Programmieren

in

C

Manuskript zum Kurs "Programmieren in C"

Fachrichtung: Wirtschaftsinformatik

Verfasser: Haas

Auflage: 5. Auflage, Stand Januar 2000

Inhalt:

Inhalt:

3. Programmiertechnik und Programmerstellung

Jedes Progamm läßt sich in kleinere logisch geordnete Blöcke unterteilen. Diese sollten für sich eine Einheit bilden und allein lauffähig sein. Bei der Erstellung eines Progamms ist i.a. wie folgt vorzugehen:

1. **Aufgabendefinition**
   Jede Programmiertätigkeit beginnt mit der präzisen Definition der Aufgabe.

2. **Reihenfolgeplanung**
   Struktur und Abfolge der Befehlsschritte in Form eines Struktogramms oder Ablaufplanes entwickeln.

3. **Codierphase**
   Befehle der entsprechenden Programmiersprache in Anlehnung an das erstellte Struktogramm aufschreiben, bzw. Programmtext erstellen.

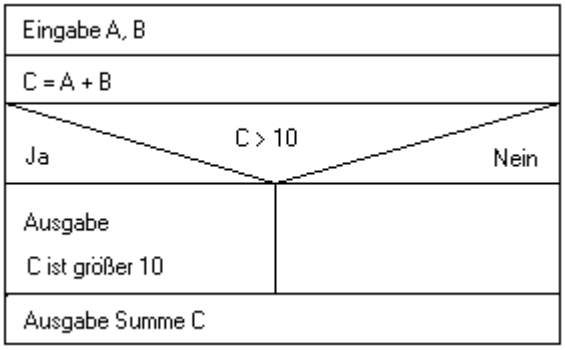
4. **Test und Beseitigung evtl. formaler und logischer Fehler.**

3.1 Das Struktogramm

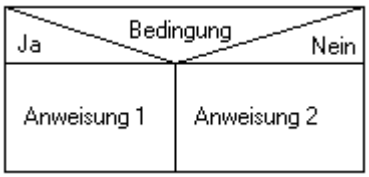
Struktogramme sind sehr gut geeignet strukturierte Programme in einer symbolischen Schreibweise darzustellen. Sie werden immer von oben nach unten und von links nach rechts gelesen.

Beispiel: 2 Zahlen sollen addiert werden. Ist das Ergebnis gröáer als 10, so ist eine entsprechende Meldung zusätzlich zur Summe der beiden Zahlen auszugeben. Es soll ein Struktogramm erstellt werden.

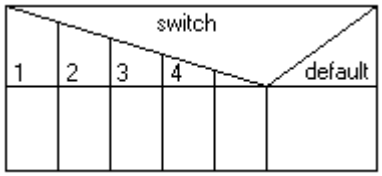
Struktogramm Beispiel:



3.2 Bedingungs-Block

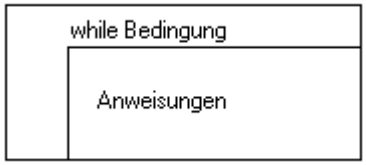


3.3 Auswahl-Block



3.4 Schleifen-Block

(a) While-Schleife



(b) Do While-Schleife

4.5 Aufbau eines Programms / Formale Schreibweise

(1) Programmkopf
(2) Deklarationsteil - Konstanten, Makros - Typen, Strukturen, globale Variablen - Prototypdefinition
(3) Funktionen
(4) Hauptprogramm (main)

Jedes Programm erh„lt einen sogenannten Progammkopf. Hier werden Daten wie Datum der Erstellung, Funktionsbeschreibung, Schnittstellen, Name des Erstellers und sonstige wichtige Informationen zum jeweiligen Programm eingetragen.

In C ist es möglich das Format des entsprechenden Programmes völlig frei zu gestalten. Daher ist es zweckmäßig sich gleich am Anfang eine übersichtliche und gut lesbare Form bei der Erstellung der Quelltexte anzugewöhnen. Man wird hier i.a. die Blockstruktur durch Einrücken der Befehle um mind. 2 Stellen realisieren.

Da in C zwischen Groß -und Kleinschreibung unterschieden wird, müssen C-Kommandos immer klein geschrieben werden, damit der Compiler die Standardbefehle interpretieren kann. Konstanten werden möglichst groß geschrieben. Variablen können groß oder klein oder eine Kombination aus Groß- und Kleinschreibung sein, wenn dies der Verdeutlichung dient, ebenso bei den Funktionen.

Weiterhin sollte man auch die Möglichkeit ausnutzen, Kommentare in den Progammtext mit einzubauen. Auch diese Mühe lohnt sich, wenn man z. B. bei größeren Projekten nach Fehlern suchen muß.

Kommentaranfang	/*
Kommentarende	*/

z.B: /* Dies ist ein Kommentar */

Kommentare können sich auch über mehrere Zeilen erstrecken (siehe Programmkopf).

Bei größeren Projekten empfiehlt sich eine Modularisierung der Programmkomponenten. Dadurch wird das Projekt in mehrere kleine funktionsgemäß zusammengehörende Module unterteilt, die dann einzeln compiliert und getestet werden. Zum Schluß werden alle Module zu einem lauffühigen Programm gebunden.

Ein weiteres gutes Hilfsmittel zur übersichtlichen Programmerstellung ist die objektorientierte Programmierung. Mit dieser Programmiermethode kann Programmcode durch Vererbung effizient genutzt werden, was sich zum Teil erheblich in der Programmröße auswirkt. Weiterhin verbessert sich die Gesamtstruktur, was sich auf die Fehleranfälligkeit der Programme auswirkt.

4. Einführung in die Programmiersprache C

4.1 Geschichtliche Entwicklung, Entstehung

Vorläufer von C war zunächst die Programmiersprache BCPL, die von Martin Richards speziell zur Erstellung von Compilern und Betriebssystemen entwickelt wurde. 1970 entstand daraus durch Ken Thomson die Sprache B. Daraus wurde 1972 die Programmiersprache C. Sie wurde zunächst in den Bell Laboratories von Dennis M. Ritchie entwickelt. Er schrieb diese Sprache um das Betriebssystem UNIX in einer Hochsprache neu zu schreiben um so ein hohes Maß an Portabilität zu erreichen. 1973/74 wurde C von Brian W. Kernigham weiter verbessert.
Bis heute wurde C ständig weiterentwickelt und optimiert und wird in nahezu allen Bereichen der Softwareentwicklung eingesetzt, technische wie auch komerzielle Softwarepakete werden in C geschrieben. Vor allem bei Anwendungen, die unter UNIX, WINDOWS, WINDOWS NT oder OS/2 laufen wird vorwiegend C eingesetzt, da alle Betriebssysteme auch in C geschrieben sind. Durch die Einführung von C++ können auch objektorientierte Strukturen unter C verwirklicht werden.

4.2 Zahlensysteme

Das Dezimalsystem ist unser gebräuchlichstes Zahlensystem, mit dem wir im täglichen Leben umgehen. Ein Computer dagegen kennt nur 2 Zustände (0 oder 1) und kann daher nur mit dem Binärsystem rechnen.
Um die internen Zusammenhänge eines Rechners und deren Datenspeicherung zu verstehen, werden im folgenden die in der EDV gebräuchlichsten Zahlensysteme kurz erläutert.

- Binärsystem (Basis 2)
- Oktalsystem (Basis 8)
- Dezimalsystem (Basis 10)
- Hexadezimalsystem (Basis 16)

Die einzelnen Zahlensysteme unterscheiden sich nur durch ihre Basis: 2,8,10,16. Beim Hexadezimalsystem werden, da die Ziffern nicht mehr ausreichen, zusätzlich noch die Buchstaben A-F (entspricht 10-15) verwendet.

Beispiel:	im Dezimalsystem:	123	=	$1*10^2$	$+ 2*10^1$	$+3*10^0$	
				100	+ 20	+3	= 123
	im Binärsystem:	10	=	$1*2^2$	$+ 1*2^1$	$+ 0*2^0$	
				4	+ 2	+ 0	= 6
	im Hexadezimalsystem:	A5	=	$A*16^1$	$+5*16^0$		
				160	+ 5		= 165

4.3 Speichern von Informationen, interne Darstellungen

Die kleinste Einheit eines Rechners ist eine Speicherstelle und diese wird als Bit bezeichnet. Ein Bit kann lediglich die Informationen 0 oder 1 tragen. Somit ist auch die Bedeutung des Binärsystems und der Zusammenhänge zu anderen Zahlensystemen verständlich.

Die nächst größere Einheit ist das Byte, es umfaßt 8 Bit. Das Byte ist eine wichtige Einheit in der Datenverarbeitung.

Ganzahldatentypen:

char	0 0000000	= 8 Bit	= 1 Byte
int	0 0000000 00000000	= 16 Bit	= 2 Byte
long int	0 0000000 00000000 00000000 00000000	= 32 Bit	= 4 Byte

Vorzeichenbit

Fließkommadatentypen:

float	Vorzeichen(1)	Exponent(8)	Mantisse(23)	= 32 Bits = 4 Bytes
double	Vorzeichen(1)	Exponent(11)	Mantisse(52)	= 64 Bits = 8 Bytes
long double	Vorzeichen(1)	Exponent(15)	Mantisse(112)	= 128 Bits = 16 Bytes

(x) : x=Anzahl Bits

Exponent normalisiert:	float	127	(7Fh)
	double	1023	(3FFh)
	long double	16383	(3FFFh)

4.4 Datenformate in C

Zur Speicherung der Daten stehen in C zahlreiche Datentypen zur Verfügung.

Ganzzahldatentypen:

Datentyp	**Wertebereich**	**AIX**	**DOS**	**Verwendung**
char	128..127	1	1	kleine Zahlen und ASCII-Zeichen
unsigned char	0..255	1	1	kleine Zahlen und ASCII-Zeichen
short	-32768..32767	2	2	Ganzzahlen mit Vorzeichen
unsigned short	0..65535	2	2	Ganzzahlen ohne Vorzeichen
int	-32768..32767	4	2	Ganzzahlen mit Vorzeichen
unsigned int	0..65535	2	4	Ganzzahlen ohne Vorzeichen
long	-2147483648..2147483647	4	4	große Ganzzahlen mit Vorzeichen
unsigned long	0..4294967296	4	4	große Ganzzahlen ohne Vorzeichen

Fließkommadatentypen:

float	$3.4*10\text{-}38..3.4*10^{+38}$	4	4	Fließkomma, 7-stellige Genauigkeit
double	$1.7*10\text{-}308..1.7*10^{+308}$	8	8	Fließkomma, 15-stellige Genauigkeit
long double	$3.4*10\text{-}4932..3.4*10^{+4932}$	16	10	Fließkomma, 19-stellige Genauigkeit

Strings
Ein String ist eine Folge von Zeichen, die alle vom Typ char oder unsigned char sind.

Sonstige Datentypen

Zeiger (Pointer) Adresse, die auf einen Datenbereich zeigt.

void spezieller Typ-Spezifizierer, der anzeigt, dass kein Wert vorhanden ist. Er wird bei Funktionen eingesetzt, die keinen Rückgabewert oder keine Parameter haben (siehe Funktionen). Weiterhin kann void als universeller bzw. typloser Zeiger verwendet werden.

struct definiert Strukturen. Eine Struktur ist ein abgeleiteter Typ der es erlaubt Variablen unter-schiedlicher Datentypen unter einem neuen Typnamen zusammenzufassen (siehe Strukturen).

typedef Datentypnamen können beliebig verändert werden.

enum Aufzählungstyp. Mit diesem Datentyp kann eine Liste von Daten erstellt werden, denen fortlaufende oder auch beliebige konstante Werte zugewiesen werden.
Z.B. *enum wochentag { montag, dienstag, mittwoch };*
Montag bekommt den Wert 0, dienstag 1, Mittwoch 2 usw.

4.5 Deklaration von Variablen

Variablen sind Speicherbereiche, die mit einem eindeutigen Namen und einem Datentyp verknüpft sind. Für die Namensgebung einer Variable sind folgende Regeln zu beachten:

- Jede Variable muß mit einem Buchstaben oder einem Unterstrich beginnen. Weitere Zeichen sind Ziffern oder/und Buchstaben, nicht aber Sonderzeichen und Umlaute.Z.B. weg, x1, geschwindigkeit,Test_01.

- Es wird zwischen Groß- und Kleinschreibung unterschieden z.B. ist Weg nicht gleichbedeutend mit weg.

- Gründen der besseren Lesbarkeit sollten lange Variablennamen vermieden werden.

- folgende von C verwendete Schlüsselwörter dürfen nicht als Variablennamen verwendet werden:

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

- Variablennamen sollten immer so gewählt werden, dass sie ein hohes Maß an Aussagefähigkeit besitzen. Z.B. sagt der Varaiblenname vorname mehr aus als xxxx. Dadurch wird Ihr Quelltext leichter lesbar.

- Jede Variable muß vor dem Gebrauch deklariert werden !

Syntax: ***Datentyp Variable[=Ausdruck];***

Beispiele:
```
int   i,j,k;
float x;          /* Fließkommazahl */
char  zeichen;    /* einzelnes Zeichen */
char  st[10];     /* String mit max. 9 Zeichen */
long  Test_Eins;
```

Variablen können bei ihrer Deklaration auch Werte zugewiesen werden. Dadurch ist eine Initialisierung möglich.

Beispiele:
```
int     i=2,j=5,k=i;
long    x=1234567890;
char    st[20]="Thomas";
int     a[5]={1,2,3,4,5};
float   sum=10.234;
```

4.5.1 Wertzuweisung

einfache Zuweisung mit dem Zuweisungsoperator =

Syntax: ***Variable=Ausdruck;***

Der Variablen Variable wird der Wert des Ausdruckes Ausdruck zugewiesen.

Beispiele:
```
i=5;
f=4.56*45;
b[2]=3;
```

In C sind auch Kettenzuweisungen erlaubt:

Syntax: ***Variable1=Variable2=...Variablen=Ausdruck;***

Der rechts stehende Ausdruck wird allen links stehenden Variablen zugewiesen.

Beispiele:
```
i=j=k=0;
f=g=1.23;
```

4.6 Typumwandlung

In C können Datentypen mit Hilfe des Cast-Operators "(Datentyp)" in andere Datentypen umgewandelt werden.Treten in einem Ausdruck unterschiedliche Datentypen auf, also z.B. int, char, float usw., dann wird immer der schwächere Typ in den stärkeren umgewandelt. Z.B. wird int und long in long umgewandelt.Findet eine Zuweisung statt, so wird das Resultat des rechts vom Zuweisungsoperator stehenden Ausdrucks in den Typ der links vom Zuweisungsoperator stehenden Variablen umgewandelt.

Syntax: ***(Datentyp)Ausdruck***

Beispiel:
```
int     a;
float   x;

x=(float)a;     /* int in float umwandeln */
```

4.7 Operatoren in C

4.7.1 Arithmetische Operatoren

+	Addition
-	Subtraktion
++	Inkrementieren (x=x+1, x++, ++x)
--	Dekrementieren (x=x-1, x--, --x)
*	Multiplikation
/	Division
%	Modulo (Rest einer Ganzzahldivision)

Beispiele:
```
x=y+10;
++x-20;
x++*10;
a=10%2;
```

4.7.2 Vergleichsoperatoren

==	gleich
!=	ungleich
>	größer
>=	größer gleich
<	kleiner
<=	kleiner gleich

Beispiele:
```
if(a==b) printf(...);
x=a!=b;
x=a>=b;
```

4.7.3 Logische Operatoren

!	Negation
&&	UND-Verknüpfung
||	ODER-Verknüpfung

Besipiele:
```
IF(!a) b=2;
x=a&&b;
if((a==2)||(a==5)) printf(...);
```

Ein Wert ungleich Null bedeutet "true" (wahr), gleich Null bedeutet "false" (falsch).!a ist gleichbedeutend mit a==0.

4.7.4 Operatoren zur Bit-Manipulation

&	bitweise UND
|	bitweise ODER
^	bitweise Exclusive-Oder-Verknüpfung
<	Bit-Verschiebung nach links
>>	Bit-Verschiebung nach rechts
~	Bilden des Einerkomplements

Beispiele:
```
a=x&2;
x=x|2;
x=5^2;
x=x<<2;
a=~x;
```

4.7.5 Zuweisungsoperatoren

Mit Hilfe der Zuweisungsoperatoren kann eine Zuweisung mit einem Operator (+,-,*,/ usw.) kombiniert werden.

Syntax***:*** ***E1 op= E2;*** entspricht E1 = E1 op E2;

So ist z.B. E1+=10 gleichbedeutend mit E1=E1+10. Der Variablen E1 wird hierbei der Wert 10 hinzuaddiert. E1 muß dabei eine modifizierbare Variable sein. op steht für einen der folgenden Operator-Zeichen:

*, %, +, -, <<, >>, &, ^, |

Beispiele:
```
x*=4;      /* x wird mit 4 multipliziert und das Ergebnis  in x gespeichert */
x/=4;      /* x wird durch 4 dividiert */
x-=2;      /* von x wird 2 subtrahiert */
x<<=2;     /* x wird um 2 Bitpos. nach links verschoben */
x|=1;      /* Bit 0 wird in x gesetzt */
```

4.7.6 Sonstige Operatoren

&	Adreßoperator
*	Indirektionsoperator
sizeof	belegter Speicher eines Operanten ermitteln
.	Mitgliedoperator für Strukturen
->	Verweisoperator für Strukturen
()	Funktionsaufrufoperator, bzw. Cast-Operator
[ ]	Arrayindizierungsoperator

Beispiele:
```
int *x;  x=&a;  *x=2;
x=sizeof(long);
printf("%s\n",adr.name);
printf("%s\n",adr->name);
int  a[100];  a[5]=10;
```

4.8 Header-Dateien, Objektdateien und Libraries

Bei der Arbeit mit C stehen dem Anwender einige hundert Funktionen und Makros zur Verfügung, die innerhalb von C-Programmen aufgerufen werden können und die verschiedensten Aufgaben erfüllen. Ein/Ausgaben auf Bildschirm, Drucker und Datei, mathematische Berechnungen, Stringmanipulation und vieles mehr vereinfachen die Programmierung ganz erheblich. All diese Funktionen sind in einer sog. Library enthalten. Der Programmcode dieser Libraries ist aber ein bereits compilierter Objektcode, der vom Anwender nicht beeinflußt werden kann. Dieser Code wird beim Linken eines Programms automatisch gebunden. Libraries können auch vom Anwender selbst erstellt werden. Dadurch läßt sich ein Programmsystem in funktional zusammenhängende Modulkomponenten zerlegen.

Header-Dateien sind sozusagen die Schnittstellen zu den Libraries. In ihnen sind alle Funktionsprototypen (Funktionskopf mit Parametern) symbolische Konstanten, Datentypen und globale Variablen enthalten. Header-Dateien haben immer die Dateiendung ".H".

Um eine Library-Funktion zu nutzen muß die entsprechende Header-Datei in das Programm eingebunden werden. Dies erfolgt mit dem Präprozessorkommando #include.

Syntax:	***#include***	***<name.h>***	suchtin dem Standardincludeverzeichnis /usr/include
	#include	***"name.h"***	sucht im aktuellen Verzeichnis

Im folgenden wird eine šbersicht der im ANSI C-Standard vorgeschriebenen Header-Dateien dargestellt:

assert.h	(assertion=Forderung) Prüfungen eines Programms
ctypes.h	(character type=Zeichentypen) Definiert Makros zur Klassifizierung von Zeichen (Tests auf Buchstabe, Ziffer usw.) und Funktionen zur Umwandlung einzelner Zeichen.
errno.h	(error number=Fehlernummer) Definiert symbolische Konstanten für Fehlernummern und Fehlermeldungen des Systems.
float.h	(floating point=Fließkomma) Definiert symbolische Konstanten für die Fieißkomma-Routinen.
limits.h	(limits=Grenzen) Definiert die Grenzen der Ganzzahldatentypen sowie des Compilers.
locale.h	(locale environment=ortsübliche Umgebung) Deklariert Funktionen, die länderspezifische Informationen liefern.
math.h	(mathemathics) Deklariert höhere mathematische Fuktionen.
setjmp.h	(set jump=Sprünge) Funktionen für nichtlokales goto.
signal.h	(signal) Definiert signal und raise.
stdarg.h	(standard arguments=Standard-Parameter) Definiert Makros und Funktionen für variable Parameter.
stddef.h	(standard definitions) Definiert allgemein verwendete Datentypen und Makros.
stdio.h	(standard input/output stream=Standard Eingabe/Ausgabe-Kanäle) Definiert grundlegende Datentypen und Makros für standardisierte Ein/Ausgaben (nach Kerigham & Ritchie).
stdlib.h	(standard library) Deklariert diverse allgemeine Funktionen zur Konvertierung, zum Durchsuchen und Sortieren von Arrays usw.
string.h	(string) Deklariert Funktionen zur Manipulation von Strings und Speicherbereichen.
time.h	(time=Datum/Uhrzeit) Definiert Strukturen und Funktionen zur Bearbeitung von Datum und Uhrzeit.

Eine Beschreibung der wichtigen Funktionen einiger dieser Header-Dateien wird unter den folgenden Kapiteln durchgeführt.

4.9 Das Hauptprogramm main

C ist eine strukturierte Programmiersprache, d.h. ein Programm besteht aus mehreren Modulen, die in C als Funktionen bezeichnet werden. Jede Funktion hat ihren eigenen Namen. Viele dieser Funktionen sind in den Libraries enthalten. Jedes C-Programm muß aber mindestens eine Funktion enthalten, die Hauptfunktion main(). Diese Funktion kennzeichnet den Beginn eines C-Programms. Von main() werden dann wiederum weitere Funktionen aufgerufen. Die Hauptfunktion ist wie folgt zu definieren:

```
void main()
{
  ...
}
```

Die geschweiften Klammern schließen die Anweisungen von *main* oder generell aller Funktionen ein. Dabei ***{*** ist der Blockbeginn und ***}*** das Blockende.

4.9.1 Die geschweiften Klammern { }

markieren den Beginn und das Ende eines Anweisungsblockes. Sie werden überall dort verwendet, wo eigentlich die C-Syntax nur eine Anweisung zuläßt.

4.9.2 Das Semikolon ;

Jede Anweisung muß in C mit einem Semikolon (;) abgeschlossen werden. Z.B. a=5;

4.10 Ein- und Ausgabe

Um eine Kommunikation mit dem Anwender oder mit periferen Geräten zu ermöglichen stellt C Funktionen für Ein- und Ausgabe von Daten zur Verfügung. In diesem Kapitel werden nur Funktionen zur Ein- und Ausgabe auf dem Terminal behandelt. Ein- und Ausgaberoutinen, die auch Dateien auf Festplatten mit einbeziehen, werden wir in späteren Kapiteln kennenlernen.

ESC-Steuerzeichen

Innerhalb von Ausgabestrings können verschiedene Steuerzeichen verwendet werden (z.B Zeilenvorschub, Signalton usw.). Diese bestehen aus einem sog. Fluchtsymbol und einem nachfolgenden Buchstaben.

\a	Signalton	\b	Backspace (Rücktaste)
\f	Formfeed (Seitenumbruch)	\n	Zeilenvorschub
\r	Carriage Return	\t	horizontaler Tabulator
\v	vertikaler Tab	\\	Backslash
\'	Apostroph	\"	Anführungszeichen
\?	Fragezeichen	\OOO	oktaler Wert
\xHHHH	Hexadezimaler Wert		

4.10.1 Ausgabe

printf

printf ist eine sehr mächtige Funktion zur formatierten Ausgabe von Daten auf dem Standardausgabegerät (Bildschirm). Das Ausgabeformat und die Anzahl der Ausgaben sowie ihr Format werden in einem sog. Formatstring festgelegt. Anschließend folgen die Daten selbst in der gleichen Reihenfolge ihrer Definition. Z.B. printf("Summe: %d+%d=%d\n",a,b,s);

Definition in: stdio.h
Syntax: ***int printf (const char *format [,item[,item ...]);***

Der Format-String kann normale Zeichen enthalten (werden direkt ausgegeben) und auch Konvertierungs-Anweisungen. Diese werden immer mit einem %-Zeichen eingeleitet.

% [Flags] [Breite] [.Präzision] [h|l|L] Typ

Flags
eine Zeichenfolge über numerische Vorzeichen, Dezimalpunkte, führende und folgende Nullen, links- und rechtsbündige Darstellung.

-	linksbündige Ausgabe
+	numerische Ausgabe mit Vorzeichen
blank	positive Werte mit führendem Blank ausgeben

Breite
minimale auszugebende Zeichenzahl.

n	gibt mind. n Zeichen aus und stellt ggf. Leerzeichen voran
0n	gibt mind. n Zeichen aus und stellt ggf. Nullen voran
*	die Breite steht in der Parameterliste direkt vor dem auszugebendem Wert als Zahlenwert. Dadurch ergibt sich die Möglichkeit die Feldbreite zur Laufzeit festzulegen.

.Präzision
maximale Anzahl von Dezimalstellen.

.n	Ausgabe von n signifikanten Dezimalstellen

h|l|L - explizite Größenangaben

h	Argument wird als short interpretiert (d,i,o,u,x,X)
l	wird als long (d,i,o,u,x,X) bzw. double (e,E,f,g,G) interpretiert
L	wird als long double interpretiert (e,E,f,g,G)

Typ
auszugebender Datentyp.

numerische Werte:	d	signed int (dezimal)
	o	unsigned int (oktal)
	u	unsigned int (dezimal)
	x	unsigned int (hexadezimal, a..f)
	X	unsigned int (hexadezimal, A..F)
	f	Fließkomma [-]dddd.dddd
	e	Fließkomma im Exponetialformat [-]d.dddd e[+|-]ddd
	g	Fließkomma im f oder e Format. e-Format wird dann verwendet, wenn f-Konvertierung mehr als Präzision Stellen ergibt.

	E	Fließkomma wie e, aber großes E. [-].d.dddd E[+|-]ddd
	G	wie g aber großes E beim Exponentialformat
Zeichen:	c	einzelnes Zeichen
	s	String
	%	Ausgabe des %-Zeichens selbst

Rückgabe: Anzahl der ausgegebenen Zeichen oder EOF (-1) bei Fehler.

Beispiele:
```
printf("%d\n");          /* Ganzzahl int mit Vorzeichen */
printf("%+7.2f\n");      /* Fließkomma mit Vorzeichen, 7 Stellen, rechtsbündig,
                         /* 2 Nachkommastellen *
printf("%7ld\n");              /* Ganzzahl long rechtsbündig mit 7 Stellen */
printf("%s\t%s"\n");     /* String Tabulator String  */
```

sprintf

Ausgabe erfolgt in einen Stringpuffer und nicht auf der Standardausgabe. Ansonsten gelten alle Regeln von printf.

Definition: stdio.h
Syntax: ***int sprintf(char *buf, const char *format[,...]);***

Beispiel:
```
#include <stdio.h>          /* Umwandlung int -> string mit sprintf*/
main()
{  char       st[20];
   int        i=20;

    sprintf(st,"%0d5",i);
    printf("%s\n",st);
}
```

putchar

Ausgabe eines einzelnen Zeichens.

Definition: stdio.h
Syntax: ***int putchar (int c);***

Beispiel:
```
#include <stdio.h>          /* Zeichen ausgeben mit putchar */

main()
{   int        i=65;
    char       c='B';

    putchar(i);             /* Ausgabe AB */
    putchar(c);
    putchar('\n');          /* Zeilenvorschub */
}
```

puts

Ausgabe eines Strings mit anschließendem Zeilenvorschub.

Definition: stdio.h
Syntax: int puts (const char *s);

Beispiel:
```
#include <stdio.h>          /* String mit puts ausgeben */

main()
{   char      st[9]="Autobahn";

    puts(st);               /* Ausgabe "Autobahn"  und Zeilenvorschub */
}
```

4.10.2 Eingabe

scanf

Eingaben in formatierter Form vom Standardeingabegerät (Tastatur) lesen.

Definition: stdio.h
Syntax: ***int scanf(const char *format[,item[,item]...]);***

Anmerkung: item muß immer eine Adresse (Zeiger) sein, evtl. &-Operator verwenden um eine Adresse zu erzeugen (siehe Beispiele).

Über den Format-String wird die Art der Konvertierung und die Anzahl der Parameter festgelegt. Überzählige Parameter werden ignoriert. Leerzeichen trennen Eingaben voneinander.

% [Breite] [|h|l|L] Typ

Breite
maximal zu lesende Zeichenzahl.

h|l|L - explizite Größenangaben

h	Argument wird als short interpretiert (d,i,o,u,x)
l	wird als long (d,i,o,u,x) bzw. double (e,E,f,g) interpretiert
L	wird als long double interpretiert (e,E,f,g)

Typ
einzugebender Datentyp

numerische Werte:	d	Integer (dezimal)
	o	Integer (oktal)
	u	vorzeichenloser Integer (dezimal)
	x	Integer (hexadezimal, a..f)
	f	Fließkomma [-]dddd.dddd
	e	Fließkomma im Exponetialformat [-]d.dddd e[+|-]ddd
	g	Fließkomma im f oder e Format

Zeichen: c einzelnes Zeichen
s String

Rückgabe: Anzahl der fehlerfrei gelesenen Felder oder EOF beim Versuch über das Ende eines Strings hinauszuschreiben.

Beispiele:
```
scanf("%d  %c %f",&zahl,&zeichen,&kommazahl);
scanf("%s  %s",st1,st2);
```

getchar

Einzelnes Zeichen von Standardeingabe lesen.

Definition: stdio.h
Syntax: ***int getchar(void);***

Beispiel:
```
#include <stdio.h>          /* Zeichen mit getchar von Standardeingabe lesen */
msin()
{  int   wahl,ret;

   printf("Bitte ein Zeichen eingeben:");
   wahl=getchar();                          /* Zeichen      einlesen */
   ret=getchar();                           /* Return einlesen */
   printf("%d  %d\n",wahl,ret);
}
```

gets

Eingabe eines Strings. Die Eingabe wird durch einen Zeilenvorschub (Return) beendet.

Definition: stdio.h
Syntax: ***char *gets(char *s);***

Beispiel:
```
#include <stdio.h>          /* String von Standardeingabe mit gets lesen */
main()
{  charst[20];
   printf("Bitte einen String eingeben:");
   gets(st);                /* String einlesen */
   puts(st);                /* String wieder ausgeben */
}
```

4.10.3 Bildschirmsteuerung

In einigen C-Versionen (z.B. Turbo C++ von Borland) sind spezielle Funktionen zur Bildschirmsteuerung verfügbar. Ist dies allerdings nicht der Fall, dann können sog. ESC-Steuersequenzen verwendet werden um z.B. den Bildschirm zu löschen, den Cursor zu positionieren, usw. ESC-Steuersequenzen werden durch eine Ausgabeanweisung (z.B. printf) an das Standardausgabegerät (i.a. der Bildschirm) übertragen. Dieses interpretiert die Steuerzeichen und führt anschließend die entsprechende Aktion aus. Alle Steuersequenzen werden mit dem ESC-Zeichen (ASCII 27) eingeleitet. Darauf folgen ein oder mehrere Zeichen, die einer entsprechenden Funktion zugeordnet sind.

Beispiel:
```
printf("\33J");      Bildschirm Löschen
printf("\33A");      Cursor nach oben
```

4.11 Auswahl-Anweisungen

Auswahl-Anweisungen wählen aus alternativen Aktionsläufen aus. Dies geschieht durch das Testen von bestimmten Werten. Es gibt zwei Arten von Auswahl-Anweisungen:

4.11.1 If-Anweisung

Syntax:
```
if(bedingung)
   Anweisung1;
else
   Anweisung2;
```

Ist der Ausdruck *bedingung* erfüllt (true=1), wird Anweisung1 bearbeitet, ist *bedingung* hingegen nicht erfüllt (false=0), dann wird Anweisung2 ausgeführt. Die If-Anweisung kann auch ohne die Alternative else verwendet werden.

Syntax:
```
if(bedingung)
   Anweisung1;
```

Anweisung1 wird dann ausgeführt, wenn *bedingung* erfüllt ist, sonst wird Anweisung1 übersprungen und die auf *Anweisung1* fogenden Anweisungen ausgeführt. Sollen mehrere Anweisungen zu einer if-Anweisung ausgeführt werden, so müssen diese in geschweifte Klammern eingeschlossen werden.

Beispiel:
```
#include <stdio.h>              /* Vergleich mit if */
main()
{   char    op;

    op=getchar();
    if(op=='+')
      puts("Addieren");
    else
      if(op=='-')
        puts("Subtrahieren");
      else
        puts("Falsche Eingabe");
}
```

4.11.2 Die switch-Anweisung

Die switch-Anweisung ermöglicht die direkte Auswahl definierter Anweisungsblöcke. Der entsprechende Anweisungsblock wird durch einen Wert bestimmt, der der switch-Anweisung übergeben wird.

Syntax:
```
switch(ausdruck)
{
   case      ausdruck1:   Anweisung1;      break;
   case      ausdruck2:   Anweisung2;      break;
   default:               Anweisung3;      break;
}
```

Je nach dem Wert von *ausdruck* wird bei der Übereinstimmung eines entsprechenden *case*-Ausdrucks der passende Anweisungsblock ausgeführt. Trifft keine der *case*-Ausdrücke zu, werden die Anweisungen nach *default* abgearbeitet. *default* kann auch entfallen.

Die *break*-Anweisung

Syntax: ***break;***

Mit der *break*-Anweisung können *switch*-Anweisungen unterbrochen werden, d.h. dass die switch-Anweisung unmittelbar verlassen und die ihr folgenden Anweisungen ausgeführt werden. Ebenso können auch Schleifen mit der *break*-Anweisung abgebrochen werden (siehe 4.12).

Würde z.B. *ausdruck* mit *ausdruck1* übereinstimmen und die *break*-Anweisungen nicht vorhanden sein, dann wird nicht nur *Anweisung1* ausgeführt, sondern auch *anweisung2* und *anweisung3*. Somit muß also jeder *case*-Anweisungsblock mit einer *break*-Anweisung beendet werden.

Beispiel:
```
#include <stdio.h>                    /* Auswahl mit switch */

main()
{   char   op;

    op=getchar()
    switch(op)
    {
       case '+':
          puts("Addieren");
          break;
       case '-':
          puts("Subtrahieren");
          break;
       default:
          puts("Falsche Eingabe");
          break;
    }
}
```

4.12 Schleifen

Schleifen oder Wiederholungsanweisungen werden verwendet, wenn eine oder mehrere Anweisungen mehrmals hintereinander ausgeführt werden sollen. Hierzu stehen in C drei verschiedene Schleifenarten zur Verfügung:

4.12.1 for-Schleife

Syntax:
```
for (Init; Test; Inkrement)
   Anweisung;
```

Die *for*-Anweisung besteht aus drei Teilen:

Init
Im Init-Teil wird der Schleifenvariablen (Laufvariablen) ein Startwert zugewiesen z.B. i=0;. Es können auch mehrere Variablen initialisiert werden. Dazu müssen die einzelnen Zuweisungen durch Kommas getrennt werden.

Test
Im Test-Teil wird ein Ausdruck ausgewertet. Ergibt dieser einen Wert der ungleich 0 (true) ist, wird die Schleifenanweisung ausgeführt. Andernfalls wird die Schleife beendet. Z.B. i<10, die Schleife wird beendet, wenn i=10 ist, da der Ausdruck dann nicht mehr wahr (true) ist. Ist die Bedingung zum Schleifenbeginn nicht erfüllt, so wird die Schleife nicht durchlaufen.

Inkrement
Im Inkrement-Teil werden ein oder mehrere Variablen erhöht bzw. erniedrigt.

Beispiel:
```
#include <stdio.h>                  /* Schleife mit for */
main()
{   int i,j;

    for(i=0, j=4; i<5; i++, j--)
        printf("%2d     %2d\n",i,j);
}
```

Die break-Anweisung

kann auch in Schleifen verwendet werden. Die Schleife wird dann sofort abgebrochen und die Programmausführung hinter der Schleife fortgesetzt.

Beispiel:
```
#include <stdio.h>                  /* Schleife mit for und break */
main()
{   int          i;

    for(i=0; i<10; i++)
    {
      if(i>5)
        break;
      printf("%2d\n");
    }
}
```

Die Continue-Anweisung

Syntax: ***continue;***

Im Unterschied zur *break*-Anweisung unterbricht *continue* nur den aktuellen Schleifendurchlauf. Es wird also sofort zum nächsten Schleifendurchlauf fortgefahren, sofern eine Abbruchbedingung noch nicht erreicht ist.

Beispiel:
```
#include <stdio.h>
main()
{   int  i;
    for(i=0; i<10; i++)
    {
        if(i%2!=0)
          continue;
        printf("%2d   %2d\n",i,i/2);
    }
}
```

4.12.2 while-Schleife

Syntax: ***while(Bedingung)***
Anweisung;

Die *while*-Schleife wird ausgeführt, solange die Bedingung erfüllt (true) ist. Ist Bedingung schon vor der Ausführung nicht erfüllt, wird die Schleife nicht durchlaufen.

Beispiel:
```
#include <stdio.h>
main()
{   int  i=0;
    while(i<10)
    {
       printf("%5d   %5d   %5d\n",i,i*i,i*i*i);
       i++
    }
}
```

4.12.3 do .. while-Schleife

Syntax:
```
do
   Anweisung;
while(Bedingung);
```

Die *do..while*-Schleife wird ausgeführt, solange die Bedingung erfüllt (true) ist. Da die *do..while*-Schleife die Bedingung nicht vor, sondern nach der Ausführung der Schleifenanweisung testet, wird sie immer mindestens einmal durchlaufen, auch dann, wenn *Bedingung* nicht erfüllt ist.

Beispiel:
```
#include <stdio.h>
main()
{   int  zahl;

    do
    {
       printf("Bitte geben Sie eine Zahl zwischen 0 und 100 ein:");
       scanf("%d",&zahl);
    }
    while((zahl<0)||(zahl>100));
}
```

4.13 Zeiger und Adressen

Der Arbeitsspeicher eines Rechners setzt sich aus einer großen Anzahl von Speicherstellen zusammen. Jede dieser Speicherstellen wird durch eine eigene Adresse gekennzeichnet. Der Inhalt von Speicherstellen kann nur über diese Adressen abgefragt werden. Bei jeder Variablendeklaration (z.B. long x;) wird intern eine Adresse einer Variablen zugeordnet. Es findet also immer eine Umwandlung von einem Variablennamen in eine
Adresse und umgekehrt statt.

Definition des Begriffs "Zeiger":
Ein Zeiger ist eine Variable, die die Adresse einer anderen Variablen enthält (nach Kernighan - Ritchie). Zeigervariablen enthalten somit nur eine Adresse auf Datenbereiche, z.B. Variablen.

Deklaration von Zeigervariablen:
Zur Deklaration einer Zeigervariablen muß wie bei allen Variablen ein Datentyp verwendet werden (z.B. int, float usw.). Jede Zeigervariable wird durch Voranstellen eines * vor dem Variablennamen definiert.

Syntax: ***Datentyp *Variablenname;***

Typlose Zeiger, also Zeiger, die an keinen Datentyp gebunden sind, können mit *void* deklariert werden. Typlose Zeigervariablen können also Daten beliebigen Typs sein.

Syntax: ***void *Variablenname;***

Beispiele:
```
int    *a;        /* Zeiger vom Typ int */
float  *sum;      /* Zeiger vom Typ float */
char   *string;   /* Zeiger vom Typ char */
void   *daten;    /* typloser Zeiger */
```

Dynamische Speicherplatzverwaltung mit Hilfe von Zeigervariablen

Kriterien für die Verwendung von Zeigervariablen

- Speicherplatz soll nur dann belegt werden, wenn er auch wirklich benötigt wird
- unbestimmte Datenmengen, die zum Zeitpunkt der Programmierung noch nicht erkennbar sind

malloc

Fordert Systemspeicher an und gibt einen Zeiger auf den Beginn des Speicherbereiches als Funktionswert zurück.

Definition: stdlib.h
Syntax: ***void *malloc(int size);***

free

Gibt einen mit *malloc* reservierten Speicherblock wieder frei.

Definition: stdlib.h
Syntax: ***void free(void *ptr);***

Beispiele:
```
char *p;
p=(char*)malloc(1024);          /* 1024 Bytes anfordern */

long *x;
x=(long*)malloc(sizeof(long));  /* 4 Bytes anfordern */
```

Zeigerarithmetik

Die Zeigerarithmetik gestattet die Manipulation von Zeigern. Dabei können Addition, Subtraktion und Vergleich auch auf Zeiger angewendet werden. Arithmetische Operationen bei Zeigern vom Typ "Zeiger auf Typ" berücksichtigen automatisch die Größe des Typs typ, d.h. die Anzahl der Bytes, die für das Typ-Objekt notwendig sind.

```
Beispiele:  int *x;                  /* int-Zeiger */
            x=(int*)malloc(100);     /* 100 Bytes Speicher anfordern *)
            *x=5;                    /* 1. Element mit 5 belegen */
            x++;                     /* Zeiger auf 2. Element verschieben */
            *x=12;                   /* 2. Element mit 12 belegen */
            x--;                     /* Zeiger wieder auf 1. Element */
            *(x+2)=30;               /* 3. Element mit 30 belegen */
            x[3]=40;                 /* 4. Element mit 40 belegen */

            char st[]="Teststring"; /* Stringkonstante */
            st+=4;                   /* Zeiger um 4 Zeichen verschieben */
            printf("%s\n",st);       /* Ausgabe von "string" */
```

4.14 Vektoren (Arrays)

C bietet die Möglichkeit, eine größere Anzahl von Variablen, die gleichartige Informationen enthalten, unter einem Vektor(Array) zusammenzufassen. Es muß nicht für jeden Wert eine eigene Variable vereinbart werden, sondern die Werte werden unter einer einzigen Variablen zusammengefaßt. Dabei muß hinter dem Variablennamen, in eckiger Klammer, die Größe des Vektors, also die Anzahl der Datenfelder, angegeben werden.

Syntax: ***Datentyp Vektorname[Anzahl Elemente];***

```
Beispiele:  long   werte[10];
            char   ein_string[1024];
            float  eingabe[20];
            int    tabelle[5]={1,2,3,4,5};
```

Auf die einzelnen Elemente (Felder) des Vektors kann mit dem Indizierungsoperator zugegriffen werden. Dabei muß die Elementnummer in eckigen Klammern hinter der Variablen angegeben werden. Jeder Vektor beginnt bei Index 0. Der maximale Index eines Vektors ist immer n-1. Die Indexnummer stellt sozusagen die Position eines Elementes bezogen auf den Vektorbeginn dar.

```
Beispiele:  a=werte[5];
            werte[2]=a;
            eingabe[2]=eingabe[0];
```

4.15 mehrdimensionale Vektoren (Arrays)

Mehrdimensionale Vektoren werden durch weitere eckige Klammern mit entsprechenden Größenangaben definiert.

```
Syntax:  Datentyp  Vektorname[Größe1][Größe2];          /* 2-dimesional */
         Datentyp  Vektorname[Größe1][Größe2][Größe3]; /* 3-dimensional */
```

4.16 Zeichenketten (Strings)

Eine Zeichenkette (String) besteht aus mehreren Zeichen, wobei jedes einzelne Zeichen 1 Byte Speicher belegt. Um Zeichenketten zu speichern, muß Speicherplatz mit Hilfe von Vektoren oder Zeigern reserviert werden. Zur Bearbeitung von Zeichenketten stehen in C entsprechende Funktionen zur Verfügung. Alle Stringfunktionen beginnen mit den Zeichen "str". Sie sind alle in der Header-Datei "string.h" definiert, welche mit ***#include <string.h>*** in das entsprechende Programm eingebunden werden muß.

Interner Aufbau eines Strings

T e s t s t r i n g \0

Jeder String benötigt Anzahl Zeichen + 1 Byte Speicher, da das Ende eines Strings immer mit einem zusätzlichen Zeichen \0 gekennzeichnet ist, womit das Ende eines Strings gekennzeichnet wird. Dies ist auch bei der Deklaration von Strings zu beachten. Soll z.B. ein String für max. 10 Zeichen zur Verfügung gestellt werden, so muß tatsächlich ein String mit 11 Zeichen deklariert werden. Also 10 Zeichen + \0 = 11 Zeichen.

Deklaration

Wie oben schon erwähnt ist ein String ein Vektor vom Typ char und genauso wird dieser auch deklariert:

Syntax: ***char stringname[stringlaenge];***

Beispiele:
```
char   name[20];      /* String mit 19 Nutzzeichen + 0 */
char   vorname[15];   /* String mit 14 Nutzzeichen + 0 */
```

Stringkonstanten

Bei dieser Deklarationsart wird vom Compiler automatisch der für den String benötigte Speicherplatz reserviert.

Syntax: ***char *st="String";***

Beispiel: char *meld1="Bitte Zahl eingeben";

Funktionen zur Stringbearbeitung

strcat

Syntax: ***char *strcat(char *dest, const char *src);***

Beschreibung: Der String *src* wird an den String *dest* angefügt. *dest* muß genügend Speicherplatz für beide Teilstrings bereitstellen.

Rückgabe: Zeiger auf das erste Zeichen vom String *dest*

strncat

Syntax: ***char *strncat(char *dest, const char *src, size_t maxlen);***

Beschreibung: Hängt *maxlen* Zeichen vom String *src* an den String *dest* an. Es werden immer *maxlen* Zeichen kopiert, auch wenn *maxlen* größer als die Stringlänge von *src* ist. Hierbei werden evtl. Nullzeichen mitkopiert. Ist *maxlen* kleiner als *src*, so wird kein abschließendes Nullzeichen mitkopiert.

Rückgabe: Zeiger auf das erste Zeichen vom String *dest*.

strcmp

Syntax: ***int strcmp(const char *s1, const char *s2);***

Berschreibung: Vergleicht die Inhalte der Strings *s1* und *s2*.

Rückgabe: Zurückgeliefert wird ein int-Wert, der das Ergebnis des Vergleichs wie folgt beschreibt:

=0	beide Strings *s1* und *s2* sind gleich
>0	*s1* ist größer als *s2*
<0	*s1* ist kleiner als *s2*

stricmp

Syntax: ***int stricmp(const char *s1, const char *s2);***

Beschreibung: Wie *strcmp*, jedoch wird Groß- und Kleinschrift nicht unterschieden. Achtung: Umlaute werden nicht in die entsprechenden Großbuchstaben konvertiert.

Rückgabe: Wie bei *strcmp.*

strncmp

Syntax: ***int strncmp(const char *s1, const char *s2, size_t n);***

Beschreibung: Es werden n Zeichen der beiden Strings *s1* und *s2* miteinander verglichen.

Rückgabe: Wie bei *strcmp*.

strcpy

Syntax: ***char *strcpy(char *dest, char *src);***

Beschreibung: Kopiert den String *src* in den String *dest*.

Rückgabe: Zeiger auf den String *dest*.

strncpy

Syntax: ***char *strncpy(char *dest, char *src, size_t maxlen);***

Beschreibung: Kopiert *maxlen* Zeichen aus dem String *src* in den String *dest*. strncpy kopiert immer *maxlen* Zeichen. Wenn *src* weniger als *maxlen* Zeichen enthält, wird eine entsprechende Anzahl von Nullzeichen nach *dest* kopiert. Enthält *src* mehr als *maxlen* Zeichen, so enthält der String *dest* kein abschließendes Nullzeichen.

Rückgabe: Zeiger auf den String *dest*.

strlen

Syntax: ***int strlen(const char *s);***

Beschreibung: Ermittelt die Länge eines Strings in Byte.

Rückgabe: Stringlänge in Byte, bzw. Zeichen.

strstr

Syntax: ***char *strstr(const char *s1, const char*** *s2);

Beschreibung: Sucht den String *s1* nach dem ersten Vorkommen im String *s2*.

Rückgabe: Zeiger auf den Beginn von *s1* in *s2*.

strtok

Syntax: ***char *strtok(const char *s1, const char *s2);***

Beschreibung: Betrachtet den über *s1* angegebenen String als Folge von "Token", die voneinander durch die in *s2* definierten Zeichenfolgen getrennt sind. Erneute Aufrufe von *strtok* mit NULL anstelle des Parameters *s1* liefern weitere "Token" zurück. Achtung: *s1* wird durch *strtok* zerstört.

Rückgabe: Zeiger auf String *s1*, der den String bis zum ersten Token enthält.

Funktionen zur Speichermanipulation

memcmp

Syntax: ***int memcmp(const void *s1, const void *s2, size_t n);***

Beschreibung: Vergleicht die ersten n Bytes der Speicherbereiche *s1* und *s2* miteinander.

Rückgabe: Wie bei Strings wird das Ergebnis des Vergleichs wie folgt interpretiert:

=0	beide Speicherbereiche *s1* und *s2* sind gleich
>0	*s1* ist größer als *s2*
<0	*s1* ist kleiner als *s2*

memcpy

Syntax: ***void *memcpy(void *dest, const void *src, size_t n);***

Beschreibung: Kopiert einen Speicherbereich von *src* nach *dest*. Dabei dürfen sich die Speicherbereiche nicht überschneiden.

Rückgabe: Zeiger auf den Speicherbereich *dest*.

memmove

Syntax: ***void *memmove(void *dest, const void *src, size_t n);***

Beschreibung: Identisch zu *memcpy*, allerdings dürfen sich die Speicherbereiche *src* und *dest* überschneiden.

Rückgabe: Zeiger auf den Speicherbereich *dest*.

memset

Syntax: ***void *memset(void *s, int c, size_t n);***

Beschreibung: Setzt die ersten n Bytes des Speicherbereiches *s* auf den Wert *c*.

Rückgabe: Zeiger auf Speicherbereich *s*.

4.17 Funktionen

Funktionen sind elementare Bestandteile von C. Sie übernehmen ganz bestimmte Teilaufgaben innerhalb eines C-Programms, z.B. Eingabe, Ausgabe, Drucken usw. Alle Funktionen zusammen bilden das Programm. Jedes Programm hat mindestens eine Funktion, die Hauptfunktion main, die den Beginn des Programms markiert. Jede Funktion kann entweder vom Hauptprogramm, von anderen Funktionen oder von sich selbst (rekursiv) aufgerufen werden.

4.17.1 Prototypen

Alle Funktionen sollten sog. Prototypen besitzen, wie sie auch in den schon bekannten Header-Dateien verwendet werden. Ein Funktionsprototyp beinhaltet die entsprechenden Funktionsparameter, also Datentypen des Funktionsergebnisses und der zu übergebenden Parameter, nicht aber den Programmcode der Funktion selbst. Z.B.

```
int summe(int a, int b);
```

Der Compiler verwendet diese Informationen, um den Funktionsaufruf auf seine Gültigkeit hin zu überprüfen. Er kann dann auch automatisch Argumente in den benötigten Typ konvertieren. Prototypen sollten immer am Beginn eines Programms oder in einer Header-Datei definiert werden. In C und C++ ist die Definition von Prototypen nicht unbedingt erforderlich, wenn der Programmcode einer Funktion bereits vor dem Aufruf bekannt ist. Da sich die Übersichtlichkeit und die Fehlersuche allerdings erheblich vereinfacht, sollten Prototypen grundsätzlich verwendet werden.

Allgemeine Syntax für Prototypen:

```
typ func(typ param1, typ param2 ..., typ param n);
```

Parameterübergabe bei Funktionen

Funktionen sollten als eigenständige kleine Programme behandelt werden, d.h. dass sie nicht nur in Verbindung mit einem ganz bestimmten Programm lauffähig sind. Da aber fast alle Funktionen auf Daten eines Programms zugreifen, müssen diese Daten als Parameter an die Funktion übergeben werden. Dies ermöglicht einen hohen Einsatzgrad einer Funktion, da diese mit verschiedenen Programmen aber unterschiedlichen Daten verwendet werden kann, auch wenn die Programmumgebung (z.B. Namen der Variablen) nicht identisch ist. Parameter können zur Funktion hin und zurück übergeben werden. Dabei wird grundsätzlich zwischen zwei Arten von Funktionen unterschieden:

"call by value" (Wertübergabe zur Funktion)
Bei dieser Art der Parameterübergabe wird nicht direkt der Variablenwert übergeben, sondern eine Kopie des Wertes. Wird die entsprechende Variable in der Funktion verändert, so wirkt sich dies nicht auf die Originalvariable aus. Es wird hierbei nur der Wert der Kopie verändert.

"call by reference" (Referenzieren der Parameter über Zeiger)
Hier wird eine Referenz zur Originalvariablen über einen Zeiger hergestellt. Alle Änderungen innerhalb einer Funktion wirken sich auch auf die Originalvariable aus.

4.17.2 "call by value", Funktionen ohne Rückgabewert

Dieser Funktionstyp wird eingesetzt, wenn Daten nur zur Funktion übergeben werden, aber die Funktion selbst kein Ergebnis zurückliefert, also z.B. für Bildschirmausgaben.Ein Bezeichner void zeigt an, dass die Funktion keinen Parameter zurückgibt, bzw. der Funktion kein Parameter übergeben wird. Alle Deklarationen innerhalb einer Funktion sind lokal und außerhalb der Funktion nicht bekannt.

Syntax ohne Parameter:

```
void name(void)
{   Deklarationsteil

    Anweisungsteil
}
```

Syntax mit Parameter:

```
void name(typ param1, typ param2 ... typ param n)
{   Deklarationsteil

    Anweisungsteil
}
```

Beispiel:

```
#include <stdio.h>
void hallo()
{
   printf("Hallo, ich bin eine Funktion\n");
}

void ausgabe(int a)
{
   printf("Wert: %d\n",a);
}

main()
{
   hallo();
   ausgabe(10);
}
```

4.17.3 "call by value", Funktionen mit Rückgabewert

Dieser Funktionstyp wird dort eingesetzt, wo ein Funktionsergebnis gefordert ist, also z. B. das Ergebnis einer Berechung. Die Funktion kann dann auch Teil eines Ausdrucks sein. Z.B. y=sin(x)/z;
Eine Funktion mit Rückgabewert kann auch wie eine Funktion ohne Rückgabewert aufgerufen werden. Das Funktionsergebnis wird in diesem Fall ignoriert.

Syntax ohne Parameter:

```
typ name(void)
{   Deklarationsteil

    Anweisungsteil
    return(Rückgabewert);
}
```

Syntax mit Parameter:

```
typ name(typ param1, typ param2, ..., typ param n)
{       Deklarationsteil

        Anweisungsteil
        return(Rückgabewert);
}
```

Return zur Rückgabe eines Funktionsergebnisses

Mit Hilfe der Anweisung return kann ein Funktionsergebnis zurückgegeben werden. Die Funktion wird sofort beendet und zum aufrufenden Programm/Funktion zurückgekehrt. Jede Funktion mit Rückgabewert kann
mehrere return-Anweisungen enthalten, mindestens aber eine.

Syntax für return: ***return(wert); oder return wert;***

Funktionen ohne Rückgabewert können ebenfalls eine oder mehrere Return-Anweisungen enthalten um z.B. eine Funktion vorzeitig zu beenden.

Syntax für return: ***return;***

Beispiel:

```
#include <stdio.h>                  /* Funktionen mit Rückgabewert */

int add(int a, int b)
{
   return(a+b);
}

void ausgabe( int a)
{
   printf("Wert: %d\n",a);
}

main()
{  int   s;

   a=add(2,5);
   ausgabe(s);
}
```

4.17.4 "call by reference" , Rückgabe von Daten über Zeigerparameter

Dieser Funktionstyp wird dort eingesetzt, wo Funktionen Daten einer aufrufenden Funktion verändern und die Rückgabewerte nicht Teil eines Ausdrucks sein müssen. Kombinationen aus Wertübergabe und Zeigerparameter, sowie Rückgabewerten sind möglich.

Syntax ohne Rückgabewert:
```
void  name(typ *param1, typ *param2, ..., typ *param n)
{   Deklarationsteil

    Anweisungsteil
}
```

Syntax mit Rückgabewert:
```
typ  name(typ *param1, typ *param2, ..., typ *param n)
{   Deklarationsteil

    Anweisungsteil
    return(Rückgabewert);
}
```

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

void eingabe(char *a, int *b)
{
   printf("Name: ");
   scanf("%s,a);
   printf("Alter : ");
   scanf("%d",b);
}

main()
{  int            x=0;
   char st[20]="";

   eingabe(st,&x);
   printf("Name: %s, Alter: %d\n",st,x);
}
```

4.17.5 Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich wiederholt selbst aufrufen. In C kann jede Funktion eine rekursive Funktion sein. Der Anwendungsbereich von rekursiven Funktionen liegt vor allem in Bearbeitung von Listen- und Baumstrukturen, die in späteren Kapiteln angewandt werden.

4.17.6 Übergabe von Parametern aus der Kommandozeile

Wie bei UNIX-Kommandos können auch an C-Programme entsprechende Parameter übergeben werden. Dazu müssen beim Aufruf des Hauptprogrammes main die Parameter argc und argv definiert werden. Diese Parameter werden benötigt um Zeichenketten von der Kommandozeile zu übernehmen.

Syntax: ***main(int argc, char *argv[])***
{
..
}

In *argc* wird dabei die Anzahl der angegebenen Parameter übergeben. *argv* ist ein Vektor von Zeigern, wobei jeder Zeiger auf den Beginn eines Strings zeigt. Die Belegung des Vektors ist wie folgt aufgebaut:

argv[0]	Programmname
argv[1]	1. Parameterstring
...	
argv[n]	n. Parameterstring

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

main(int argc, char *argv[])
{  int   i;

   for(i=0; i< argc; i++)
      printf("Parameter  %d: %s\n",i,argv[i]);
}
```

4.17.7 Variable Parameter bei Funktionen

In C ist es möglich Funktionen zu entwickeln, die eine variable Anzahl von Parametern verarbeiten können. Ein Beispiel dafür ist die Funktion *printf*. Dabei muß jede Funktion mit variablen Parametern mindestens 1 fixen Parameter besitzen. Die Adresse der nachfolgenden variablen Parameter wird in einer Zeigervariable gespeichert. Diese Adresse wird nun jeweils auf den nächsten Parameter verschoben.

Syntax für Funktionen mit variablen Parametern:

Syntax: ***typ name(typ fixparam, ...);***

Die drei Punkte bezeichnen eine Liste variabler Parameter.Zur Vereinfachung der variablen Parameter sind in der Headerdatei "stdarg.h" folgende Makros definiert:

va_list

Syntax: ***va_list argp;***

Beschreibung: Deklaration einer Zeigervariablen für variable Parameter.

va_start

Syntax: ***void va_start(va_list argp, last_param);***

Aufruf: *va_start(va_list-Zeiger,letzter Parameter);*

Beschreibung: Ermittelt anhand des fixen Parameters *param* einen Zeiger auf den folgenden variablen Parameter und weist den Zeiger der Variablen *argp* zu. Dieser Zeiger beinhaltet dann die Adresse des ersten variablen Parameters.

va_arg

Syntax: ***typ va_arg(va_list argp,typ);***

Aufruf: *typ=va_arg(va_list-Zeiger,Datentyp des Parameters);*

Beschreibung: Gibt den nächsten Parameter vom Datentyp *typ* als Funktionswert zurück. *argp* ist dabei ein Zeiger auf einen variablen Datenbereich. Der Zeiger wird anschließend um *sizeof(typ)* Bytes erhöht und zeigt dann auf den nächsten variablen Parameter. Folgende Aufrufe von *va_arg* liefern weitere variable Parameter zurück.

va_end

Syntax: ***void va_end(va_list argp);***

Aufruf: *va_end(va_list-Zeiger);*

Beschreibung: Beendet die Auswertung variabler Parameter.

Probleme beim Arbeiten mit variablen Parametern:

Eine spezielle Erkennung des letzten Parameters beim Aufruf von va_arg gibt es nicht. Hier muß ein entsprechender Wert (0, -1, NULL) zur Erkennung verwendet werden. Eine weitere Möglichkeit ist im ersten fixen Parameter die Anzahl der noch folgenden variablen Parameter anzugeben.

Beispiel:
```
#include <stdio.h>
#include <stdarg.h>

void test(int anz, ...)
{   int         i;
    va_list     p;

    va_start(p,anz);
    for(i=0; i<anz; i++)
      printf("%d\n",va_arg(p,int));
    va_end(p);
}
```

4.18 Strukturen (struct)

Strukturen sind abgeleitete Datentypen, die mehrere Komponenten (Variablen) unter einem einzigen Namen zusammenfassen. Dabei können die einzelnen Variablen völlig unterschiedliche Datentypen besitzen. Es lassen sich somit völlig neue Datentypen erzeugen. Anwendungsgebiete sind z.B. die Dateiverarbeitung sowie Listen- und Baumstrukturen.

Syntax:
```
struct [typname]   { typ  komponente 1;
                     typ  komponente 2;
                          ...
                     typ  komponente n;
                   } [variable];
```

typname ist der Name des neuen Datentyps (vergleichbar z.B. mit *int*), der es ermöglicht Variablen zu vereinbaren. *variable* ist eine Variable, die sofort bei der Deklaration des Strukturtyps vereinbart wird. Wird nur eine Variable einer Struktur gebraucht, so kann *typname* entfallen. *variable* kann entfallen, wenn keine sofortige Variablendeklaration erfolgen soll.

Deklaration einer Strukturvariablen

Vor Verwendung einer Struktur muß zunächst eine Variable vom Datentyp der Struktur vereinbart werden. Dabei muß in C immer *struct* vor dem Namen der Struktur angegeben werden.

Syntax: ***struct typname variable;***

Zugriff auf die Strukturkomponenten

Auf die Komponenten einer Strukturvariablen kann über den Mitgliedsoperator (**.**) oder den Verweisoperator (**->**) bei Zeigern auf Strukturen zugegriffen werden.

Syntax für Zuweisung: ***strukurvariable.komponente=wert;***

Syntax zum Auslesen: ***wert=strukturvariable.komponente;***

Beispiel:
```
#include <stdio.h>
struct   adrtyp { char  name[20];
                  char  abteilung[10];
                  float  gehalt;
                };
void main()
{   struct adrtyp    adr;

    strcpy(adr.name,"Maier");
    adr.gehalt=5124.45;
    printf("Name: %s, Gehalt: %8.2f\n",adr.name,adr.gehalt);
}
```

4.19 Typen definieren (*typedef*)

Mit Hilfe von *typedef* können Datentypnamen geändert werden. Vorsicht, es werden keine neuen Datentypen gebildet, sondern nur bereits bestehende Datentypen ein neuer Name zugeordnet. Der alte Datentypname bleibt aber weiterhin erhalten.

Syntax: ***typedef typ_alt typ_neu;***

4.20 Dateiverarbeitung

Bisher wurden die Daten, mit denen die Programme gearbeitet haben nur im Arbeitsspeicher (z.B. Variablen) gespeichert. Nach Beendigung des jeweiligen Programms waren die eingegeben Daten wieder verloren. Um Daten dauerhaft zu speichern, werden Dateien verwendet, die die Daten nicht im Arbeitsspeicher sondern auf einem externen Speicher (z.B. Festplatte oder Diskette) ablegen. Das Lesen und Schreiben von Dateien wird vom Betriebssystem gesteuert. Dabei wird für jede Datei ein Dateiname im entsprechenden Verzeichnis angelegt.

Bevor mit einer Datei gearbeitet werden kann muß diese zunächst geöffnet werden. Anschließend können Daten in die Datei geschrieben oder aus ihr gelesen werden. Nach der Bearbeitung muß die Datei wieder geschlossen werden. In C stehen eine ganze Reihe von Kommandos zur Dateibearbeitung zur Verfügung, die nachfolgend erläutert werden.

Dateizeiger

Dateizeiger (Filepointer) sind Zeiger auf eine Struktur *FILE*, die in *stdio.h* definiert ist. Diese Struktur enthält alle für den Datentransfer notwendigen Informationen, wie z.B. Puffergröße, Dateibeschreibung, Dateistatus usw. Soll eine Datei eröffnet werden, dann muß ein Zeiger auf diese Struktur *FILE* vereinbart werden.

Syntax: ***FILE *dateivariable;***

Diese Deklaration erstellt eine Verbindung zwischen dem Betriebssystem, das die Datei eröffnet und dem C-Programm. Man bezeichnet die Datei auf dem Datenträger auch als physikalische Datei und den Dateizeiger als logische Datei. Über den Dateizeiger kann jetzt auf die Dateien zugegriffen werden.

Standarddateizeiger

Auch Bildschirm und Tastatur sind im Prinzip Dateien und werden auch dementsprechend verwaltet. Hierzu stehen folgende Standarddateizeiger zur Verfügung, die in der Datei *stdio.h* definiert sind.

stdin	(standard input), Standardeingabe, normalerweise die Tastatur
stdout	(standart output) Standard-Ausgabe, normalerweise der Bildschirm
stderr	(standard error) Standard Fehlerausgabe, normalerweise der Bildschirm

Gepufferte Ein- und Ausgabe

Bei jedem Öffnen einer Datei wird ein Dateipuffer für die entsprechende Datei angelegt. Alle Operationen, ob Lesen oder Schreiben von Daten, werden immer über diesen Puffer abgewickelt. Schreibt man Daten in eine Datei, so werden diese zunächst in den dafür bereitgestellten Puffer kopiert. Erst wenn dieser vollständig gefüllt ist, die Datei geschlossen wird oder die Funktion *fflush* aufgerufen wird, werden die Daten auf den Datenträger geschrieben.

4.20.1 Konstanten zur Dateiverarbeitung:

EOF Entspricht dem Wert -1 und zeigt das Ende einer Datei an. Wird bei einigen Dateifunktionen als Rückgabewert verwendet.

4.20.2 Funktionen zur Dateiverarbeitung:

fopen

Syntax: ***FILE *fopen(const char *filename, const char *type);***
Aufruf: *stream=fopen(filename,type);*

Beschreibung: Öffnet die durch *filename* bezeichnete Datei. Der Parameter *type* legt fest, wie die Datei eröffnet wird. *type* kann folgende Werte annehmen:

r (read)	Datei wird nur zum Lesen eröffnet.
w (write)	Datei wird nur zum Schreiben eröffnet.
a (append)	Datei wird zum Schreiben eröffnet, wobei an das Ende einer bereits vorhandenen Datei angehängt wird. Ist die Datei noch nicht vorhanden, so wird sie neu angelegt.
r+	Eröffnung einer Datei für Lese- und Schreiboperationen. Dabei muß die Datei bereits existieren.
w+	Eröffnung einer neuen Datei für Lese- und Schreiboperationen, wobei die Datei immer neu erstellt wird. Der Inhalt einer Datei gleichen Namens wird dabei zerstört.
a+	Eröffnung einer Datei zum Lesen und Anhängen von Daten.
b (binary)	Datei wird im Binärmodus eröffnet. Dieser Wert kann mit allen oben genannten kombiniert werden. Z.B. "r+b" oder "wb"

Rückgabe: Bei fehlerfreier Ausführung wird ein Dateizeiger zurückgeliefert. Trat beim Öffnen ein Fehler auf, so wird der Wert NULL (0) zurückgegeben.

fclose

Syntax: ***int fclose(FILE *stream);***
Aufruf: *fclose(stream);*

Beschreibung: Schließt die durch *stream* angegebene Datei. Vor dem Schließen werden Daten, die sich noch im Puffer befinden mit dem Kommando *fflush* in die Datei geschrieben und der belegte Pufferspeicher wieder freigegeben (siehe auch *fflush*).

Rückgabe: Bei fehlerfreier Ausführung wird der Wert 0 zurückgeliefert, im Fehlerfall EOF (-1).

feof

Syntax: ***int feof(FILE *stream);***
Aufruf: *end=feof(stream);*

Beschreibung: Prüft, ob das Ende einer Datei erreicht ist.

Rückgabe: Liefer einen Wert ungleich 0 (*true*), wenn das Ende der Datei erreicht ist, im Fehlerfall wird der Wert 0 (*false*) zurückgegeben.

ferror

Syntax: ***int ferror(FILE *stream);***
Aufruf: *error=ferror(stream);*

Beschreibung: Prüft ob zu der angegebenen Datei *stream* das Fehlerflag gesetzt ist.

Rückgabe: Liefert einen Wert ungleich 0 (*true*) wenn ein Fehler aufgetreten ist, sonst wird 0 (*false*) zurückgegeben. Das Fehlerflag wird nicht automatisch zurückgesetzt, dies muß mit der Funktion *clearerr* manuell durchgeführt werden.

clearerr

Syntax: ***void clearerr(FILE *stream);***
Aufruf: *clearerr(stream);*

Beschreibung: Löscht das Fehler-Flag der angegebenen Datei *stream*.

fflush

Syntax: ***int fflush(FILE *stream);***
Aufruf: *fflush(stream);*

Beschreibung: Erzwingt das physikalische Schreiben des Dateipuffers auf den Datenträger.

Rückgabe: Liefert 0 (*false*), wenn kein Fehler auftrat und EOF (-1) im Fehlerfall.

fprintf

Syntax: ***int fprintf(FILE *stream, const char *format [,item[,item] ...]);***
Aufruf: *fprintf(stream,format,item,item,...);*

Beschreibung: Analog zu *printf* werden die Daten formatiert in die angegebene Datei *stream* geschrieben.

Rückgabe: Liefert die Anzahl der geschriebenen Zeichen. Im Fehlerfall wird EOF zurückgegeben.

Beispiel:

```
#include <stdio.h>                /* Textdatei erstellen */
void main()
{  int        i;
   FILE       *dat;

   dat=fopen("test.txt","w");
   if(dat!=NULL)
   {
      for(i=0; i<10;  i++}
         fprintf(dat,"%02d:Teststring\n",i );
      fclose(dat);
   }
}
```

fscanf

Syntax: ***int fscanf(FILE *stream, const char *format [,item[,item] ...]);***
Aufruf: *anzahl=fscanf(stream,format,item,item,...);*

Beschreibung: Analog zu *scanf* werden die Daten formatiert aus der angegebenen Datei gelesen.

Rückgabe: Anzahl der fehlerfrei gelesenen Zeichen. Im Fehlerfall wird EOF zurückgegeben.

Beispiel:
```
#include <stdio.h>                /* Textdatei lesen */
void main()
{   FILE   *dat;
    char   st[40];
    int    anzahl,x;

    dat=fopen("test.txt","r");
    if(dat!=NULL)
    {
       while(fscanf(dat,"%d%s",x,st)!=EOF)
          printf("%d  %s\n",x,st);
       fclose(dat);
    }
}
```

fgetc

Syntax: ***int fgetc(FILE *stream);***
Aufruf: *c=getc(stream);*

Beschreibung: Liest ein Zeichen aus der angegebenen Datei *stream*.

Rückgabe: Das gelesene Zeichen *c*, wenn kein Fehler auftrat oder EOF im Fehlerfall.

fputc

Syntax: ***int fputc(int c, FILE *stream);***
Aufruf: *putc(c,stream);*

Beschreibung: Das Zeichen *c* wird in die angegebene Datei *stream* geschrieben.

Rückgabe: Das Zeichen *c*, wenn kein Fehler auftrat oder EOF im Fehlerfall.

fgets

Syntax: ***char *fgets(char *s, int n, FILE *stream);***
Aufruf: *p=fgets(s,sizeof(s),stream);*

Beschreibung: Analog zu *gets* liest die Funktion einen String mit maximal n-1 Zeichen aus der angegebenen Datei *stream*.

Rückgabe: Einen Zeiger auf den gelesenen String wenn kein Fehler auftrat oder NULL im Fehlerfall.

fputs

Syntax: ***int fputs(const char *s, FILE *stream);***
Aufruf: *puts(s,stream);*

Beschreibung: Der String *s* wird in die angegebene Datei *stream* geschrieben. Dabei wird kein automatischer Zeilenvorschub angehängt, wie dies bei *puts* der Fall ist.

Rückgabe: Das zuletzt geschriebene Zeichen, wenn kein Fehler auftrat oder EOF im Fehlerfall.

fread

Syntax: ***size_t fread(const void *ptr, size_t size, size_t nitems, FILE *stream);***
Aufruf: *count=fread(&data,sizeof(data),1,stream);*

Beschreibung: Liest *nitems* Datenelemente, die jeweils die Größe *size* haben, von der angegebenen Datei in den Speicherbereich, auf den der Zeiger *ptr* zeigt.

Rückgabe: Anzahl der gelesenen Elemente.

Beispiel:
```
#include <stdio.h>                  /* binäre Datei lesen */
void main()
{
   int        i;
   FILE       *dat;
   struct     { int     nr;
                char    text[10];
              } satz;

   dat=fopen("test.dat","rb");
   if(dat)
   {
     while(fread(&satz,sizeof(satz),1,dat)>0)
       printf("%d\t%s\n",satz.nr,satz.text);
     fclose(dat);
   }
}
```

fwrite

Syntax: ***size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);***
Aufruf: *fwrite(&data,sizeof(data),1,stream);*

Beschreibung: Schreibt *nitems* Datenelemente, die jeweils die Größe *size* haben, in die angegebene Datei, die von dem Speicherbereich gelesen werden, auf den der Zeiger *ptr* zeigt.

Rückgabe: Anzahl der geschriebenen Datenelemente.

Beispiel:

```
#include <stdio.h>                /* binäre Datei erstellen */
void main()
{
   int        i;
   FILE       *dat;
   struct     { int     nr;
                char    text[10];
              } satz;

   dat=fopen("test.dat","wb");
   if(dat)
   {
     strcpy(satz.text,"Test");
     for(i=0; i<10; i++)
     {
       satz.nr=i;
       fwrite(&satz,sizeof(satz),1,dat);
     }
     fclose(dat);
   }
}
```

fseek

Syntax: ***int fseek(FILE *stream, long offset, int whence);***
Aufruf: *fseek(stream,offset,whence);*

Beschreibung: Setzt die Position des Dateizeigers in der durch *stream* bezeichneten Datei. Hierbei kann die Positionierung relativ zum Anfang der Datei, relativ zur vorherigen Position oder relativ zum Dateiende erfolgen. Der Parameter *offset* gibt die zu setzende Entfernung in Bytes an. Die Art der Positionierung wird über drei in *stdio.h* definierte Parameter für *whence* eingestellt.

SEEK_SET	0	relativ zum Dateianfang
SEEK_CUR	1	relativ zur vorherigen Position
SEEK_END	2	relativ zum Dateiende

Rückgabe: 0 bei fehlerfreier Ausführung oder ungleich 0 bei Fehler.

rewind

Syntax: ***void rewind(FILE *stream);***
Aufruf: *rewind(stream);*

Beschreibung: Setzt den Dateizeiger auf die Position 0, also auf den Dateianfang. Das Kommando ist Äquivalent zu *fseek(SEEK_SET,0);*

ftell

Syntax: ***long ftell(FILE *stream);***
Aufruf: *pos=ftell(stream);*

Beschreibung: Liefert die aktuelle Position des Dateizeigers *stream* als Funktionsergebnis zurück. Die Position bezieht sich immer auf den Anfang der Datei.

Rückgabe: Aktuelle Position des Dateizeigers oder -1 bei Fehler.

Beispiel:

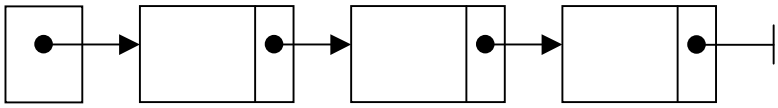
```
#include <stdio.h>                  /* Dazeigröße ermitteln */
void main()
{
   FILE     *dat;
   char     fname[10]="test.dat"
   long     x;

   dat=fopen(fname,"rb");
   if(dat)
   {
     fseek(dat,0,SEEK_END);
     x=ftell(dat);
     printf("Die Datei '%s' ist %ld Bytes groß.\n",fname,x);
     fclose(dat);
   }
}
```

4.21 Dynamische Datenstrukturen

Dynamische Datenstrukturen sind Listen und Bäume. Dabei handelt es sich um grundlegende Datenstrukturen, die eine Vielfalt von Anwendungsmöglichkeiten besitzen. Die Algorithmen zum bearbeiten dieser Strukturen sind in der Regel rekursiv aufgebaut.

4.21.1 Listen

Listen sind dynamische Elemente, die über Zeiger linear miteinander verbunden sind. Dabei existiert ein Startzeiger (Kopf), der auf das erste. Element der Liste verweist. Jedes Element verfügt weiterhin über einen Zeiger, der jeweils auf das nächste Element zeigt usw.



Jedes Listenelement wird dabei durch eine entsprechende Struktur beschrieben, z.B.

```
struct  listtyp  {
                   int       key;
                   listtyp   *next;
                 }
```

Folgende Listenoperationen sind möglich:

- Erstellen einer neuen Liste
- Einfügen eines Listenelementes am Kopf oder am Ende der Liste
- Einfügen eines Listenelementes vor und nach einem anderen Listenelement
- Suchen eines Elementes in der Liste
- Löschen eines Listenelementen
- Verketten von zwei Listen

Für die oben genannten Listenoperationen werden vorwiegend rekursive Funktionen eingesetzt.

Beispiel: Rekursive Ausgabe einer Liste

```
struct listtyp    { int      key;
                    listtyp  *next;
                  };

void print(listtyp *x)
{
  if(x!=NULL)
  {
    printf("%d\n",x->key);
    print(x->next);
  }
}
```

Beispiel: Listenelement anhängen (mit Referenz)

```
listtyp anhaengen(listtyp *&x, int a)
{
  if(x==NULL)
  {
     x=(listtyp*)malloc(sizeof(listtyp));
     x->key=a;
     x->next=NULL;
  }
  else
    anhaengen(x->next,a);
}
```

Beispiel: Listenelement löschen (mit Referenz)

```
void loeschen(listtyp *&x, int key)
{
  if(x!=NULL)
  {
     if(x->key!=key)
       loeschen(x->next,key);
     else
     { listtyp  *y=x;
       x=x->next;
       free(y);
     }
  }
}
```

Beispiel: Listenelement anhängen (Zeiger auf Zeiger)

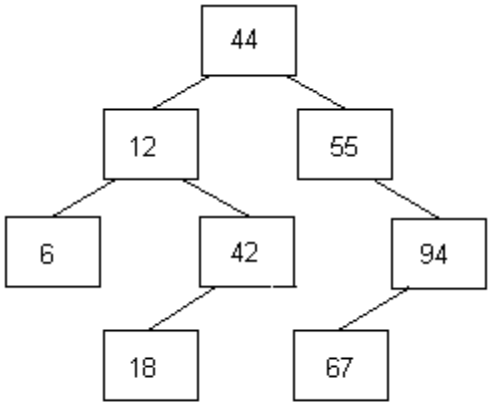
```
listtyp anhaengen(listtyp **x, int a)
{
  if(*x==NULL)
  {
     *x=(listtyp*)malloc(sizeof(listtyp));
     (*x)->key=a;
     (*x)->next=NULL;
  }
  else
    anhaengen(&(*x)->next,a);
}
```

Beispiel: Listenelement löschen (Zeiger auf Zeiger)

```
void loeschen(listtyp **x, int key)
{
  if(*x!=NULL)
  {
     if((*x)->key!=key)
        loeschen(&(*x)->next,key);
     else
     { listtyp  *y=*x;
       *x=(*x)->next;
       free(y);
     }
  }
}
```

4.21.2 Bäume

Binäre Bäume sind ebenfalls dynamische Datenstrukturen, die allerdings nicht nur einen, sondern zwei (oder auch mehr) Nachfolger haben. Bäume werden gewöhnlich von oben nach unten gezeichnet, d.h. die Wurzeln der Bäume zeigen nach oben. Ein Baum verfügt über Knoten und über Blätter.



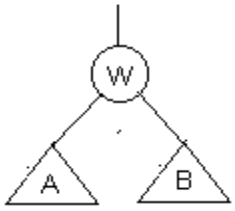
Begriffe zu Bäumen:

root	Der oberste Knoten (Stufe 0)
Nachfolger	Ein Knoten y direkt hinter einem Knoten x
Vorgänger	Ein Knoten x direkt über einem Knoten y
Stufe	Die Stufe der Wurzel ist 0. Ist die Sufe eines Vorgängers i, so ordnet man seinem Nachfolger die Stufe (i+1) zu.
Höhe	Die größte Stufe eines Elementes im Baum
Blatt	Endelement, also ein Knoten ohne Nachfolger
Grad	Die Zahl der direkten Nachfolger eines Knotens.
Binärer Baum	Baum mit Grad 2.
ausgeglichener Baum	Die Anzahl der Knoten im Linken und rechten Teilbaum unterscheiden sich höchstens um eins.

Jeder Baumelement (Knoten) wird dabei durch eine entsprechende Struktur beschrieben, z.B.

```
struct   treetyp {
                  int       key;
                  treeyp   *left, *right;
                }
```

Auch bei Bäumen sind vielfältige Operationen möglich. Dabei werden die Knoten eines Baumes nach bestimmten Regeln durchlaufen. Es gibt im wesentlichen drei Ordnungen. Dabei bedeutet **W** die Wurzel und **A** und **B** den linken und rechten Teilbaum.



1. Preorder: W, A, B (besuche Wurzel vor den Teilbäumen)
2. Inorder: A, W, B
3. Postorder: A, B, W (besuche Wurzel nach den Teilbäumen)

Beispiel: binärer Baum

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

struct treetyp    { int      key;
                    treetyp  *left,*right;
                  };

void printtree(treetyp *x)
{
  if(x!=NULL)
  {
    printtree(x->left);
    printf("%d\n",x->key);
    printtree(x->right);
  }
}

void search(treetyp *&x, int a)
{
  if(x==NULL)
  {
    x=(treetyp*)malloc(sizeof(treetyp));
    x->left=NULL;
    x->right=NULL;
    x->count=1;
    x->key=a;
    return;
  }

  if(a<x->key)
    search(x->left,a);
  else
    if(a>x->key)
      search(x->right,a);
    else
      x->count++;
}

void main()
{ treetyp *root=NULL;
  int     a,i;

  do
  {
    printf("Key (Ende mit -1): ");
    scanf("%d",&a);
    if(a>=0)
      search(root,a);
  }
  while(a>=0);

  printf("----------------\n");
  printtree(root);
}
```