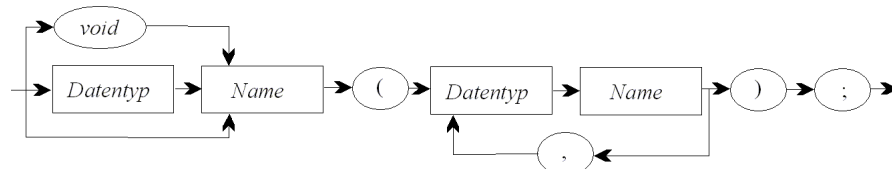
Genau eine Funktion des Programmes muss den Funktionsnamen `main` tragen. Dieser Funktion wird nach Aufruf des Programms die Steuerung durch das Betriebssystem übertragen.

```
void nichts() {};           /* Minimalfunktion: parameterlos
                           und macht nichts          */
int fkt(char a, long b) {   /* 2 Parameter          */
    ...; return(0);         /* liefert 0 als Ergebniswert */
}
void main() {               /* "Hauptfunktion" mit
    ...                     Programmsteuerung      */
}
```

## 10.2 Deklaration von Funktionen (Funktionsprototypen)

Funktionen, die in anderen Quelltextfiles definiert wurden und als Bibliotheksfunktionen vom eigenen Programm genutzt werden sollen, müssen vor ihrer Verwendung deklariert werden. Dies geschieht zumeist in sogenannten Headerfiles (z.B. für die Standardbibliothek `stdlib.h`, siehe Seite 66) und ermöglicht es dem Compiler, den korrekten Funktionsaufruf zu überprüfen. Bei der Funktionsdeklaration wird der Funktionskörper der Definition durch ein Semikolon ersetzt.

Syntax:

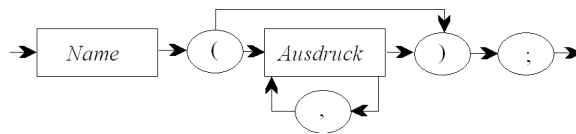


```
int fkt(char a, long b);
```

## 10.3 Funktionsaufruf und Parameterübermittlung

Funktionen werden über ihren Identifikator aufgerufen. In runden Klammern kann eine Liste aktueller Parameter (Argumente) stehen, die der bei der Funktionsdefinition bestimmten Liste formaler Parameter entsprechen muss. Dies sind Ausdrücke, deren Werte beim Funktionsaufruf berechnet und an die Funktion übergeben werden.

Syntax:



Ein Funktionsaufruf kann insgesamt selbst wieder als Ausdruck angesehen werden, da er (falls in der Definition vereinbart) einen Wert liefert, der dann weiter verwendet werden kann (aber nicht muss!).

```
nichts();           /* Aufruf der Funktion nichts      */
fak(4);             /* Aufruf von fak mit Parameter 4 */
a=fak(5);           /* Aufruf von fak mit Parameter 5
                   und Zuweisung Ergebnis an a      */
b=6; a=fak(b);      /* Aufruf von fak mit b=6         */
```

Bei einem Funktionsaufruf wird jeder aktuelle Parameter auf den Speicherplatz eines formalen Parameters kopiert, wobei diese Zuordnung über die jeweilige Stellung in der Parameterliste erfolgt. Bei Nichtübereinstimmung der Typen zwischen aktuellem und formalen Parameter gelten die üblichen (automatisch angewandten) Typkonvertierungsregeln. Ist der Typkonflikt hierüber nicht aufzuheben, gilt der Funktionsaufruf als fehlerhaft und es wird zum Übersetzungszeitpunkt eine Fehlermeldung ausgegeben. Die aufgerufene Funktion arbeitet also mit einer Wertekopie der Parameter und kann somit den Originalwert nicht ändern. Man nennt diese Methode „call by value“. Die aktuellen Parameter können Ausdrücke sein, die beim Funktionsaufruf ausgewertet werden (von rechts nach links!).

Will man jedoch die Werte der Variablen in der aktuellen Parameterliste ändern, so muss man der Funktion die Variablenadresse als Parameter übermitteln. Beachte, dass Vektoren als Funktionsparameter automatisch als Zeiger behandelt werden. D.h., es wird der Funktion die Anfangsadresse des Vektors übergeben, ohne eine Kopie des Vektors anzulegen. Damit ist jede Änderung des Vektorinhaltes auch außerhalb der Funktion sichtbar. Diese Methode nennt man „call by referenz“.

```
int plus1(int x, int y) {           /* Funktion mit
    return x+=y;                    2 int-Parametern */
}
int plus2(int *x, int *y) {         /* Funktion mit
    return (*x)+=(*y);              2 Zeiger-Parametern */
}
...
int a,b,c[5];
a=b=1;
c[0]=plus1(a,b);                   /* c[0]=2 aber keine Wert-
                                   Änderung von a und b */
c[0]=plus2(&a,&b);                  /* c[0]=2 und Wertänderung
```

```

a=plus2(c,&b);           von a=2                */
                        /* a=3 und Wertänderung
                        von c[0]=3              */
a=plus1(c,b);           /* unzulässiger
                        Funktionsaufruf         */

```

## 10.4 Funktionsergebnis

In der Funktion wird der Rückgabewert mit der **return**-Anweisung spezifiziert. Mit dieser Anweisung gibt die Funktion auch gleichzeitig die Steuerung wieder an die aufrufende Programmeinheit. Falls die Funktion keinen Wert zurückliefern soll, ist für sie der Typ **void** zu vereinbaren. In diesem Fall wird die **return**-Anweisung ohne Parameter verwendet, oder die Funktion endet mit dem Erreichen des Blockendes. Wird kein Typ angegeben, ist die Funktion automatisch vom Typ **int** und liefert einen **int**-Wert zurück.

Beispiel 1:

```

plus(int a, int b, int c) {           /* Funktion bildet Summe
    return a+b+c;                      aus 3 Zahlen    */
}
printf("Summe: %i",plus(1,2,3)); /* Ausgabe Summe      */

```

Beispiel 2:

```

long fak(int x) {                     /* Funktion zur
    long y=1;                          Fakultätsberechnung  */
    while (x>1) y*=x--;
    return y;
}
printf("%l\n",fak(6));                /* Berechnung, Ausgabe der
                                      Fakultät von 6      */

```

## 10.5 Zeiger auf Funktionen

In C kann man auch Zeiger auf Funktionen definieren, wobei jedoch die Operatorpriorität zu beachten ist.

```

int a() {...}                       /* parameterlose mit int-Ergebniswert */
int *b() {...}                      /* liefert Zeiger auf int-Wert !!!    */
int (*c)();                         /* Zeiger auf parameterlose Funktion,
                                   die int-Wert liefert                */
c=&a;                                /* c wird Adresse der Funktion a
                                   zugewiesen */

```

Ein Zeiger auf eine Funktion kann zum Aufruf dieser Funktion verwendet werden.

```
int d;  
d=(*c)();           /* Aufruf der Funktion a über Zeiger c */  
d=c();              /* bewirkt dasselbe                      */
```

Da sich Funktionszeiger auch als Parameter an Funktionen übergeben lassen, sind sehr flexible Funktionsdefinitionen möglich.

## 10.6 Argumente aus der Kommandozeile

Viele Programme sind parametrisierbar, d.h. sie können mit Parametern aufgerufen werden. Bei der Programmierung eines solchen Programms müssen diese Programmparameter natürlich vom Programmierer abgefragt und ausgewertet werden. Hierfür muss die Funktion `main` mit folgenden zwei Argumenten definiert werden:

```
int main(int argc, char *argv[]) {  
    ...  
    return a;  
}
```

Das erste Funktionsparameter `argc` gibt die Anzahl der Programmparameter **+1** an. Das Element `argv[0]` enthält die Adresse einer Zeichenkette mit dem Namen unter dem das Programm aufgerufen wurde, das Element `argv[1]` die Adresse einer Zeichenkette mit dem ersten Programmparameter, das Element `argv[2]` die Adresse einer Zeichenkette mit dem zweiten Programmparameter usw. Dem letzten Argument folgt ein `NULL`-Pointer.

Folgender Programmaufruf enthält z.B. zwei Programmparameter

```
convert datei.ps datei.pdf
```

Dann ist `argc` gleich 3 und

```
argv[0] zeigt auf "convert"  
argv[1] zeigt auf "datei.ps"  
argv[2] zeigt auf "datei.pdf"  
argv[3] enthält den Wert NULL
```

Der `return`-Wert, den die `main()`-Funktion bildet, wird an die aufrufende Console zurückgeliefert und kann dort z.B. in Shell-Skripten ausgewertet werden.

Beachte, dass die Zeichenketten in C intern mit einem `'\0'` abgeschlossen sind.

## 10.7 Rekursive Funktionsaufrufe

Funktionen können sich selbst sowohl direkt als auch indirekt aufrufen. Einen solchen Funktionsaufruf nennt man rekursiv. Der Vorteil rekursiver Funktionen liegt vorrangig in der Kompaktheit und einer besseren Verständlichkeit der Funktionscodes.

```
long fak(int x) {    /* Funktion zur Fakultätsberechnung */
    if (x==1) return 1;
        else return x*fak(x-1);
}
```

## Kapitel 11

# Präprozessoranweisungen

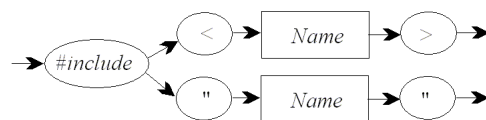
Ein Programm kann aus mehreren Quelltextfiles bestehen und eine Vielzahl von Bibliotheksfunktionen nutzen. Hierfür ist es notwendig, gemeinsam zu nutzende Programmteile einheitlich und zentral zu definieren und diese Definitionen vor der Übersetzung hinzuladen. Vor allem für plattformübergreifende Programme ist es wichtig, in Abhängigkeit von einer Bedingung, Quelltext für den Compiler sichtbar bzw. unsichtbar zu machen. Diese Möglichkeiten bietet C durch einen sogenannten Präprozessor, der vor dem Compiler arbeitet und in Abhängigkeit von gegebenen Präprozessoranweisungen den „eigentlichen“ Quelltext für den Compiler generiert.

Eine Präprozessoranweisung beginnt immer in der 1. Position einer Zeile mit dem Zeichen '#' und ist zeilengebunden (d.h. sie darf nur eine Zeile lang sein). Fortsetzungszeilen können dadurch erzeugt werden, dass ein '\' (Backslash) jeweils vor dem Zeilenende gesetzt wird. Über Präprozessoranweisungen können Fileeinschluss, Definitionsanweisungen und bedingte Übersetzung realisiert werden.

### 11.1 include-Anweisung

Durch die `#include`-Anweisung wird in den C-Quelltext an die Stelle der Anweisung der Inhalt des spezifizierten Files eingefügt, die ebenfalls C- Quelltext enthalten müssen. Das „hinzugeladene“ File darf wiederum `#include`-Anweisungen enthalten.

Syntax:



Es muss ein im zugrundeliegenden Betriebssystem gültige Bezeichnung für ein File angegeben werden. Diese Datei wird in allen Verzeichnissen gesucht, die dem Compiler als „Include-Verzeichnisse“ bekannt sind (meist durch Setzen einer Umgebungsvariable IN-

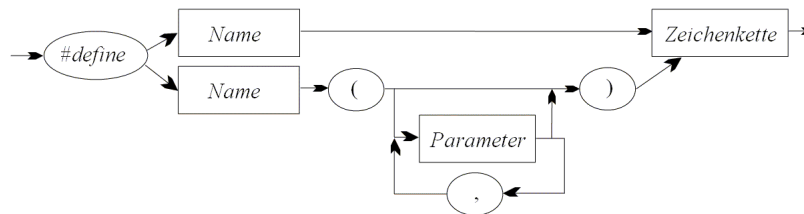
CLUDE oder spezielle Compilerparameter). Wird der Dateiname in Anführungszeichen geschrieben, so wird die Datei vorher noch zusätzlich im aktuellen Arbeitsverzeichnis gesucht.

```
#include <stdio.h>      /* Header E/A-Funktionen      */
#include "projekt.h"     /* eigene Projektdefinitionen */
```

## 11.2 define-Anweisung

Mit der `#define`-Anweisung kann man einfache Makrokonstanten definieren und diese auch parametrisieren.

Syntax:



Bei der Definition einfacher Makrokonstanten wird die Zeichenkette bis zur erneuten Definition dieses Makros bzw. bis zum Löschen der Definition für den Identifikator eingesetzt (substituiert). Oft werden für Identifikatoren von Makros Großbuchstaben verwendet, um im Quelltext eine Makrokonstante von anderen Definitionen abzuheben. Hierzu besteht aber syntaktisch keinerlei Zwang.

```
#define TRUE      (-1)
#define BUFFERSIZE 1024
```

Makrokonstanten können parametrisiert werden, wobei zu beachten ist, dass kein Leerzeichen zwischen dem Identifikator der Makrokonstante und der öffnenden runden Klammer stehen, die die Parameterliste einleitet. Bei der Ersetzung des Identifikators im Quelltext werden ebenfalls der/die formalen Parameter im Ersetzungsausdruck durch die aktuellen ausgetauscht. Die formalen Parameter sind im Ersetzungsausdruck zu klammern, um die Operatorprioritäten im Ersetzungsausdruck für den Fall zu erhalten, dass Ausdrücke als Parameter übergeben werden.

Fehlt bei der Makrodefinition der Ersetzungsausdruck, so löscht der Präprozessor die symbolische Konstante bzw. den Makroidentifikator aus dem nachfolgenden Quelltext.

```
#define ABS(x)    ((x)<0?-(x):(x))
#define MAX(x,y) ((x)<(y)?(y):(x))
```



Alle derart vereinbarten Identifikatoren können (z.B. vor dem Quelltextende) wieder gelöscht werden.

Syntax:

```
#undef Identifikator
```

```
#undef TRUE
#undef ABS
#undef MAX
#undef BUFFERSIZE
```

## 11.3 Bedingte Übersetzung

Mit Hilfe der Präprozessoranweisungen für die bedingte Übersetzung können komplexe Programme relativ schnell an die Gegebenheiten spezieller Plattformen angepasst (portiert) werden.

Die bedingte Übersetzung wird mit `#if`, `#ifdef` bzw. `#ifndef` eingeleitet. Der auf diese Anweisungen folgende Quelltext wird nur dann für den Compiler bereitgestellt, wenn der auf `#if` folgende Konstantenausdruck den Wert „wahr“ (`!=0`) liefert, oder der auf `#ifdef` folgende Identifikator vorher in der `#define`-Anweisung definiert bzw. der auf `#ifndef` folgende nicht definiert wurde. Die bedingte Übersetzung kann mit `#elif` um alternative Testausdrücke erweitert werden. Kann keiner dieser Ausdrücke mit „wahr“ ausgewertet werden, so wird der Quelltext nach der `#else`-Anweisung generiert. Die bedingte Übersetzung endet mit der `#endif`-Anweisung bzw. bei Quelltextende.

```
#define LINUX    1          /* Wahl der Plattform          */
#define WINDOWS 0

#if LINUX
...                        /* Quelltext wird übersetzt  */
#else
...                        /* Quelltext wird ausgeblendet */
#endif
```

## Kapitel 12

# ANSI-C Bibliotheksfunktionen

Nachfolgend werden die wichtigsten Funktionen der ANSI-C Standardbibliotheken kurz vorgestellt. Dies geschieht in enger Anlehnung die Ausführungen der Autoren Axel Stutz und Peter Klingebiel an der FH Fulda, siehe <http://www.fh-fulda.de/~klingebiel/>.

### 12.1 stdio.h – Standard E/A-Funktionen

Die Ein- und Ausgabefunktionen, Typen und Makros, die in `stdio.h` vereinbart sind, machen nahezu ein Drittel der Bibliothek aus.

Ein Datenstrom (stream) ist Quelle oder Ziel von Daten und wird mit einer Datei oder einem anderen Peripheriegerät verknüpft. Die Bibliothek unterstützt zwei Arten von Datenströmen, für Text und binäre Information, die allerdings bei manchen Systemen und insbesondere bei UNIX identisch sind. Ein Textstrom ist eine Folge von Zeilen; jede Zeile enthält eine (ggf. leere) Folge von Zeichen und ist mit `'\n'` abgeschlossen. Eine Umgebung muss möglicherweise zwischen einem Textstrom und einer anderen Repräsentierung umwandeln (also zum Beispiel `'\n'` als Wagenrücklauf und Zeilenvorschub abbilden). Wird ein Binärstrom geschrieben und auf dem gleichen System wieder eingelesen, so entsteht die gleiche Information.

Wenn die Ausführung eines Programms beginnt, sind die drei Ströme `stdin`, `stdout` und `stderr` bereits eröffnet.

#### 12.1.1 Dateioperationen

Die folgenden Funktionen beschäftigen sich mit Dateioperationen. Der Typ `size_t` ist der vorzeichenlose, ganzzahlige Resultattyp des `sizeof`-Operators.

`FILE *fopen(const char *filename, const char *mode)`

eröffnet die angegebene Datei und liefert einen Datenstrom oder NULL bei Mißerfolg.

Zu den erlaubten Werten von `mode` gehören

<code>r</code>	Textdatei zum lesen und eröffnen
<code>w</code>	Textdatei zum Schreiben erzeugen; gegebenenfalls alten Inhalt verwerfen
<code>a</code>	anfügen; Textdatei zum Schreiben am Dateiende eröffnen oder erzeugen
<code>r+</code>	Textdatei zum Ändern eröffnen (Lesen und Schreiben)
<code>w+</code>	Textdatei zum Ändern erzeugen; gegebenenfalls alten Inhalt verwerfen
<code>a+</code>	anfügen; Textdatei zum Ändern eröffnen oder erzeugen, Schreiben am Ende

Ändern bedeutet, dass die gleiche Datei gelesen und geschrieben werden darf; `fflush` oder eine Funktion zum Positionieren in Dateien muss zwischen einer Lese- und einer Schreiboperation oder umgekehrt aufgerufen werden. Enthält `mode` nach dem ersten Zeichen noch `b`, also etwa `rb` oder `w+b`, dann wird auf eine binäre Datei zugegriffen. Dateinamen sind auf `FILENAME_MAX` Zeichen begrenzt. Höchstens `FOPEN_MAX` Dateien können gleichzeitig offen sein.

`FILE *freopen(const char *filename, const char *mode, FILE * stream)`  
eröffnet die Datei für den angegebenen Zugriff `mode`. Das Resultat ist ein Stream oder `NULL` bei einem Fehler. Mit `freopen` ändert man normalerweise die Dateien, die mit `stdin`, `stdout` oder `stderr` verknüpft sind.

`int fflush(FILE *stream)`  
Bei einem Ausgabestrom sorgt `fflush` dafür, dass gepufferte, aber noch nicht geschriebene Daten geschrieben werden; bei einem Eingabestrom ist der Effekt undefiniert. Die Funktion liefert `EOF` bei einem Schreibfehler und sonst `NULL`.  
`fflush(NULL)` bezieht sich auf alle offenen Dateien.

`int fclose(FILE *stream)`  
schreibt noch nicht geschriebene Daten, wirft noch nicht gelesene, gepufferte Eingaben weg, gibt automatisch angelegte Puffer frei und schließt den Datenstrom. Die Funktion liefert `EOF` bei Fehlern und sonst `NULL`.

`int remove(const char *filename)`  
entfernt die angegebene Datei, so dass ein anschließender Versuch, sie zu eröffnen, fehlschlagen wird. Die Funktion liefert bei Fehlern einen von `NULL` verschiedenen Wert.

`int rename(const char *oldname, const char *newname)`  
ändert den Namen einer Datei und liefert `!=0`, wenn der Versuch fehlschlägt.

`FILE *tmpfile(void)`  
erzeugt eine temporäre Datei mit Zugriff `wb+`, die automatisch gelöscht wird, wenn der Zugriff abgeschlossen wird, oder wenn das Programm normal zu Ende geht. `tmpfile` liefert einen Datenstrom oder `NULL`, wenn die Datei nicht erzeugt werden konnte.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`

kontrolliert die Pufferung bei einem Datenstrom; die Funktion muss aufgerufen werden, bevor gelesen oder geschrieben wird, und vor allen anderen Operationen. Hat `mode` den Wert `_IOFBF`, so wird vollständig gepuffert, `_IOLBF` sorgt für zeilenweise Pufferung bei Textdateien und `_IONBF` verhindert Puffern. Wenn `buf` nicht `NULL` ist, wird `buf` als Puffer verwendet; andernfalls wird ein Puffer angelegt. `size` legt die Puffergröße fest. Bei einem Fehler liefert `setvbuf` `!=NULL`.

`void setbuf(FILE *stream, char *buf)`

Wenn `buf` den Wert `NULL` hat, wird der Datenstrom nicht gepuffert. Andernfalls ist `setbuf` äquivalent zu `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

### 12.1.2 Formatierte Ausgabe

Die `printf`-Funktionen ermöglichen Ausgabe-Umwandlungen unter Formatkontrolle.

`int fprintf(FILE *stream, const char *format, ...)`

`fprintf` wandelt Ausgaben um und schreibt sie in `stream` unter Kontrolle von `format`. Der Resultatwert ist die Anzahl der geschriebenen Zeichen; er ist negativ, wenn ein Fehler aufgetreten ist.

Die Format-Zeichenkette enthält zwei Arten von Objekten:

- gewöhnliche Zeichen, die in die Ausgabe kopiert werden, und
- Umwandlungsangaben, die jeweils die Umwandlung und Ausgabe des nächstfolgenden Arguments von `fprintf` veranlassen.

Jede Umwandlungsangabe beginnt mit dem Zeichen `%` und endet mit einem Umwandlungszeichen. Zwischen `%` und dem Umwandlungszeichen kann, der Reihenfolge nach, folgendes angegeben werden:

- Steuerzeichen (flags) (in beliebiger Reihenfolge), die die Umwandlung modifizieren:

-	veranlasst Ausrichtung des umgewandelten Arguments in seinem Feld nach links.
+	bestimmt, dass die Zahl immer mit Vorzeichen ausgegeben wird.
, ' ,	wenn das erste Zeichen kein Vorzeichen ist, wird ein Leerzeichen vorangestellt.
0	legt bei numerischen Umwandlungen fest, dass bis zur Feldbreite mit führenden Nullen aufgefüllt wird.
#	verlangt eine alternative Form der Ausgabe.
o	Als erste Ziffer wird eine Null ausgegeben (oktal).
x,X	Es werden 0x oder 0X einem von Null verschiedenen Resultat vorangestellt (hexadezimal).
e,E,f,g,G	Die Ausgabe erhält einen Dezimalpunkt.
g,G	Die Nullen am Schluß werden nicht unterdrückt.

- Eine Zahl, die eine minimale Feldbreite festlegt. Das umgewandelte Argument wird in einem Feld ausgegeben, das mindestens so breit ist und bei Bedarf auch breiter. Hat das umgewandelte Argument weniger Zeichen als die Feldbreite verlangt, wird links (oder rechts, wenn Ausrichtung nach links verlangt wurde) auf die Feldbreite aufgefüllt. Normalerweise wird mit Leerzeichen aufgefüllt, aber auch mit Nullen, wenn das entsprechende Steuerzeichen angegeben wurde.
- Ein Punkt, der die Feldbreite von der Genauigkeit (precision) trennt. Eine Zahl, die Genauigkeit, die die maximale Anzahl von Zeichen festlegt, die von einer Zeichenkette ausgegeben werden, oder die Anzahl von Ziffern, die nach dem Dezimalpunkt bei `e`, `E`, oder `f` Umwandlungen ausgegeben werden, oder die Anzahl signifikanter Ziffern bei `g` oder `G` Umwandlung oder die minimale Anzahl von Ziffern, die bei einem ganzzahligen Wert ausgegeben werden sollen (führende Nullen werden dann bis zur gewünschten Breite hinzugefügt).
- Ein Buchstabe als Längenangabe: `h`, `l` oder `L`.  
`h` bedeutet, dass das zugehörige Argument als `short` oder `unsigned short` ausgegeben wird;  
`l` bedeutet, dass das Argument `long` oder `unsigned long` ist;  
`L` bedeutet, dass das Argument `long double` ist.

Als Feldbreite oder Genauigkeit kann jeweils `*` angegeben werden; dann wird der Wert durch Umwandlung von dem nächsten oder den zwei nächsten Argumenten festgelegt, die den Typ `int` besitzen müssen.

- Einer der nachfolgenden Buchstaben, der Datentyp des auszugebenden Wertes angibt.

<b>d,i</b>	<b>int</b>	dezimal mit Vorzeichen.
<b>o</b>	<b>int</b>	oktal ohne Vorzeichen (ohne führende Null).
<b>x,X</b>	<b>int</b>	hexadezimal ohne Vorzeichen (ohne führendes 0x oder 0X) mit abcdef bei 0x oder ABCDEF bei 0X.
<b>u</b>	<b>int</b>	dezimal ohne Vorzeichen.
<b>c</b>	<b>int</b>	int einzelnes Zeichen, nach Umwandlung in unsigned char.
<b>s</b>	<b>char*</b>	aus einer Zeichenkette werden Zeichen ausgegeben bis zum Sringendeckennzeichen oder so viele Zeichen, wie die Genauigkeit verlangt.
<b>f</b>	<b>double</b>	dezimal als [-]mmm.ddd, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
<b>e,E</b>	<b>double</b>	dezimal als [-]m.ddddde±xx oder [-]m.dddddE±xx, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
<b>g,G</b>	<b>double</b>	%e, %E oder %f, je nachdem was kürzer dargestellt werden kann. Null und Dezimalpunkt am Schluß werden nicht ausgegeben.
<b>p</b>	<b>void*</b>	als Zeiger (Darstellung hängt von Implementierung ab).
<b>%</b>		kein Argument wird umgewandelt; ein % wird ausgegeben.

`int printf(const char *format, ...)`  
ist äquivalent zu `fprintf(stdout, ...)`.

`int sprintf(char *s, const char *format, ...)`  
funktioniert wie `printf`, nur wird die Ausgabe in den Zeichenvektor `s` geschrieben und mit `\0` abgeschlossen. `s` muss groß genug für das Resultat sein. Im Resultatwert wird `\0` nicht mitgezählt.

### 12.1.3 Formatierte Eingabe

Die `scanf`-Funktionen behandeln Eingabe-Umwandlungen unter Formatkontrolle.

`int fscanf(FILE *stream, const char *format, ...)`

`fscanf` liest von `stream` unter Kontrolle von `format` und legt umgewandelte Werte mit Hilfe von nachfolgenden Argumenten ab, die alle Zeiger sein müssen. Die Funktion wird beendet, wenn `format` abgearbeitet ist. `fscanf` liefert EOF, wenn vor der ersten Umwandlung das Dateiende erreicht wird oder ein Fehler passiert; andernfalls liefert die Funktion die Anzahl der umgewandelten und abgelegten Eingaben. Die Format-Zeichenkette enthält normalerweise Umwandlungsangaben, die zur Interpretation der Eingabe verwendet werden. Die Format-Zeichenkette kann folgendes enthalten:

- Leerzeichen oder Tabulatorzeichen, die ignoriert werden.

- Gewöhnliche Zeichen (nicht aber %), die dem nächsten Zeichen nach Zwischenraum im Eingabestrom entsprechen müssen.
- Umwandlungsangaben, bestehend aus %; einem optionalen Zeichen \*, das die Zuweisung an ein Argument verhindert; einer optionalen Zahl, die die maximale Feldbreite festlegt; einem optionalen Buchstaben h, l, oder L, der die Länge des Ziels beschreibt; und einem Umwandlungszeichen.

Eine Umwandlungsangabe bestimmt die Umwandlung des nächsten Eingabefelds. Normalerweise wird das Resultat in der Variablen abgelegt, auf die das zugehörige Argument zeigt. Wenn jedoch \* die Zuweisung verhindern soll, wie bei %\*s, dann wird das Eingabefeld einfach übergangen und eine Zuweisung findet nicht statt.

Ein Eingabefeld ist als Folge von Zeichen definiert, die keine Zwischenraumzeichen sind. Es reicht entweder bis zum nächsten Zwischenraumzeichen, oder bis eine explizit angegebene Feldbreite erreicht ist. Daraus folgt, dass scanf über Zeilengrenzen hinweg liest, um seine Eingabe zu finden, denn Zeilentrenner sind Zwischenraumzeichen. Zwischenraumzeichen sind Leerzeichen, Tabulatorzeichen \t, Zeilentrenner \n, Wagenrücklauf \r, Vertikaltabulator \v und Seitenvorschub \f.

Das Umwandlungszeichen gibt die Interpretation des Eingabefelds an. Das zugehörige Argument muss ein Zeiger sein. Folgende Umwandlungszeichen sind erlaubt:

d	dezimal	int	
i	ganzzahlig	int	Der Wert kann oktal (mit 0 am Anfang) oder hexadezimal (mit 0x oder 0X am Anfang) angegeben sein.
o	oktal	int	mit oder ohne 0 am Anfang
u	dezimal	unsigned int	ohne Vorzeichen
x	hexadezimal	int	mit oder ohne 0x oder 0X am Anfang
c	Zeichen	char	Auch Trennzeichen (Zwischenraumzeichen) werden dabei gelesen.
s	Zeichenfolge	char-Vektor	Folge von Nicht-Zwischenraum-Zeichen (ohne Anführungszeichen), \0 wird nachfolgend angehängt
e,f,g	Gleitpunkt	float	Das Eingabeformat erlaubt für float ein optionales Vorzeichen, eine Ziffernfolge, die auch einen Dezimalpunkt enthalten kann, und einen optionalen Exponenten, bestehend aus E oder e und einer ganzen Zahl, optional mit Vorzeichen.
p	Zeiger	void*	wie ihn printf("%p") ausgibt
[ ]	Zeichenfolge	char-Vektor	Erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen die zwischen [ und ] angegeben wurden. Dazu kommt \0.
[^ ]	Zeichenfolge	char-Vektor	Erkennt die längste nicht-leere Zeichenkette bis zu einem der Eingabezeichen die zwischen [^ und ] angegeben wurden. Dazu kommt \0.
%			erkennt %; eine Zuweisung findet nicht statt.

Den Umwandlungszeichen d, i, n, o, u und x kann h vorausgehen, wenn das Argument ein Zeiger auf short statt int ist, oder der Buchstabe l, wenn das Argument ein

Zeiger auf `long` ist. Vor den Umwandlungszeichen `e`, `f` und `g` kann der Buchstabe `l` stehen, wenn ein Zeiger auf `double` und nicht auf `float` in der Argumentliste steht, und `L`, wenn es sich um einen Zeiger auf `long double` handelt.

`int scanf(const char *format, ...)`  
ist äquivalent zu `fscanf(stdin, ...)`.

`int sscanf(const char *s, const char *format, ...)`  
ist äquivalent zu `scanf(...)`, mit dem Unterschied, dass die Eingabezeichen aus der Zeichenkette `s` stammen.

#### 12.1.4 Ein- und Ausgabe von Zeichen

`int fgetc(FILE *stream)`  
liefert das nächste Zeichen aus `stream` als `unsigned char` (umgewandelt in `int`) oder EOF bei Dateiende oder bei einem Fehler.

`char *fgets(char *s, int n, FILE *stream)`  
liest höchstens die nächsten `n-1` Zeichen in `s` ein und hört vorher auf, wenn ein Zeilentrenner gefunden wird. Der Zeilentrenner wird im Vektor abgelegt. Der Vektor wird mit `\0` abgeschlossen. `fgets` liefert `s`, `NULL` bei Dateiende oder bei einem Fehler.

`int fputc(int c, FILE *stream)`  
schreibt das Zeichen `c` (umgewandelt in `unsigned char`) in `stream`. Die Funktion liefert das ausgegebene Zeichen oder EOF bei Fehler.

`int fputs(const char *s, FILE *stream)`  
schreibt die Zeichenkette `s` (die `'\n'` nicht zu enthalten braucht) in `stream`. Die Funktion liefert einen nicht-negativen Wert oder EOF bei einem Fehler.

`int getc(FILE *stream)`  
ist äquivalent zu `fgetc`, kann aber ein Makro sein und dann das Argument für `stream` mehr als einmal bewerten.

`int getchar(void)`  
`getchar` ist äquivalent zu `getc(stdin)`.

`char *gets(char *s)`  
liest die nächste Zeile von `stdin` in den Vektor `s` und ersetzt dabei den abschließenden Zeilentrenner durch `\0`. Die Funktion liefert `s` oder `NULL` bei Dateiende oder bei einem Fehler.

`int putc(int c, FILE *stream)`  
ist äquivalent zu `fputc`, kann aber ein Makro sein und dann das Argument für `stream` mehr als einmal bewerten.



**int putchar(int c)**

putchar(c) ist äquivalent zu putc(c, stdout).

**int puts(const char \*s)**

schreibt die Zeichenkette **s** und einen Zeilentrenner in **stdout**. Die Funktion liefert EOF, wenn ein Fehler passiert, andernfalls einen nicht- negativen Wert.

**int ungetc(int c, FILE \*stream)**

stellt **c** (umgewandelt in **unsigned char**) in **stream** zurück, von wo das Zeichen beim nächsten Lesevorgang wieder geholt wird. Man kann sich nur darauf verlassen, dass pro Datenstrom ein Zeichen zurückgestellt werden kann. EOF darf nicht zurückgestellt werden. **ungetc** liefert das zurückgestellte Zeichen oder EOF bei einem Fehler.

### 12.1.5 Direkte Ein- und Ausgabe

**size\_t fread(void \*ptr, size\_t size, size\_t nobj, FILE \* stream)**

liest aus **stream** in den Vektor **ptr** höchstens **nobj** Objekte der Größe **size** ein. **fread** liefert die Anzahl der eingelesenen Objekte; das kann weniger als die geforderte Zahl sein. Der Zustand des Datenstroms muss mit **feof** und **ferror** untersucht werden.

**size\_t fwrite(const void \*ptr, size\_t size, size\_t nobj, FILE \*stream)**

schreibt **nobj** Objekte der Größe **size** aus dem Vektor **ptr** in **stream**. Die Funktion liefert die Anzahl der ausgegebenen Objekte; bei Fehler ist das weniger als **nobj**.

### 12.1.6 Positionieren in Dateien

**int fseek(FILE \*stream, long offset, int origin)**

setzt die Dateiposition für **stream**. Eine nachfolgende Lese- oder Schreiboperation wird auf Daten von der neuen Position ab zugreifen. Für eine binäre Datei wird die Position auf **offset** Zeichen relativ zu **origin** eingestellt; dabei können die Werte **SEEK\_SET** (Dateianfang) **SEEK\_CUR** (aktuelle Position) oder **SEEK\_END** (Dateiende) angegeben werden. Für einen Textstrom muss **offset** 0 sein oder ein Wert, der von **ftell** stammt (dafür muss dann **origin** den Wert **SEEK\_SET** erhalten). **fseek** liefert einen von 0 verschiedenen Wert bei Fehler.

**long ftell(FILE \*stream)**

liefert die aktuelle Dateiposition für **stream** oder -1L bei Fehler.

**void rewind(FILE \*stream)**

ist äquivalent zu **fseek(stream, 0L, SEEK\_SET)**.

**int fgetpos(FILE \*stream, fpos\_t \*ptr)**

speichert die aktuelle Position für **stream** bei **\*ptr**. Der Wert kann später mit **fsetpos** verwendet werden. Der Datentyp **fpos\_t** eignet sich zum Speichern von solchen Werten. Bei Fehler liefert **fgetpos** einen von 0 verschiedenen Wert.

`int fsetpos(FILE *stream, const fpos_t *ptr)`  
positioniert `stream` auf die Position, die von `fgetpos` in `*ptr` abgelegt wurde. Bei Fehler liefert `fsetpos` einen von 0 verschiedenen Wert.

### 12.1.7 Fehlerbehandlung

Viele der Bibliotheksfunktionen notieren Zustandsangaben, wenn ein Dateiende oder ein Fehler gefunden wird. Diese Angaben können explizit gesetzt und getestet werden. Außerdem kann der Integer-Ausdruck `errno` (der in `errno.h` deklariert ist) eine Fehlernummer enthalten, die mehr Information über den zuletzt aufgetretenen Fehler liefert.

`void clearerr(FILE *stream)`  
löscht die Dateiende- und Fehlernotizen für `stream`.

`int feof(FILE *stream)`  
liefert einen von NULL verschiedenen Wert, wenn für `stream` ein Dateiende notiert ist.

`int ferror(FILE *stream)`  
liefert einen von NULL verschiedenen Wert, wenn für `stream` ein Fehler notiert ist.

`void perror(const char *s)`  
gibt `s` und eine von der Implementierung definierte Fehlermeldung aus, die sich auf die Fehlernummer in `errno` bezieht.

## 12.2 stdlib.h – Allgemeine Hilfsfunktionen

Die Definitionsdatei `stdlib.h` vereinbart Funktionen zur Umwandlung von Zahlen, für Speicherverwaltung und ähnliche Aufgaben.

`double atof(const char *s)`  
wandelt `s` in `double` um; die Funktion ist äquivalent zu  
`strtod(s, (char**) NULL)`

`int atoi(const char *s)`  
wandelt `s` in `int` um; die Funktion ist äquivalent zu  
`(int)strtol(s, (char**) NULL, 10)`

`long atol(const char *s)`  
wandelt `s` in `long` um; die Funktion ist äquivalent zu  
`strtol(s, (char**) NULL, 10)`

`int rand(void)`  
liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX`; dieser Wert ist mindestens 32767.

**void srand(unsigned int seed)**  
srand benutzt **seed** als Ausgangswert für eine neue Folge von Pseudo- Zufallszahlen. Der erste Ausgangswert ist 1.

**void \*calloc(size\_t nobj, size\_t size)**  
liefert einen Zeiger auf einen Speicherbereich für einen Vektor von **nobj** Objekten, jedes mit der Größe **size**, oder NULL, wenn die Anforderung nicht erfüllt werden kann. Der Bereich wird mit NULL-Bytes initialisiert.

**void \*malloc(size\_t size)**  
liefert einen Zeiger auf einen Speicherbereich für ein Objekt der Größe **size** oder NULL, wenn die Anforderung nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

**void \*realloc(void \*p, size\_t size)**  
ändert die Größe des Objekts, auf das **p** zeigt, in **size** ab. Bis zur kleineren der alten und neuen Größe bleibt der Inhalt unverändert. Wird der Bereich für das Objekt größer, so ist der zusätzliche Bereich uninitiatisiert. **realloc** liefert einen Zeiger auf den neuen Bereich oder NULL, wenn die Anforderung nicht erfüllt werden kann; in diesem Fall ist **\*p** unverändert.

**void free(void \*p)**  
gibt den Bereich frei, auf den **p** zeigt; die Funktion hat keinen Effekt, wenn **p** den Wert NULL hat. **p** muss auf einen Bereich zeigen, der zuvor mit **calloc**, **malloc** oder **realloc** angelegt wurde.

**void abort(void)**  
sorgt für eine anormale Beendigung des Programms.

**void exit(int status)**  
beendet das Programm regulär. Offene Ströme werden geschlossen, und die Kontrolle geht an die Umgebung des Programms zurück. Wie **status** an die Umgebung des Programms geliefert wird, hängt von der Implementierung ab, aber NULL gilt als erfolgreiches Ende. Die Werte **EXIT\_SUCCESS** und **EXIT\_FAILURE** können ebenfalls angegeben werden.

**int atexit(void (\*fcn)(void))**  
hinterlegt die Funktion **fcn**, damit sie aufgerufen wird, wenn das Programm normal endet, und liefert einen von 0 verschiedenen Wert, wenn die Funktion nicht hinterlegt werden kann.

**int system(const char \*s)**  
liefert die Zeichenkette **s** an die Umgebung zur Ausführung. Hat **s** den Wert NULL, so liefert **system** einen von NULL verschiedenen Wert, wenn es einen Kommandoprozessor gibt. Wenn **s** von NULL verschieden ist, dann ist der Resultatwert implementierungsabhängig.

`char *getenv(const char *name)`

liefert die zu `name` gehörende Zeichenkette aus der Umgebung oder `NULL`, wenn keine Zeichenkette existiert. Die Details hängen von der Implementierung ab.

`void *bsearch(const void *key, const void *base, size_t n,  
              size_t size, int (*cmp)(const void *keyval, const void *datum))`  
durchsucht `base[0] ... base[n-1]` nach einem Eintrag, der gleich `*key` ist. Die Funktion `cmp` muss einen negativen Wert liefern, wenn ihr erstes Argument (der Suchschlüssel) kleiner als ihr zweites Argument (ein Tabelleneintrag) ist, `NULL`, wenn beide gleich sind, und sonst einen positiven Wert. Die Elemente des Vektors `base` müssen aufsteigend sortiert sein. `bsearch` liefert einen Zeiger auf das gefundene Element oder `NULL`, wenn keines existiert.

`void qsort(void *base, size_t n, size_t size,  
           int (*cmp)(const void *, const void *))`  
sortiert einen Vektor `base[0] ... base[n-1]` von Objekten der Größe `size` in aufsteigender Reihenfolge. Für die Vergleichsfunktion `cmp` gilt das gleiche wie bei `bsearch`.

`int abs(int n)`  
liefert den absoluten Wert seines `int` Arguments.

`long labs(long n)`  
liefert den absoluten Wert seines `long` Arguments.

`div_t div(int num, int denom)`  
berechnet Quotient und Rest von `num/denom`. Die Resultate werden in den `int` Komponenten `quot` und `rem` einer Struktur vom Typ `div_t` abgelegt.

`ldiv_t ldiv(long num, long denom)`  
berechnet Quotient und Rest von `num/denom`. Die Resultate werden in den `long` Komponenten `quot` und `rem` einer Struktur vom Typ `ldiv_t` abgelegt.

## 12.3 string.h – Funktionen für Zeichenketten

In der Definitionsdatei `string.h` werden zwei Gruppen von Funktionen für Zeichenketten vereinbart. Die erste Gruppe hat Namen, die mit `str` beginnen; die Namen der zweiten Gruppe beginnen mit `mem`.

Bemerkung: Sieht man von `memmove` ab, ist der Effekt der Funktionen undefiniert, wenn zwischen überlappenden Objekten kopiert wird.

`char *strcpy(char *s, const char *ct)`  
Zeichenkette `ct` in Vektor `s` kopieren, inklusive `\0`; liefert `s`.

`char *strncpy(char *s, const char *ct, size_t n)`  
höchstens `n` Zeichen aus `ct` in `s` kopieren; liefert `s`. Mit `\0` auffüllen, wenn `ct` weniger als `n` Zeichen hat.

`char *strcat(char *s, const char *ct)`  
Zeichenkette `ct` an die Zeichenkette `s` anfügen; liefert `s`.

`char *strncat(char *s, const char *ct, size_t n)`  
höchstens `n` Zeichen von `ct` hinten an die Zeichenkette `s` anfügen und `s` mit `\0` abschließen; liefert `s`.

`int strcmp(const char *cs, const char *ct)`  
Zeichenketten `cs` und `ct` vergleichen; liefert `<0` wenn `cs<ct`, `0` wenn `cs==ct`, oder `>0`, wenn `cs>ct`.

`int strncmp(const char *cs, const char *ct, size_t n)`  
höchstens `n` Zeichen von `cs` mit der Zeichenkette `ct` vergleichen; liefert `<0` wenn `cs<ct`, `0` wenn `cs==ct`, oder `>0` wenn `cs>ct`.

`char *strchr(const char *cs, int c)`  
liefert Zeiger auf das erste `c` in `cs` oder `NULL`, falls nicht vorhanden.

`char *strrchr(const char *cs, int c)`  
liefert Zeiger auf das letzte `c` in `cs`, oder `NULL`, falls nicht vorhanden,

`size_t strspn(const char *cs, const char *ct)`  
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich in `ct` vorkommen.

`size_t strcspn(const char *cs, const char *ct)`  
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich in nicht `ct` vorkommen.

`char *strpbrk(const char *cs, const char *ct)`  
liefert Zeiger auf die Position in `cs`, an der irgendein Zeichen aus `ct` erstmals vorkommt, oder `NULL`, falls keines vorkommt.

`char *strstr(const char *cs, const char *ct)`  
liefert Zeiger auf erste Kopie von `ct` in `cs` oder `NULL`, falls nicht vorhanden.

`size_t strlen(const char *cs)`  
liefert Länge von `cs` (ohne `\0`).

`char *strerror(int n)`  
liefert Zeiger auf Zeichenkette, die in der Implementierung für Fehler `n` definiert ist.

`char *strtok(char *s, const char *ct)`  
durchsucht `s` nach Zeichenfolgen, die durch Zeichen aus `ct` begrenzt sind. Eine Folge von Aufrufen von `strtok(s,ct)` zerlegt `s` in Zeichenfolgen, die jeweils durch ein Zeichen aus `ct` begrenzt sind. Beim ersten von einer Reihe von Aufrufen ist `s` nicht

NULL. Dieser Aufruf findet die erste Zeichenfolge in `s`, die nicht aus Zeichen in `ct` besteht; die Folge wird dadurch abgeschlossen, dass das nächste Zeichen in `s` mit `\0` überschrieben wird; der Aufruf liefert dann einen Zeiger auf die Zeichenfolge. Jeder weitere Aufruf der Reihe ist kenntlich dadurch, dass NULL für `s` übergeben wird; er liefert die nächste derartige Zeichenfolge, wobei unmittelbar nach dem Ende der vorhergehenden mit der Suche begonnen wird. `strtok` liefert NULL, wenn keine weitere Zeichenfolge gefunden wird. Die Zeichenkette `ct` kann bei jedem Aufruf verschieden sein.

```
void *memcpy(void *s,const void *ct, size_t n)
    kopiert n Bytes von ct nach s; liefert s.

void *memmove(void *s,const void *ct, size_t n)
    wie memcpy, funktioniert aber auch, wenn die Objekte überlappen.

int memcmp(const void *cs,const void *ct, size_t n)
    vergleicht die ersten n Bytes von cs mit ct; Resultat analog zu strcmp.

void *memchr(const void *cs, int c, size_t n)
    liefert Zeiger auf das erste c in cs oder NULL, wenn das Byte in den ersten n Bytes
    nicht vorkommt.

void *memset(void *s, int c, size_t n)
    setzt die ersten n Bytes von s auf den Wert c, liefert s.
```

## 12.4 ctype.h – Testen und Umwandeln von Zeichen

Die Definitionsdatei `ctype.h` vereinbart Funktionen zum Testen von Zeichen. Jede Funktion hat ein `int`-Argument, dessen Wert entweder EOF ist oder als `unsigned char` dargestellt werden kann, und der Resultatwert hat den Typ `int`. Die Funktionen liefern einen von NULL verschiedenen Wert (wahr), wenn das Argument `c` die beschriebene Bedingung erfüllt; andernfalls liefern sie 0.

```
int isalnum(int c)
    isalpha(c) oder isdigit(c) ist wahr

int isalpha(int c)
    isupper(c) oder islower(c) ist wahr

int iscntrl(int c)
    Steuerzeichen

int isdigit(int c)
    dezimale Ziffer
```

```

int isgraph(int c)
    sichtbares Zeichen, kein Leerzeichen

int islower(int c)
    Kleinbuchstabe (aber kein Umlaut oder ß!)

int isprint(int c)
    sichtbares Zeichen, auch Leerzeichen

int ispunct(int c)
    sichtbares Zeichen, mit Ausnahme von Leerzeichen, Buchstabe oder Ziffer

int isspace(int c)
    Leerzeichen, Seitenvorschub (\f), Zeilentrenner (\n), Wagenrücklauf (\r), Tabula-
    torzeichen (\t), Vertikal-Tabulator (\v)

int isupper(int c)
    Großbuchstabe (aber kein Umlaut!)

```

Im 7-Bit ASCII-Zeichensatz (siehe Seite 77) sind die sichtbaren Zeichen 0x20 (' ') bis 0x7E ('~'); die Steuerzeichen sind 0 (NULL) bis 0x1F (US) sowie 0x7F (DEL).

Zusätzlich gibt es zwei Funktionen zur Umwandlung zwischen Groß- und Kleinbuchstaben. Auch diese Funktionen funktionieren natürlich nicht für deutsche Umlaute und ß.

```

int tolower(int c)
    wandelt c in einen Kleinbuchstaben um

int toupper(int c)
    wandelt c in einen Großbuchstaben um

```

## 12.5 math.h – Mathematische Funktionen

```

double sin(double x)
    Sinus von x

double cos(double x)
    Kosinus von x

double tan(double x)
    Tangens von x

double asin(double x)
     $\arcsin(x)$  mit  $x \in [-1, 1]$ .

double acos(double x)
     $\arccos(x)$  mit  $x \in [-1, 1]$ .

```

`double atan(double x)`  
 berechnet  $\arctan(x)$ .

`double atan2(double x, double y)`  
 berechnet  $\arctan(\frac{y}{x})$ .

`double sinh(double x)`  
 Sinus Hyperbolicus von `x`

`double cosh(double x)`  
 Cosinus Hyperbolicus von `x`

`double tanh(double x)`  
 Tangens Hyperbolicus von `x`

`double exp(double x)`  
 Exponentialfunktion:  $e^x$

`double log(double x)`  
 natürlicher Logarithmus:  $\ln(x), x > 0$ .

`double log10(double x)`  
 Logarithmus zur Basis 10:  $\log_{10}(x), x > 0$ .

`double pow(double x, double y)`  
 $x^y$ . Ein Argumentfehler liegt vor bei  $x = 0$  und  $y < 0$ , oder bei  $x < 0$  und  $y$  ist nicht ganzzahlig.

`double sqrt(double x)`  
 Wurzel von `x`,  $x \geq 0$ .

`double ceil(double x)`  
 kleinster ganzzahliger Wert, der nicht kleiner als `x` ist, als `double`.

`double floor (double x)`  
 größter ganzzahliger Wert, der nicht größer als `x` ist, als `double`.

`double fabs(double x)`  
 absoluter Wert  $|x|$

`double ldexp(double x,n)`  
 berechnet  $x * 2^n$

`double frexp(double x, int *exp)`  
 zerlegt `x` in eine normalisierte Mantisse im Bereich  $[\frac{1}{2}, 1]$ , die als Resultat geliefert wird, und eine Potenz von 2, die in `*exp` abgelegt wird. Ist `x` gleich NULL, sind beide Teile des Resultats NULL.



`double modf(double x, double *ip)`

zerlegt `x` in einen ganzzahligen Teil und einen Rest, die beide das gleiche Vorzeichen wie `x` besitzen. Der ganzzahlige Teil wird bei `*ip` abgelegt, der Rest ist das Resultat.

`double fmod(double x, double y)`

Gleitpunktrest von  $\frac{x}{y}$ , mit dem gleichen Vorzeichen wie `x`. Wenn `y` gleich `NULL` ist, hängt das Resultat von der Implementierung ab.

## 12.6 time.h – Datums- und Zeitfunktionen

Die Definitionsdatei `time.h` vereinbart Typen und Funktionen zum Umgang mit Datum und Uhrzeit. Manche Funktionen verarbeiten die Ortszeit, die von der Kalenderzeit zum Beispiel wegen einer Zeitzone abweicht. `clock_t` und `time_t` sind arithmetische Typen, die Zeiten repräsentieren, und `struct tm` enthält die Komponenten einer Kalenderzeit:

<code>int tm_sec;</code>	Sekunden nach der vollen Minute (0, 61) <sup>1</sup>
<code>int tm_min;</code>	Minuten nach der vollen Stunde (0, 59)
<code>int tm_hour;</code>	Stunden seit Mitternacht (0, 23)
<code>int tm_mday;</code>	Tage im Monat (1, 31)
<code>int tm_mon;</code>	Monate seit Januar (0, 11)
<code>int tm_year;</code>	Jahre seit 1900
<code>int tm_wday;</code>	Tage seit Sonntag (0, 6)
<code>int tm_yday;</code>	Tage seit dem 1. Januar (0, 365)
<code>int tm_isdst;</code>	Kennzeichen für Sommerzeit <code>tm_isdst</code> ist positiv, wenn Sommerzeit gilt, <code>NULL</code> , wenn Sommerzeit nicht gilt, und negativ, wenn die Information nicht zur Verfügung steht.

Die meisten der nachfolgend genannten Funktionen arbeiten nun mit diesen Datentypen:

`clock_t clock(void)`

liefert die Rechnerkern-Zeit, die das Programm seit Beginn seiner Ausführung verbraucht hat, oder -1, wenn diese Information nicht zur Verfügung steht.

`clock()/CLOCKS_PER_SEC` ist eine Zeit in Sekunden.

`time_t time(time_t *tp)`

liefert die aktuelle Kalenderzeit oder -1, wenn diese nicht zur Verfügung steht. Wenn `tp` von `NULL` verschieden ist, wird der Resultatwert auch bei `*tp` abgelegt.

`double difftime(time_t time2, time_t time1)`

liefert `time2-time1` ausgedrückt in Sekunden.

`time_t mktime(struct tm *tp)`

wandelt die Ortszeit in der Struktur `*tp` in Kalenderzeit um, die so dargestellt wird wie bei `time`. Die Komponenten erhalten Werte in den angegebenen Bereichen. `mktime` liefert die Kalenderzeit oder -1, wenn sie nicht dargestellt werden kann.

Die folgenden vier Funktionen liefern Zeiger auf statische Objekte, die von anderen Aufrufen überschrieben werden können.

`char *asctime(const struct tm *tp)`

konstruiert aus der Zeit in der Struktur `*tp` eine Zeichenkette der Form

Sun Jan 3 15:14:13 1988\n\0

`char *ctime(const time_t *tp)`

verwandelt die Kalenderzeit `*tp` in Ortszeit; dies ist äquivalent zu

`asctime(localtime(tp))`

`struct tm *gmtime(const time_t *tp)`

verwandelt die Kalenderzeit `*tp` in Coordinated Universal Time (UTC). Die Funktion liefert NULL, wenn UTC nicht zur Verfügung steht.

`struct tm *localtime(const time_t *tp)`

verwandelt die Kalenderzeit `*tp` in Ortszeit.

`size_t strftime(char *s, size_t smax, const char *fmt, struct tm *tp)`

formatiert Datum und Zeit aus `*tp` in `s` unter Kontrolle von `fmt`, analog zu einem `printf`-Format. Gewöhnliche Zeichen (insbesondere auch das abschließende `\0`) werden nach `s` kopiert. Jeder mit `%` markierte Platzhalter wird so wie unten beschrieben ersetzt, wobei Werte verwendet werden, die der lokalen Umgebung entsprechen. Höchstens `smax` Zeichen werden in `s` abgelegt. `strftime` liefert die Anzahl der resultierenden Zeichen, mit Ausnahme von `\0`. Wenn mehr als `smax` Zeichen erzeugt wurden, liefert `strftime` den Wert NULL.

<code>%a</code>	abgekürzter Name des Wochentags.
<code>%A</code>	voller Name des Wochentags.
<code>%b</code>	abgekürzter Name des Monats.
<code>%B</code>	voller Name des Monats.
<code>%c</code>	lokale Darstellung von Datum und Zeit.
<code>%d</code>	Tag im Monat (01 - 31).
<code>%H</code>	Stunde (00 - 23).
<code>%I</code>	Stunde (01 - 12).
<code>%j</code>	Tag im Jahr (001 - 366).
<code>%m</code>	Monat (01 - 12).
<code>%M</code>	Minute (00 - 59).
<code>%p</code>	lokales Äquivalent von AM oder PM.
<code>%S</code>	Sekunde (00 - 61).
<code>%U</code>	Woche im Jahr (Sonntag ist erster Wochentag) (00 - 53).
<code>%w</code>	Wochentag (0 - 6, Sonntag ist 0).
<code>%W</code>	Woche im Jahr (Montag ist erster Wochentag) (00 - 53).
<code>%x</code>	lokale Darstellung des Datums.
<code>%X</code>	lokale Darstellung der Zeit.

<code>%y</code>	Jahr ohne Jahrhundert (00 - 99).
<code>%Y</code>	Jahr mit Jahrhundert.
<code>%Z</code>	Name der Zeitzone, falls diese existiert.
<code>%%</code>	<code>%</code>

Es gibt weitere ANSI-C Header, die allerdings weit seltener Verwendung finden. Deshalb sollen diese hier nur genannt werden:

`assert.h`, `stdarg.h`, `float.h`, `limits.h`, `setjmp.h`, `stddef.h`, `errno.h`, `locale.h`, `signal.h`.

# Anhang A

## Zugaben

### A.1 Literatur- und Internetverweise

Brian W. Kernighan Dennis M. Ritchie: *Programmieren in C*, Hanser Verlag.

Jürgen Wolf: *C von A bis Z*, Galileo Computing, [www.pronix.de](http://www.pronix.de).

Rolf Isernhagen: *Softwaretechnik in C und C++*, Hanser Verlag.

Peter Prinz, Ulla Kirch-Prinz: *C kurz und gut*, O'Reilly.

Robert Sedgewick: *Algorithmen in C*, Addison-Wesley.

## A.2 ASCII-Tabelle

Strg	Dez	Hex.	Char	Code
^@	0	00	NUL	
^A	1	01	SOH	
^B	2	02	STX	
^C	3	03	ETX	
^D	4	04	EOT	
^E	5	05	ENQ	
^F	6	06	ACK	
^G	7	07	BEL	
^H	8	08	BS	
^I	9	09	HT	
^J	10	0A	LF	
^K	11	0B	VT	
^L	12	0C	FF	
^M	13	0D	CR	
^N	14	0E	SO	
^O	15	0F	SI	
^P	16	10	DLE	
^Q	17	11	DC1	
^R	18	12	DC2	
^S	19	13	DC3	
^T	20	14	DC4	
^U	21	15	NAK	
^V	22	16	SYN	
^W	23	17	ETB	
^X	24	18	CAN	
^Y	25	19	EM	
^Z	26	1A	SUB	
^[	27	1B	ESC	
^\	28	1C	FS	
^]	29	1D	GS	
^^	30	1E	RS	▲
^_	31	1F	US	▼
32	20		!	
33	21		"	
34	22		#	
35	23		\$	
36	24		%	
37	25		&	
38	26		'	
39	27		(	
40	28		)	
41	29		*	
42	2A		+	
43	2B		,	
44	2C		-	
45	2D		.	
46	2E		/	
47	2F		0	
48	30		1	
49	31		2	
50	32		3	
51	33		4	
52	34		5	
53	35		6	
54	36		7	
55	37		8	
56	38		9	
57	39		:	
58	3A		:	
59	3B		:	
60	3C		<	
61	3D		=	
62	3E		>	
63	3F		?	
64	40		@	
65	41		A	
66	42		B	
67	43		C	
68	44		D	
69	45		E	
70	46		F	
71	47		G	
72	48		H	
73	49		I	
74	4A		J	
75	4B		K	
76	4C		L	
77	4D		M	
78	4E		N	
79	4F		O	
80	50		P	
81	51		Q	
82	52		R	
83	53		S	
84	54		T	
85	55		U	
86	56		V	
87	57		W	
88	58		X	
89	59		Y	
90	5A		Z	
91	5B		[	
92	5C		\	
93	5D		]	
94	5E		^	
95	5F		_	
96	60		`	
97	61		a	
98	62		b	
99	63		c	
100	64		d	
101	65		e	
102	66		f	
103	67		g	
104	68		h	
105	69		i	
106	6A		j	
107	6B		k	
108	6C		l	
109	6D		m	
110	6E		n	
111	6F		o	
112	70		p	
113	71		q	
114	72		r	
115	73		s	
116	74		t	
117	75		u	
118	76		v	
119	77		w	
120	78		x	
121	79		y	
122	7A		z	
123	7B		{	
124	7C		}	
125	7D		~	
126	7E		^*	
127	7F		^*	
128	80		Ç	
129	81		ü	
130	82		ë	
131	83		ä	
132	84		ä	
133	85		ä	
134	86		ä	
135	87		ç	
136	88		è	
137	89		è	
138	8A		è	
139	8B		ï	
140	8C		ï	
141	8D		ï	
142	8E		ä	
143	8F		ä	
144	90		E	
145	91		æ	
146	92		æ	
147	93		ö	
148	94		ö	
149	95		ö	
150	96		ü	
151	97		ü	
152	98		ü	
153	99		ö	
154	9A		ö	
155	9B		ö	
156	9C		ç	
157	9D		ç	
158	9E		ç	
159	9F		f	
160	A0		ä	
161	A1		ï	
162	A2		ö	
163	A3		ü	
164	A4		ñ	
165	A5		ñ	
166	A6		ñ	
167	A7		ñ	
168	A8		ñ	
169	A9		ñ	
170	AA		ñ	
171	AB		ñ	
172	AC		ñ	
173	AD		ñ	
174	AE		ñ	
175	AF		ñ	
176	B0		ñ	
177	B1		ñ	
178	B2		ñ	
179	B3		ñ	
180	B4		ñ	
181	B5		ñ	
182	B6		ñ	
183	B7		ñ	
184	B8		ñ	
185	B9		ñ	
186	BA		ñ	
187	BB		ñ	
188	BC		ñ	
189	BD		ñ	
190	BE		ñ	
191	BF		ñ	
192	C0		l	
193	C1		l	
194	C2		l	
195	C3		l	
196	C4		l	
197	C5		l	
198	C6		l	
199	C7		l	
200	C8		l	
201	C9		l	
202	CA		l	
203	CB		l	
204	CC		l	
205	CD		l	
206	CE		l	
207	CF		l	
208	D0		l	
209	D1		l	
210	D2		l	
211	D3		l	
212	D4		l	
213	D5		l	
214	D6		l	
215	D7		l	
216	D8		l	
217	D9		l	
218	DA		l	
219	DB		l	
220	DC		l	
221	DD		l	
222	DE		l	
223	DF		l	
224	E0		α	
225	E1		β	
226	E2		γ	
227	E3		δ	
228	E4		ε	
229	E5		ζ	
230	E6		η	
231	E7		θ	
232	E8		ι	
233	E9		κ	
234	EA		λ	
235	EB		μ	
236	EC		ν	
237	ED		ξ	
238	EE		ο	
239	EF		φ	
240	F0		χ	
241	F1		ψ	
242	F2		ω	
243	F3		⋈	
244	F4		⋈	
245	F5		⋈	
246	F6		⋈	
247	F7		⋈	
248	F8		⋈	
249	F9		⋈	
250	FA		⋈	
251	FB		⋈	
252	FC		⋈	
253	FD		⋈	
254	FE		⋈	
255	FF		⋈	

\* ASCII-Code 127 weist den Code Entf. auf. In MS-DOS hat dieser Code dieselbe Wirkung wie ASCII 8 (RS).  
Der Entf-Code kann durch Drücken von Strg+Rückschritttaste generiert werden.