L e i t f a d e n

zur

C - Programmierung

für

Anfänger und Fortgeschrittene

Programmiersprache C
(ANSI Standard 1989)

Vorwort

Die problemorientierte Programmiersprache C stellt einen Kompromiss zwischen einer höheren Programmiersprache und dem Assembler dar und ist vorzüglich im Bereich der Systemprogrammierung einsetzbar. Einerseits wird durch moderne Sprachelemente die Entwicklung gut strukturierter, übersichtlicher und portabler Programme ermöglicht; andererseits bietet C die Möglichkeit einer systemnahen Programmierung, einer leistungsfähigen Schnittstelle zur Hardware. Bei dem Versuch, mit den in C eingebauten E/A-Funktionen Bildschirmoberflächen zu realisieren, wie sie bei professionellen Programmen heute üblich geworden sind, stellt sich schnell heraus, dass dies mit den herkömmlichen E/A-Funktionen nicht zu schaffen ist. Die Entwicklung von SAA-konformen Benutzeroberflächen unter Nutzung der entsprechenden C - Werkzeuge setzt jedoch fundierte Kenntnisse über den Umgang mit der Sprache C voraus.

C wird von seinen Entwicklern als sog. „low level“ (niedrige) Programmiersprache eingestuft. Sie bietet ein reiches Spektrum an Operatoren, die sehr vorteilhaft einsetzbar sind. Dem Programmierer werden viele Entwicklungsfreiräume eingeräumt. Zur Laufzeit werden hinsichtlich der Datenobjekte im Gegensatz zu den meisten anderen Programmiersprachen keinerlei Prüfungen vorgenommen.

In diesem Kurs wird die Syntax sowie die Semantik der Sprache C entsprechend eines vom Verfasser erarbeiteten Schulungskonzeptes kurz und sachlich vermittelt. Der hier dargestellte C Dialekt entspricht dem **ANSI-Standard per 1989**. Die Weiterentwicklungen, wie etwa C++, Visual C++ oder Objective-C 2.0, basieren letztlich auf diesem ANSI-Standard. Für Programmieraufgaben in diesen Dialekten sind nach wie vor solide C-Kenntnisse unverzichtbarer Bestandteil Ihrer täglichen Arbeit.

Die vorliegende Gliederung hat sich in mehreren Vorlesungen und Kursen als methodisch nützlich erwiesen und ist

° für Anfänger als Unterstützung im Selbststudium,
° für Lernende als Lernhilfe im Unterricht oder der Vorlesung,
° für Fortgeschrittene als Nachschlagewerk bei Syntaxproblemen und
° für Lehrende als Unterrichtshilfe zur Umsetzung des zu vermittelnden Stoffes

gedacht.

Die Syntax einer Sprache definiert die Zeichenketten, mit denen ein C-Programm gebildet werden kann. Die niedrigste Ebene der Syntaxregeln ist die Buchstabierungsrichtlinie für die Grundsymbole der Sprache.

Die Semantik einer Sprache beschreibt die Bedeutung der Ausdrücke (Programmterme), die als Grundlage der Sprache selbst fungieren.

Außer der im Punkt 10 aus methodischen Gründen nach zwei Kriterien angegebene Aufstellung der Ein - und Ausgabefunktionen und einer ausführlichen Beschreibung der Ausgabefunktion **printf( )** werden in diesem Kurs keine Angaben zu den Standardfunktionen gemacht; es sei hier auf die umfangreiche Literatur für die einzelnen C Compiler sowie auf die Help-Angaben der Testsysteme verwiesen.

Alle 43 angeführten Beispiele wurden auf einem IBM-PS/2 unter dem Betriebssystem IBM DOS Version 5.0 mit dem **Compiler Microsoft C Compiler Version 5.10 und dem Microsoft (R) Overlay Linker Version 3.65** getestet und liegen als Übungsbeispiele vor, versehen mit allen notwendigen Ein- und Ausgabefunktionen. In den Textbeispielen wurde anfänglich auf diese Funktionen verzichtet; die Ergebnisse wurden als Kommentare wiedergegeben.

Die Compiler - Protokollausdrucke erfolgten mittels der **Source Listing Option des Microsoft C Compilers 5.10**. Neben der vollständigen Auflistung einiger Beispiele sind viele C - Listings nur teilweise wiedergegeben worden. Die lokalen und globalen Symboltabellen wurden ebenfalls nur dann abgedruckt, wenn sie der Darstellung des entsprechenden Beispiels dienlich sind.

Der vorliegende Text wurde mit Office **WORD 2013** unter dem Betriebssystem **WINDOS 7 Professional** auf einem ThinkPad S540 überarbeitet und als PDF-File unter dem Namen **ANSI-C V2.1** zur Verfügung gestellt.

INHALTSVERZEICHNIS

1. GRUNDLEGENDE SPRACHELEMENTE, BEGRIFFSBESTIMMUNG 6

1.1 SYNTAXDIAGRAMME 6
1.2 IDENTIFIKATOREN, BEZEICHNER 6
1.3 TRENNZEICHEN 6
1.4 SCHLÜSSELWÖRTER 7
1.5 KONSTANTEN 8
1.5.1 Integerkonstanten 8
1.5.2 Realkonstanten 8
1.5.3 Zeichenkonstanten 9
1.5.4 Zeichenkettenkonstanten 10
1.5.5 Symbolische Konstanten 10
1.5.6 Programmbeispiel für Symbolische Konstanten 10
1.5 PROGRAMMAUFBAU 11

2 DATENTYPEN UND SPEICHERKLASSEN VON EINFACHEN VARIABLEN 13

2.1 TYPBEZEICHNER 13
2.1.1 Ganzzahlvariable 14
2.1.2 Gleitkommavariable 14
2.1.3 EindimensionaleFelder 14
2.1.4 Zeichenketten 15
2.2 DEFINITION NEUER TYPBEZEICHNER 15
2.3 SPEICHERKLASSEN 15
2.3.1 Speicherklasse auto 16
2.3.2 Speicherklasse extern 16
2.3.3 Speicherklasse static 16
2.3.4 Speicherklasse register 16
2.4 UMWANDLUNG VON DATENTYPEN 17
2.5 VARIABLENZUSÄTZE 18
2.5.1 Datentypen mit signed und unsigned 18
2.5.2 Aufzählungstyp enum 18
2.5.3 Datentypen mit const 19
2.5.4 Datentypen mit volatile 19
2.6 INITIALISIERUNG 19

3 OPERATOREN, AUSDRÜCKE, ANWEISUNGEN 21

3.1 PRIORITÄT DER ABARBEITUNGSFOLGE 21
3.2 DIE EINSTELLIGEN, DIE UNÄREN OPERATOREN 22
3.2.1 Adressoperatoren 22
3.2.2 Arithmetische Operatoren 22
3.2.3 Logische Operatoren 23
3.2.4 Bitoperator 23
3.2 CAST - OPERATOR 23
3.2.6 sizeof - Operator 24
3.3 DIE ZWEISTELLIGEN, DIE BINÄREN OPERATOREN 25
3.3.1 Arithmetische Operatoren 25
3.3.2 Vergleichsoperatoren 25
3.3.3 Bitoperatoren 25
3.3.4 Zuweisungsoperatoren 26
3.3.5 Logische Operatoren 27
3.3.6 Bedingungsoperator 27
3.3.7 Kommaoperator 27

4. STEUERSTRUKTUREN 28

4.1 DIE BEDINGTE ANWEISUNG 28
4.2 DIE FALLUNTERSCHEIDUNG SWITCH 29
4.3 PROGRAMMSCHLEIFEN 30
4.3.1 Die Abweisschleife - die while Anweisung 30
4.3.2 Die Nichtabweisschleife - die do Anweisung 31

4.3.3 Die verallgemeinerte abzählbare Schleife, die for Anweisung ... *31*
4.4. Die Unterbrechungsanweisung - die break Anweisung ... 32
4.5. Die Fortsetzungsanweisung - die continue Anweisung ... 33
4.6. Die Sprunganweisung - die goto Anweisung ... 33

5. ZEIGER ... 34

5.1 Zeigervariable ... 34
5.2 Zeigeroperationen - Adressrechnungen ... 34
5.3 Zeigerinitialisierungen ... 35
5.4 Zeigerzusätze ... 36
5.5 Dynamische Speicherverwaltung ... 36

6. MEHRDIMENSIONALE FELDER ... 38

6.1 Zeiger auf Felder ... 38
6.2 Initialisierung von Feldern ... 39

7. STRUKTUREN UND VEREINIGUNGEN ... 41

7.1 Definitionen von Strukturen ... 41
7.2 Bezugnahme auf Strukturvariable und deren Komponenten ... 43
7.3 Initialisieren von Strukturen ... 44
7.4 Vereinigungen ... 45

8. FUNKTIONEN ... 46

8.1 Überblick und Klassifizierung der Standardfunktionen ... 46
8.2 Definition von Funktionen ... 46
8.3 Aufruf von Funktionen ... 47
8.4 Übergabe eines Funktionswertes ... 48
8.5 Rekursiver Funktionsaufruf ... 48
8.6 Zeiger auf Funktionen ... 49

9. DER C - PRÄPROZESSOR ... 51

9.1 File - Einfügungen ... 51
9.2 Makrosubstitution, Definitionen symbolischer Konstanten ... 51
9.3 Streichen von Makro-Definitionen ... 52
9.4 Bedingte Compilierung ... 52

10. DATEIVERWALTUNG ... 54

10.1. Eigenschaften der Dateibehandlung ... 54
10.2 Ein - und Ausgaberoutinen der Standardbibliotek ... 57
10.2.1 Zeichenweise Ein - und Ausgabe ... 57
10.2.2 Zeilenweise Ein - und Ausgabe ... 58
10.2.3 Formatgesteuerte Ein - und Ausgabe ... 59
10.3. Dateiverwaltung durch Low - und High - Level Funktionen ... 63
10.3.1 Die File - Verarbeitung, die Low - Level Funktionen ... 63
10.3.2 Die Stream - Verarbeitung, die High - Level Funktionen ... 65

11. VERZEICHNIS DER TEST- UND BEISPIELPROGRAMME ... 69

1. Grundlegende Sprachelemente, Begriffsbestimmung

Es gilt der vollständige ASCII-Zeichensatz. Er definiert 128 Zeichen, bestehend aus 33 nichtdruckbaren sowie 95 druckbaren. Zur Beschreibung der Sprache C unterscheiden wir folgende Klassen von lexikalen Einheiten (token):

° Identifikatoren
° Trennzeichen
° Schlüsselwörter
° Konstanten
° Zeichenketten
° Operatoren

1.1. Syntaxdiagramme

Nur ein syntaktisch korrektes Programm kann vom entsprechenden Compiler übersetzt werden, um die zu bewältigten Aufgaben per Programm zu lösen.
Zur Beschreibung der Syntax einer Sprache begibt man sich außerhalb dieser zu beschreibenden Sprache. Eine sinnvolle Variante zur Beschreibung der Syntax von C wäre die Meta-Sprache von Backus-Naur, die sog. **Backus-Naur-Form**. Erstmalig wurde die Syntax der Sprache ALGOL 60 vollständig mit dieser Methode beschrieben. In der Beschreibung der Syntax von C kann nur sehr vereinzelt auf diese Notation zurückgegriffen werden, da Elemente der Sprache selbst mit den sog. Konnektoren dieser Metasprache übereinstimmen und somit eine konsequente Anwendung der Backus-Naur-Form verbietet. Doch solange es der Übersichtlichkeit dient, wird von dieser Methode Gebrauch gemacht.
In dieser Niederschrift werden die Schlüsselwörter der Sprache C wegen der Übersichtlichkeit fett dargestellt, während die Syntaxbeschreibungen in den allgemeinen Formen fett kursiv dargestellt werden.

1.2. Identifikatoren, Bezeichner

Ein Identifikator bezeichnet einen bestimmten Speicherbereich und wird durch zwei Attribute charakterisiert:

° Speicherklasse und
° Datentyp

Die Speicherklasse bestimmt die Lebensdauer des Speicherplatzes. Der Datentyp bestimmt die Bedeutung des Wertes, der sich auf dem Speicherplatz befindet.

Folgende Regeln gelten zur Verwendung als Bezeichner von Variablen und Konstanten:

° die ersten 32 Zeichen sind signifikant
° das erste Zeichen muss ein Buchstabe sein
° der Unterstrich _ zählt als Buchstabe
° Groß - und Kleinbuchstaben sind voneinander verschieden
° die deutschen Umlaute sowie der Buchstabe **ß** sind nicht zulässig
° reservierte Wörter dürfen nicht als Bezeichner benutzt werden (s.1.3)

1.3. Trennzeichen

Als Trennzeichen, den sog. white space, gelten

° das Leerzeichen (Space, Hex.20)
° der Tabulator (HT, Hex.09)
° neue Zeile (NL,LF, Hex.0A)
° der Kommentar

Trennzeichen, einschließlich Kommentare, werden bei der Syntaxprüfung durch den Compiler ignoriert und beeinflussen weder die Länge des erzeugten Programmcodes noch die Ablaufgeschwindigkeit des erzeugten Programms.

Alle Zeichen, die in der Zeichenkombination **/*** und ***/** eingeschlossen sind, werden als Kommentare aufgefasst.

Hinweis:

Die Schachtelung von Kommentaren ist nach dem ANSI-Standard nicht zulässig.

Programmbeispiel 1:

```
                                                        PAGE   1
                                                        12-24-90
                                                        14:49:16

Line# Source Line                          Microsoft C Compiler Version 5.10

    1  /*  Das ist ein C-Programm mit einem Kommentar                    */
    2  main()
    3  {
    4  /* Das Listing erfolgt mittels der Source Listing                */
    5  /* Option des Micro-Soft C-Compilers 5.10 über                   */
    6  /* folgendes Kommando:  cl /Fs /c progname.c                     */
    7  /* Ein Identifikator wird mit einer Speicherklasse               */
    8  /* und einem Datentyp definiert                                  */
    9  static int ident;
   10  }

main  Local Symbols
Name                Class           Type                    Size            Offset                  Register
ident . . . . . .. . static          int             2       0000

Global Symbols
Name                ClassType           Size    Offset
main. . . . . global  near function             ***     0000

Code size = 0006 (6)
Data size  = 0000 (0)
Bss size   = 0002 (2)
No errors detected
```

1.4. Schlüsselwörter

Schlüsselwörter sind von der Sprache C reservierte Namen und dürfen nicht als Bezeichner verwendet werden. Diese Schlüsselwörter fett dargestellt.
Folgende Wörter sind nach den Erfindern der Sprache C, Brian W. Kernighan und Dennis Ritchie (K&R), Schlüsselwörter:

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

Hinweis:

Es ist zu beachten, dass es Erweiterungen bzw. Einschränkungen von reservierten Wörtern durch die unterschiedlichen Compilerarchitekturen geben kann.

Programmbeispiel 2:

```
                                                                PAGE 1
                                                                12-23-90
                                                                18:11:06

Line#  Source Line                          Microsoft C Compiler Version 5.10

    1  main()
    2  {
    3  /* Verwendung eines Schlüsselwortes als Variable                  */
    4  char char;
    ***** b2.c(4) : error C2139: type following 'char' is illegal
    5  }
    1 errors detected
```

1.5. Konstanten

Informationen, die bereits bei der Programmierung vorgegeben werden und sich während des Programmablaufs nicht ändern, nennt man Konstanten. Die Sprache erlaubt die Verwendung folgender Konstanten:

° Integerkonstanten
° Realkonstanten
° Zeichenkonstanten
° Zeichenkettenkonstanten
° symbolische Konstanten

1.5.1. Integerkonstanten

Integerkonstanten, auch numerische Konstanten genannt, können als

° Dezimalkonstanten zur Basis 10
° Oktalkonstanten zur Basis 8
° Hexadezimalkonstanten zur Basis 16
vorkommen.

Allgemein besteht eine Integerkonstante aus einer Folge von Ziffern aus der Menge der Ziffern von **0** bis **9** sowie von **A** bis **F** bzw. **a** bis **f**, wobei die zulässigen Ziffern von der verlangten Basis abhängen.

Folgende Fälle sind möglich:

° Dezimalkonstanten mit den Dezimalziffern **0** bis **9**
  - das nachgestellte Zeichen **L** oder **l** erzwingt eine Speicherung als **long**-Wert
  - das nachgestellte Zeichen **U** oder **u** behandelt die Konstante als **unsigned**-Wert (vorzeichenlos)
° Oktalkonstanten mit den Oktalziffern **0** bis **7** mit vorangestellter **0**
° Hexadezimalkonstanten mit den Ziffern **0** bis **9** und die Buchstaben **a** bis **f** bzw. **A** bis **F** mit vorangestellter Zeichenfolge **0x** bzw. **0X**

1.5.2. Realkonstanten

Realkonstanten (Gleitkommakonstanten) können aus
° einem ganzzahligen Anteil,
° einem Dezimalpunkt,
° einem gebrochenen Anteil,
° dem Zeichen **E** oder **e** als Exponent oder
° einem eventuell vorzeichenbehafteten Exponenten

bestehen. Ganzzahliger und gebrochener Anteil sowie der Exponent stellen Folgen von Dezimalziffern dar.

Hinweis:

° Alle nichtganzen Konstanten werden als Realkonstanten gedeutet, auch wenn kein Exponent spezifiziert ist. Die Speicherung erfolgt in doppelter Genauigkeit.
° Durch Anfügen des Zeichens **f** oder **F** wird eine Speicherung mit einfacher Genauigkeit erzwungen.

1.5.3. Zeichenkonstanten

Einzelne Zeichen werden durch Einschluss in Apostrophe (einfache Hochkomma) dargestellt. Der numerische Wert einer Zeichenkonstanten hängt vom verwendeten ASCII-Zeichensatz ab. Zeichen werden intern mit ihrem ASCII-Code abgespeichert. Einige Sonderzeichen werden als Zeichenkombination aus mehreren Zeichen dargestellt. Sie können nicht als ein Zeichen über die Tastatur eingegeben werden, sondern werden durch die sogenannte Fluchtsymbolschreibweise dargestellt.

Hinter dem Fluchtsymbol (**\ ,**backslash) sind folgende Kombinationen möglich:

° '***nnn***' , nnn ist eine dreistellige Zahl im Oktalsystem
° '***xhh***' , hh ist eine zweistellige Hexadezimalzahl

Durch diese beiden Möglichkeiten läßt sich jedes beliebige ASCII-Zeichen darstellen.

Als festgelegte Fluchtsymbolcodes gelten:

ASCII	Bedeutung	Hex.Code	Flucht-symbol
\a BEL	Signalton, alert	hex.07	--> '\a'
\0 NUL	Nullzeichen	hex.00	--> '\0'
\n NL,LF	Newline-Zeichen	hex.0A	--> '\n'
\t HT	Tabulator-Zeichen	hex.09	--> '\t'
\b BS	Backspace-Zeichen	hex.08	--> '\b'
\f FF	Seitenvorschub-Zeichen	hex.0C	--> '\f'
\r CR	Wagenrücklauf-Zeichen	hex.0D	--> '\r'
\v VT	Vertikal-Tabulator	hex.0B	--> '\v'
'	Single quote	hex.27	--> '\''
"	doppeltes Apostroph-Z.	hex.22	--> '\"'
****	Backslash-Zeichen	hex.5C	--> '\\'
?	Fragezeichen	hex.3F	--> '\?'

In allen Fällen wird nur ein Zeichen dargestellt.

1.5.4. Zeichenkettenkonstanten

Eine Zeichenkettenkonstante besteht aus Zeichenkonstanten, die von doppelten Anführungszeichen (doppelte Hochkomma) begrenzt sind. Der Compiler fügt automatisch das Backslash-Zeichen '\' als Endekennung an.

Hinweis:

- Die leere Zeichenkettenkonstante "" besteht nur aus der Endekennung '\0'.
- Nichtdarstellbare Zeichen werden in Zeichenkonstanten notiert
- Das Zeichen selbst ist innerhalb einer Zeichenkettenkonstante. als Escape-Folge \" darzustellen
- Während die Zeichenkonstante 'A' nur aus den ASCII- Zeichen A besteht, stellt die Zeichenkettenkonstante "A" eine Folge von zwei Zeichen dar

1.5.5. Symbolische Konstanten

Sie dienen der Lesbarkeit von C - Quelltexten. Symbolische Konstanten werden vom C - Präprozessor ausgewertet.

Folgende Regeln gelten:

- Die Vereinbarung erfolgt mittels der Präprozessor - Anweisung **#define**.
- Die Regeln zur Bildung von Namen für symbolische Konstanten sind analog zu denen für Identifikatoren.
- Der Präprozessor ersetzt die symbolische Konstante bei jedem Auftreten im nachfolgenden Quelltext durch die angegebene Zeichenfolge.

1.5.6. Programmbeispiel für Symbolische Konstanten

Programmbeispiel 3:

```
                                                        PAGE   1
                                                        01-02-98
                                                        23:20:34

Line#  Source Line                              Microsoft C Compiler Version 5.10

    1   /* Beispiel 3                                                          */
    2   /* Beispiele fuer moegliche Konstantentypen                            */
    3   /* Speicherklasse und Datentyp werden im Abschnitt 2 besprochen        */
    4   #define SYM_KONST 1000              /* Symbolische Konstante          */
    5   int dec_const = 32767;              /* Dezimale Integerkonstante      */
    6   int dec_long_const = 32770L;         /* Dezimale long-Integerkonst.   */
    7   int hex_const = 0xff;               /* Hexadezimale Integerkonst.     */
    8   int hex_const1 = 0x1f2a;
    9   int okt_const = 0100;               /* Oktale Integerkonst. dez.64    */
   10   float real_const = 123.45E10;        /* Gleitkommakonstante           */
   11   char nul_const = '\0';              /* Zeichenkonstante Nullzeichen   */
   12   char zeich_const = 'x';             /* Zeichenkonstante               */
   13   char sond_const_okt = '\100';        /* Bitkombination Zeichen '@'    */
   14   char sond_const_hex = '\x40';        /* Bitkombination Zeichen '@'    */
   15   char ket_const[] = "Kette";         /* Zeichenkettenkonstante         */
   16   main()
   17   {}

Global Symbols

Name                 Class    Type              Size   Offset

dec_const . . . . . .  global  int                 2    0000
dec_long_const. . . .  global  int                 2    0002
hex_const . . . . . .  global  int                 2    0004
hex_const1. . . . .. .  global  int                2    0006
ket_const . . . . . .  global  struct/array        6    0012
```

```
main. . . . . ..   global  near function      ***  0000
nul_const . . . . . .  global  char           1    000e
okt_const . . . . . .  global  int            2    0008
real_const. . . . . .  global  float          4    000a
sond_const_hex. . .  global  char             1    0011
sond_const_okt. . . .  global  char           1    0010
zeich_const . . . . . .  global  char         1    000f

Code size = 0006 (6)
Data size = 0018 (24)
Bss size  = 0000 (0)
No errors detected
```

1.6. Programmaufbau

Ein C-Programm ist für den Compiler eine beliebig lange Programmzeile, die beliebig in Zeilen kürzerer Längen untergliedert werden kann.

Wichtige Merkmale der Sprache C sind:

° C ist blockorientiert; allerdings werden im Gegensatz zu anderen Sprachen begin und end durch die Zeichen **{** und **}** ersetzt
° in C ist jedes Programm eine Funktion und kann als ein Baustein von anderen Funktionen aufgerufen werden
° eine Funktion, die von der Benutzeroberfläche gestartet wird, muss immer den Namen **main** tragen

Allgemeine Form des prinzipiellen Aufbaus einer Funktion

typbezeichner funktionsname (liste der formalen parameter)
deklaration der formalen parameter
{
deklarationen
anweisungen
}

Erläuterungen:

° ***typbezeichner*** legt den Datentyp des Funktionswertes fest
° ***liste der formalen parameter*** kann leer sein, es muss aber in jedem Fall das Klammernpaar **( )** notiert werden
° ***deklarationen*** legen den Datentyp der in der Funktion benutzten Variablen fest (siehe Punkt 2.)
° ***anweisungen*** legen die Steuerung des Programmablaufs mit sprachlichen Mitteln fest, die mit anderen Programmiersprachen korrespondieren

Programmbeispiel 4:

```
                                                                PAGE 1
                                                                12-23-90
                                                                18:11:46

Line#  Source Line                              Microsoft C Compiler Version 5.10

1  main()
2  {
3  /*    Das ist ein C Programm                 */
4  }
Global Symbols

Name            Class       Type          Size      Offset
main. . . . . . . . . . . global  near function    ***      0000

Code size = 0006 (6)
Data size  = 0000 (0)
Bss size     = 0000 (0)
No errors detected
```

2. Datentypen und Speicherklassen von einfachen Variablen

In C herrscht im Gegensatz zu anderen Sprachen Deklarationszwang. Alle Identifikatoren müssen mit Typbezeichner und Speicherklassenbezeichner zu Beginn eines Blockes vereinbart werden. Um einen Überblick zu geben, welche Variablentypen sich für welche Wertebereiche eignen, zeigt für vorzeichenbehaftete (**signed**) Variablen folgende Tabelle:

Typ	Bite	Wertebereiche			Verwendung
char	8	-128	bis	+127	ASCII-Zeichen
enum	16	-32767	bis	+32767	Ganzzahl
int	16	-32767	bis	+32767	Ganzzahl
short int	16	-32767	bis	+32767	Ganzzahl
long int	32	-2147483648	bis	+2147483647	Ganzzahl
float	32	3.4E-38	bis	3.4E+38	Gleitpunktzahl
		7 Stellen Genauigkeit			
double	64	1.7E-308	bis	1.7+308	Gleitpunktzahl
		15 Stellen Genauigkeit			
long double	64	maschinenabh. IEEE-Standard			Gleitpunktzahl
		19 Stellen Genauigkeit			

Hinweis:

- **short** und **long** sind sog. Modifizierer. Sie verändern Größe und Genauigkeit
- **float** und **double** Variable setzen einen Coprozessor voraus. Ist kein Coprozessor vorhanden, wird die vorhandenen Fließkommabibliothek 'EMU.LIB' eingebunden. Bei der Emulation ist eine stark reduzierte Geschwindigkeit zu verzeichnen
- **long double** setzt einen geeigneten Prozessor voraus

2.1. Typbezeichner

Die richtige Wahl des Variablentyps ist von entscheidender Bedeutung für die Geschwindigkeit und den Speicherbedarf des Programms.

In C gibt es zur Deklaration von einfachen Variablen eigentlich nur vier Grundtypen:

- **char** für 8 Bit-Zeichen entprechend ASCII
- **int** für 16 oder 32 Bit-Worte mit oder ohne (**signed** oder **unsigned**) Vorzeichen
- f**loat** für reelle Zahlen einfacher Genauigkeit (32 Bit-Worte)
- **double** für reelle Zahlen doppelter Genauigkeit (64 Bit-Worte)

Hinweis:

- Zwischen den C üblichen Abkürzungen und deren Langformen bestehen folgende Beziehungen:

Langform	Abkürzung
short int	**short**
long int	**long**
long float	**double**
unsigned int	**int**

2.1.1. Ganzzahlvariable

Sie werden durch die Typbezeichner **char** und **int** bzw. durch die Modifizierer **short** und **long** dargestellt.

Allgemeine Form der Typvereinbarung einer Ganzzahlvariablen:
speicherklasse typbezeichner identifikator;

2.1.2. Gleitkommavariable

Sie werden durch die Typbezeichner **float** und **double** dargestellt. Intern werden Gleitkommazahlen im IEEE-Format gespeichert, so dass sie mit einem eventuell vorhandenen Arithmetik-Prozessor bearbeitet werden können.

Allgemeine Form der Typvereinbarung einer Gleitkommavariablen:
speicherklasse typbezeichner identifikator;

2.1.3. Eindimensionale Felder

Felder sind eine Ansammlung von Variablen gleichen Typs, die logisch, nicht wertmäßig ähnlichen Inhalt und denselben Namen besitzen. Die einzelnen Elemente, die Komponenten, werden über einen in eckigen Klammern eingeschlossenen Index angesprochen.

Allgemeine Form der Typvereinbarung eines eindimensionalen Feldes:
speicherklasse typbezeichner identifikator [anzahl komponenten];

Hinweis:

° Der erste Index wird von intern mit 0 beginnend gezählt

Programmbeispiel 5:

```
                                                        PAGE   1
                                                        05-26-91
                                                        19:08:03

Line#  Source Line                          Microsoft C Compiler Version 5.10

    1  /*     Beispiel 5                                                  */
    2  /* Es werden unterschiedliche Definitionen eineimensionaler Felder */
    3  /* gezeigt                                                         */
    4  #include <stdio.h>
    5   int vektor[1000];
    6   double array[256];
    7   char buffer[BUFSIZ]; /* BUFSIZ ist eine symbolische Konstante und  */
    8                  /* ist in dem Headerfile stdio.h definiert         */
    9  main()
    10  {}

Global Symbols
Name                 Class      Type          Size     Offset
array . . . . . .    common  struct/array  2048     ***
buffer. . . . . .    common  struct/array  512      ***
main. . . . . .      global  near function ***      0000
vektor. . . . . .    common  struct/array  2000     ***
No errors detected
```

2.1.4. Zeichenketten

Zeichenketten sind nichts anderes als ein eindimensionales Felder von **char**-Variablen. Die Feldelemente enthalten dabei den ASCII - Code der zugehörigen Zeichen. Als Endekennung muss als letztes Zeichen **'\0'** stehen, das vom Compiler automatisch hinzugefügt wird. Die Gesamtlänge muss also immer um diese Endekennung größer definiert werden.

Allgemeine Form der Typvereinbarung einer Zeichenkette:
speicherklasse* char *identifikator [anzahl zeichen + 1];

Programmbeispiel 6:

```
...
3  /* Definition einer Zeichenkette zur Aufnahme einer                */
4  /* Zeichenkette der Länge 6                                        */
5  char kette[7]; /* Zeichenkettenvariable                           */
...
No errors detected
```

2.2. Definition neuer Typbezeichner

Allgemeine Form einer Vereinbarung:
typedef typvereinbarung identifikator;

definiert, daß ***identifikator*** als Synonym für einen Typbezeichner benutzt werden soll. Durch **typedef** werden keine neuen Datentypen erzeugt, sondern lediglich Synonyme für bereits existierende Typen.

Einsatzgebiete:

° Unterstützung der Portabilität
° Erhöhung der Lesbarkeit von C-Programmen

Programmbeispiel 7:

```
...
3  /*  Beispiel zur Definition neuer Typbezeichner                    */
4   typedef int LAENGE;
5   LAENGE L1,L2,L3;
...
main  Local Symbols

Name            Class   Type          SizeOffset   Register

L1. . . . . . . . . . . auto                -0006
L2. . . . . . . . . . . auto                -0004
L3. . . . . . . . . . . auto                -0002
No errors detected
```

2.3. Speicherklassen

Jedes Objekt ist an ein bestimmtes Speicherklassenattribut gebunden. Die Speicherklasse
° beeinflusst den Gültigkeitsbereich (scope) dieser Objekte und
° bestimmt die Lebensdauer (lifetime) von Variablen und Parametern.

Folgende Attribute sind möglich:
- ° **auto**
- ° **static**
- ° **extern** und
- ° **register**

Bei fehlender Speicherklassenangabe werden bestimmte Standardannahmen getroffen.

2.3.1. Speicherklasse auto

Der Gültigkeitsbereich von Variablen mit der Speicherklasse **auto** entspricht dem Block, in dem die Vereinbarung erfolgt. Von außerhalb kann auf diese Variable nicht zugegriffen werden. Sie sind lokal bezüglich eines Blockes. Wird eine Variable innerhalb eines Blockes ohne Angabe einer Speicherklasse vereinbart, wird **auto** als Standardangabe angenommen.

2.3.2. Speicherklasse extern

Externe Variable existieren während des gesamten Programmablaufs. Sie sind global bezüglich eines Blockes. Wird eine Variable außerhalb eines Blockes ohne Angabe einer Speicherklasse vereinbart, wird **extern** als Standardangabe angenommen.

2.3.3. Speicherklasse static

Lokale Variable sind generell dynamisch; wird der Block verlassen, verlieren sie umgehend ihren momentan aktuellen Wert. Um dieses bei bestehender Notwendigkeit zu verhindern, benutzen Sie das Speicherklassenattribut **static**. Trotzdem bleiben diese Variablen lokal; sie sind erst bei Wiedereintritt in den entsprechenden Block mit ihren alten Wert verfügbar.

Hinweis:

° Im Gegensatz zu dynamisch lokalen Variablen, die auf dem Stack abgelegt werden, werden statische Variablen im Datensegment gespeichert. Speicherung im Datensegment bedeutet, dass diese Variablen bei Rekursion nicht erneut kopiert werden.

2.3.4. Speicherklasse register

Wenn eine Variable besonders häufig benutzt wird, kann ihr das Speicherklassenattribut **register** zugeordnet werden. Sie wird zwecks Geschwindigkeitssteigerung direkt in ein Prozessorregister abgelegt.

Hinweis:

° Registervariablen dürfen nur vom Typ **int** oder **short** sein
° Registervariablen müssen lokal sein
° maximal sind nur zwei Registervariable erlaubt
° Registervariable sind nicht adressierbar

Programmbeispiel 8:

```
                                                                PAGE   1
                                                                12-23-90
                                                                18:23:02

Line#  Source Line                              Microsoft C Compiler Version 5.10

1  /* Programmbeispiel 8 zeigt den Zusammenhang der Speicherklassen          */
2  /* und deren Gueltigkeitsbereichen    */
3   int var_ext;                           /* externe globale Variable       */
4   main()
5   {
6     int var_lokal;                       /* auto lokale Variable           */
7     static int var_static;               /* static lokale Variable         */
8     register var_reg;                    /* register lokale Variable       */
9     funkt1();                            /* Funktionsaufruf funkt1         */
10     funkt2();                           /* Funktionsaufruf funkt2         */
11   }main  Local Symbols
Name                 Class   Type          Size   Offset  Register
var_lokal . . . . . . . . auto                -0004
var_reg . . . . . . . . . auto                -0002
var_static. . . . . . . . static  int         2    0000

   12   funkt1()                        /* Funktionskoerper funkt1  */
   13   {
   14     extern int var_ext;           /* externe globale Variable   */
   15     int var_lokal;                /* auto lokale Variable       */
   16   }

funkt1  Local Symbols
Name                 Class   Type          Size   Offset  Register
var_lokal . . . . . . . . auto                -0002

   17   funkt2()                        /* Funktionskoerper funkt2   */
   18   {
   19    extern int var_ext;            /* externe globale Variable    */
   20    int var_lokal;                 /* auto lokale Variable        */
   21   }

funkt2  Local Symbols
Name                 Class   Type          Size   Offset  Register
var_lokal . . . . . . . . auto                -0002

Global Symbols
Name                 Class   Type             Size   Offset
funkt1. . . . . . . . . . global  near function    ***    0014
funkt2. . . . . . . . . . global  near function    ***    0022
main. . . . . . . . . . . global  near function     ***    0000
var_ext . . . . . . . . . common  int              2      ***
No errors detected
```

2.4. Umwandlung von Datentypen

Da die Typumwandlung bei Zuweisungen, die sog. arithmetischen Konvertierungen, nach anderen Regeln abläuft als bei der Ausdrucksbildung, müssen alle notwendigen Umwandlungsfälle angegeben werden.

Grundsätzlich gelten bei Ausdrucksbildung mit Operanden unterschiedlichen Typs folgende Regeln:

- Typ **char** und Typ **short** wird in den Typ **int** automatisch umgewandelt
- Typ **float** wird automatisch in den Typ **double** umgewandelt

Alle Umwandlungen in Ergibtanweisungen sollten

- mittels **cast -** Operator oder
- über die eingefügten Funktionen **ato...** und **...toa** realisiert werden.

Ein - und Ausgabedatenumwandlungen werden über die Funktionen **scanf(...)** sowie **printf(...)** bzw. Varianten beider Funktionen in möglichst höchster Genauigkeit durchgeführt.

Hinweis:

° Man sollte sich bei Konvertierungen von dem Grundsatz leiten lassen, stets in den kleinsten Datentyp umzuwandeln der genügend groß ist, um den betreffenden Wert aufnehmen zu können

2.5. Variablenzusätze

In C besteht die Möglichkeit, Variablentypen durch Zusätze zu modifizieren.

2.5.1. Datentypen mit signed und unsigned

Mit **signed** und **unsigned** kann festgelegt werden, ob Ganzzahlvariable Werte mit oder ohne Vorzeichen aufnehmen soll.

Typ	Wertebereich		
unsigned char	0	bis	255
char	-128	bis	127
unsigned short		0	bis 65 535
short	-32 768	bis	32 767
unsigned int	0	bis	65 535
int	-32768	bis	32767
unsigned long		0	bis 4 294 967 295
long	-2 147 483 648	bis	2 147 483 647

Programmbeispiel 9:

```
...
3  /*  Beispiele für vorzeichenlose int- Variablen */
4    unsigned ganz_var1;         /*  int wird impliziert                    */
5    unsigned short ganz_var2;
6   }
No errors detected
```

2.5.2. Aufzählungstyp enum

Mittels Aufzählungstyp **enum** besteht die Möglichkeit, Datentypen zu definieren, deren Wertebereich explizit durch Angabe einer Bezeichnerliste festgelegt wird. Die einzelnen Elemente eines Aufzählungstypen werden intern entsprechend ihrer Reihenfolge in dem Bereich der natürlichen Zahlen (0,1,...,n) abgebildet; sie müssen daher vom Typ **int** sein.

Allgemeine Form:

enum ***identifikator { bezeichnerliste };***

Hinweis:

° Mittels **enum** lassen sich Konstantendefinitionen bequem zusammenfassen, die ansonsten mit dem Präprozessorbefehl **#define** einzeln beschrieben werden müssen.

Programmbeispiel 10:

```
...
1  /* Zuordnung der numerisch definierten Farbpalette zu symbolischen                */
2  /* Konstanten mit 0 beginnend                                                     */
3  enum COLORS
4  {
5   BLACK,BLUE,GREEN,CYAN,RED,MAGENTA,BROWN,LIGHTGRAY,
6   DARKGRAY,LIGHTBLUE,LIGHTGREEN,LIGHTCYAN,LIGHTRED,LIGHTMAGENTA,
7   YELLOW,WHITE }

No errors detected
```

2.5.3. Datentypen mit const

Datentypen mit dem Zusatz·**const** befinden sich im Status einer Konstanten, die bei der Deklaration sofort initialisiert werden muss. Wertzuweisungen sind nicht erlaubt.

Allgemeine Form:

const ***speicherklasse typbezeichner identifikator = init-konstante;***

Hinweis:

° Bei einigen Compilerversionen ist die Datentype **const** noch nicht verfügbar

2.5.4. Datentypen mit volatile

Durch den Zusatz· volatile wird dem Compiler mitgeteilt, dass der Inhalt einer Variablen sich auch ohne explizite Wertzuweisung durch Interrupt-Routinen oder einen I/O-Port ändern kann. **volatile** stellt praktisch das Gegenteil von **const** dar.

Allgemeine Form:

volatile ***speicherklasse typbezeichner identifikator = konstante;***

Hinweis:

° Enthält eine Variable die momentane Zeit, so wird sie durch den Zusatz **volatile** regelmäßig aktualisiert.
° Bei einigen Compilerversionen ist die Datentype **volatile** nicht verfügbar

2.6. Initialisierung

Bei der Definition einer Variablen kann sofort eine Anfangswertzuweisung, eine Initialisierung, vorgenommen werden.

Allgemeine Form:

speicherklasse typvereinbarung identifikator = konstante;

Hinweis:

° Variable mit Speicherklassen **static** und **extern** werden vom Compiler implizit mit 0 initialisiert
° Variablen mit **register** und **auto** vom Compiler keine Initialisierung mit Null durchgeführt; der Wert der Variablen hat bei Eintritt in den Block einen nicht vorhersehbaren Wert

Programmbeispiel 11:

```
...
1  /* Beispiele fuer Initialisierungen einfacher Variablen        */
2   int ext_var;         /* externe Ganzzahlvariable mit 0 init .   */
3   main()
4   {
5    static int null;     /* statische Ganzzahlvariable mit 0 init.  */
6    int int_var = 32767; /*   Ganzzahlvariable                     */
7    short short_var = 32767; /*   Ganzzahlvariable                 */
8    long long_var = 2147483647; /*   Ganzzahlvariable              */
9    float float_var = 3.4e+38; /* Gleitkommavariable               */
10  double double_var = 1.7e+308; /* Gleitkommavariable              */
11  }
 No errors detected
```

3. Operatoren, Ausdrücke, Anweisungen

Operatoren dienen der Verknüpfung und Manipulation von Daten. C verfügt über eine große Anzahl leistungsfähiger Operatoren mit einer großen Mächtigkeit.
Ausdrücke (expressions) bestehen aus Operatoren und Operanden, die Variablen, Konstanten oder wiederum Ausdrücke sein können.

Je nach Art der Verknüpfung kann man verschiedene Klassen von Operatoren unterscheiden:

° die einstelligen, die unären Operatoren
° die zweistelligen, die binären Operatoren sowie
° die Zuweisungsoperatoren

Ein Ausdruck kann sein
° ein Name
° eine Konstante
° eine Zeichenkette
° ein Array
° eine Funktion
° ein Element einer Struktur
° eine Vereinigung (**union**)
° ein in Klammern gehaltener Ausdruck.

Bei der Angabe von Ausdrücken und der unären Operatoren haben die sog. lvalues (leftvalues) eine besondere Bedeutung. Ein lvalues ist ein Ausdruck, der auf ein Objekt verweist.

3.1. Priorität der Abarbeitungsfolge

Bei der Berechnung von Ausdrücken muss bei deren Programmierung die Reihenfolge der Abarbeitung beachtet werden.
Vorrang (Priorität) und Auswertungsrichtung (Assoziativität) sind der nachfolgenden Tabelle zu entnehmen, wobei die Priorität von oben nach unten abnimmt.

Art des Operators	Operator	Abarbeitung von
1. Operatoren. f. Felder, Strukturen,Funktionen	**( ) [ ] -> .**	links nach rechts
2. Einstellige Operatoren	**! ~ ++ -- + - (typ) * &**	rechts nach links
3. Zweistellige Operatoren	*** / %**	links nach rechts
	+ -	
	<< >>	
	< > <= >= == !=	
	&	
	^	
	|	
	&&	
	||	
	?:	
4. Zuweisungsoperatoren	**= *= /= += -= %= <<= >>= &= |= ^=**	rechts nach links
5. Kommaoperator	**,**	links nach rechts

Hinweis:

° Es sollte grundsätzlich vermieden werden Ausdrücke zu kodieren, die von der Auswertungsreihenfolge abhängen. Durch die speziellen Gegebenheiten der verschiedenen Compilerimplementationen der jeweiligen Rechnerarchitektur ist eine eindeutige Auswertung der Folge der Operatoren nicht gegeben. Das kann insbesondere dann zu Problemen führen, wenn diese Ausdrücke Nebeneffekte enthalten, die vor allem durch Inkrement- und Dekrementoperatoren verursacht werden.

3.2. Die einstelligen, die unären Operatoren

Auf den meisten Rechnerarchitekturen können Operationen mit derartigen Operatoren durch den Compiler effektiv auf entsprechende Maschineninstruktionen abgebildet werden.

3.2.1. Adressoperatoren

Der Adressoperator **&** liefert die Adresse eines Objektes,
& lvalue.

Das Ergebnis ist ein Zeiger auf das Objekt.

Der IndirektAdressoperator ***** kann eine Adressbezugnahme auflösen. Es wird auf das Datenelement 'indirekt' auf seinen Inhalt zugegriffen, auf das der Operand verweist,
**** zeiger.***

3.2.2. Arithmetische Operatoren

Folgende unäre arithmetische Operatoren sind verfügbar:

+ausdruck	>	der positive Wert von ausdruck wird bestimmt
-ausdruck	>	der negative Wert von ausdruck wird bestimmt
++ausdruck	>	Inkrementoperator als Präfixnotation; der Wert von ausdruck wird vor seiner Verwendung um 1 erhöht
ausdruck++	>	Inkrementoperator als Postfixnotation; der Wert von ausdruck wird nach seiner Verwendung um 1 erhöht
--ausdruck	>	Dekrementoperator als Präfixnotation; der Wert von ausdruck wird vor seiner Verwendung um 1 vermindert
ausdruck--	>	Dekrementoperator als Postfixnotation; Wert von ausdruck wird nach seiner Verwendung um 1 vermindert

Hinweis:

° Die Verwendung von Inkrement- und Dekrementoperatoren erzeugen einen günstigen Objektcode, da sie durch geeignete Maschineninstruktionen direkt am Ort ausgeführt werden. Sie werden effektiv von entsprechenden Maschinenbefehlen abgearbeitet.

° Der Ausdruck ***zielvar = quellvar -- ;***
bedeutet ***zielvar = quellvar;***
quellvar = quellvar - 1;
während dagegen ***zielvar = -- quellvar;***
die Anweisungen ***quellvar = quellvar - 1;***
zielvar = quellvar;
ergibt.

3.2.3. Logische Operatoren

Logische Operatoren werden ebenfalls direkt am Ort ausgeführt.

!ausdruck -> Der Negationsoperator erzeugt als Ergebnis 1, wenn der Wert des Ausdrucks 0 ist; das Ergebnis ist 0 , wenn der Wert des Ausdrucks ungleich 0 ist.

Hinweis:

- In C gibt es keine logischen Variablen. Als ***ausdruck*** sind hier nur Identifikatoren vom Typ **char** oder **int** zu verstehen.
- Die Wirkung des Negationsoperators kann ausgenutzt werden, um den Wahrheitswert von Ausdrücken festzustellen

3.2.4. Bitoperator

Die bitorientierten Operatoren ermöglichen die direkte Manipulation einzelner Bits.

~ausdruck ->Bildung des bitweisen Einerkomplementes, das Vertauschen der Bitbelegung seines Operanden

3.2.5. cast - Operator

Über eine sog. cast-Konstruktion wird eine Typumwandlung explizit erzwungen. Die Wirkung einer solchen Anweisung entspricht praktisch einer Wertzuweisung der Form:
lvalue = (typ)ausdruck;

typ gibt den Zieltyp an, in den ***ausdruck*** konvertiert werden soll.

Programmbeispiel 12:

```
                                                        PAGE   1
                                                      12-23-90
                                                      22:58:57

Line#  Source Line                          Microsoft C Compiler Version 5.10

    1  /* Beispiel zur Verwendung eines cast - Operators zur expliziten          */
    2  /* Typumwandlung von Daten                                                */
    3  #include <math.h>
    4  main ()
    5  {
    6    int ganz_var;
    7    double gleit_var;
    8    gleit_var = sqrt(7.0);
    9    ganz_var = (int)gleit_var;  /* ganz_var = 2 und gleit_var = 2.645751     */
   10  }

Global Symbols
Name                  Class   Type            Size   Offset
main. . . . . global  near function     ***    0000
sqrt. . . . . .extern  near function     ***     ***
No errors detected
```

Hinweis:

° Anwendung findet die explizite Typumwandlung mittels des cast-Operators vor allem bei Funktionsaufrufen, wenn eine Angleichung des Argumenttyps an den Parametertyp erforderlich ist

3.2.6. sizeof - Operator

Der Längenoperator **sizeof** liefert einen **unsigned-**Wert, der den Speicherplatz in Bytes angibt, den der Operand belegt.

Folgende Formen sind möglich:
sizeof *ausdruck*
sizeof (***typ***)

sizeof (***typ***) ermittelt den Speicherplatzbedarf, den ein Objekt des angegebenen Typs in der entsprechenden C -Implementierung benötigt.

Hinweis:

° Ein durch **sizeof** gebildeter Ausdruck ist ein Konstantenausdruck und wird bereits bei der Compilierung ausgewertet

Programmbeispiel 13:

```
                                                                    PAGE   1
                                                                    05-26-91
                                                                    19:10:04

Line#  Source Line                          Microsoft C Compiler Version 5.10

    1  /*  Beispiel 13                                                      */
    2  /* Ermittlung des Speicherplatzbedarfs einfacher Objekte sowie       */
    3  /* eines Feldes                                                     */
    4  double feld[1000];
    5  main()
    6  {
    7    int laen_int,laen_float,laen_double,laen_feld;
    8    laen_int = sizeof(int);              /* Laenge = 2                */
    9    laen_float = sizeof(float);          /* Laenge = 4                */
   10    laen_double = sizeof(double);           /* Laenge = 8             */
   11    laen_feld = sizeof(feld);            /* Laenge = 8000             */
   12    printf(
   13    "\n laen_int = %d laen_float = %d laen_double = %d laen_feld = %d",
   14        laen_int,    laen_float,    laen_double,    laen_feld);
   15  }

main  Local Symbols
Name                 Class   Type           Size   Offset  Register

laen_int. . . . . .auto                      -0008
laen_double . . . . .auto                    -0006
laen_feld . . . . . auto                     -0004
laen_float. . . . . auto                     -0002

Global Symbols
Name                 Class   Type           Size    Offset

feld. . . . . common  struct/array     8000     ***
main. . . . . global  near function     ***     0000
printf. . . . . extern  near function   ***      ***
No errors detected
```

3.3. Die zweistelligen, die binären Operatoren

Hier stehen alle vier Grundrechenarten, Bitoperationen, logische sowie Vergleichsoperationen zur Verfügung.

Die zweistelligen Operatoren sind in folgender Form anwendbar:
operand operator operand

3.3.1. Arithmetische Operatoren

Für ***operator*** gilt:

- **+** Addition
- **-** Subtraktion
- ***** Multiplikation
- **/** Division
- **%** Modulo - Operator, Rest einer ganzzahligen Division

3.3.2. Vergleichsoperatoren

Für ***operator*** gilt:

- **<** kleiner
- **>** größer
- **<=** kleiner gleich, nicht größer
- **>=** größer gleich, nicht kleiner
- **==** gleich
- **!=** nicht gleich

3.3.3. Bitoperatoren

Für ***operator*** gilt:

- **<<** verschieben der Bits nach links um n Stellen
- **>>** verschieben der Bits nach rechts um n Stellen
- **&** bitweise konjunktive UND - Verknüpfung
- **|** bitweise inklusive ODER - Verknüpfung
- **^** bitweise exklusive ODER - Verknüpfung

Die Operanden bitorientierter logischer Operatoren werden in jeder Bitposition nach folgenden Regeln miteinander verknüpft:

var1	var2	var1 & var2	var1 | var2	var1 ^ var2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Hinweis:

- die Operanden müssen vom Typ **char**, **short, int** oder **long** sein
- die Operatoren **<<** und **>>**, die Shift-Operatoren, können sinnvoll zur Multiplikation bzw. Division von Integer-Zahlen mit Zweierpotenzen benutzt werden
- bei Verschiebeoperationen muss der Verschiebewert positiv sein
- bei Linksverschiebung erfolgt ein Auffüllen mit 0

3.3.4. Zuweisungsoperatoren

Der generelle Zuweisungsoperator ist wie in anderen Programmiersprachen das Ergibtzeichen der Form
lvalue = ausdruck;

Eine erweiterte Variante des In- und Dekrementoperators sind die Zuweisungsoperatoren. Die häufig auftretende Wertzuweisung der Form
lvalue = lvalue op (ausdruck);

können verkürzt in der allgemeinen Form
lvalue op = ausdruck;

geschrieben werden. ***op*** kann einer der binären Operatoren
+, -, *, /, %, &, |, ^, <<, >>
sein.

Programmbeispiel 15:

```
...
3   /* Demonstration  Zuweisungsoperatoren          */
4   int var1 = 1, var2 = 2, var3 = 3, var4 = 4;
5   var1 += var2 *= var3 = var4 >>= var1;
6   /* Abarbeitung von rechts nach links           */
7   /*     var4 wird nach der Verschiebung 2       */
8   /*     var3 wird nach der Wertzuweisung 2      */
9   /*     var2 wird nach der Multiplikation 4     */
10  /*     var1 wird nach der Addition 5           */
No errors detected
```

Hinweis:

- Der Compiler muss bei der Form ***lvalue op = ausdruck*** den Ausdruck links vom Zuweisungsoperator (***lvalue***) nur einmal bewerten. Es werden durch diese Art der Programmierung Voraussetzungen geschaffen, effizienten Objektcode zu generieren.
- Bei der Zerlegung von mehreren Zeichen in Operatoren geht der Compiler nach dem Grundsatz vor, die nächste größtmögliche Zeichenkette für die zusammengesetzten Operatoren zu generieren.

Programmbeispiel 15a:

```
Line#  Source Line                    Microsoft C Compiler Version 5.10

...
 4  /*  Maximum mampfen, engl. maximal munch  als Richtlinie bei der   */
 5  /*  Interpretation von mehreren Zeichen als Operatoren              */
 6   int op1, op2, ziel;
 7   op1 = 7;
 8   op2 = 3;
 9   ziel  = op1 --- op2;
10  /* wird interpretiert als  ziel = op1 -- - op2;                     */
11  /* und nicht als          ziel = op1 - -- op2;                     */
12  /*                       ziel = 4, op1 = 6, op2 = 3                */
13   ziel  = op1 --> 0;
14  /* wird interpretiert als  ziel = op1 -- > 0;                      */
15  /* und nicht als          ziel = op1 - -> 0;                       */
16  /*                        ziel = 1, op1 = 5                        */
...
No errors detected
```

3.3.5. Logische Operatoren

Die Werte der Variablen werden als logische Werte true (wahr) oder false (falsch) interpretiert, wenn die Operatoren

&& -> konjunktiv (logisches UND) bzw.

|| -> disjunktiv (logisches ODER)

verwendet werden.

Die Wirkung dieser Operatoren kann ausgenutzt werden, um den Wahrheitswert von Ausdrücken festzustellen. Das Ergebnis ist immer ein Integerwert, der entweder true ist, wenn sein numerischer Wert ungleich 0 bzw false, wenn sein numerischer Wert gleich 0 ist.

Programmbeispiel 16:

```
...
 3  /* Hinweis auf Unterschied zu den Bitoperatoren                    */
 5   int int_var1 = 1, int_var2 = 2, erg;
 6   erg = int_var1 &  int_var2;        /* erg erhaelt den Wert  0      */
 7   erg = int_var1 && int_var2;        /* erg erhaelt den Wert  1      */
 8   erg = int_var1 | int_var2;         /* erg erhaelt den Wert  3      */
 9   erg = int_var1 || int_var2;        /* erg erhaelt den Wert  1      */
...
No errors detected
```

3.3.6. Bedingungsoperator

Der Bedingungsoperator ?: verknüpft drei Ausdrücke der Form:

ausdruck1 ? ausdruck2 : ausdruck3

Der Wert des Gesamtausdrucks ist vom Wahrheitswert des Vergleichsausdrucks ***ausdruck1*** abhängig und entspricht dem Wert von ***ausdruck2*** im Fall true bzw. ***ausdruck3*** im Fall false.

Programmbeispiel 17:

```
...
 3  /*  Ermittlung des absoluten Betrag mittels Bedingungsoperators    */
 4   int var = -99, abs_betrag;
 5   abs_betrag = var < 0 ? -var : +var;  /* Ergebnis ist +99           */
...
No errors detected
```

3.3.7. Kommaoperator

Zwei durch Komma getrennte Ausdrücke werden von links nach rechts ausgewertet.
ausdruck1 , ausdruck2

Typ und Wert des rechten Ausdrucks bestimmt den Typ und Wert des Gesamtausdrucks.

Hinweis:

° Kommaoperator darf nicht mit dem Trennzeichen Komma verwechselt werden.

Programmbeispiel 18:

```
...
 3  /* Demonstration der Wirkung des Kommaoperators                    */
 4   int var1, var2 =  99;
 5   var1 = 1, var2 += var1;
 6  /* var1 wird 1 und var2  wird um diesen Wert erhoeht               */
...
No errors detected
```

4. Steuerstrukturen

Steuerstrukturen werden verwendet, um die Reihenfolge der Abarbeitung von Anweisungen festzulegen. Außerdem ermöglichen es die sprachlichen Mittel zur Steuerung des Programmablaufs gut strukturierte Programme zu schreiben.
Im Sinne von C kann ein Programm, eine Funktion, als eine Sequenz von Steuerblöcken aufgefaßt werden.

Jeder Strukturblock kann

- eine einzelne Anweisung
  ausdruck;

- eine Verbundanweisung
  {
  folge von einzelnen anweisungen
  }

- eine Blockanweisung
  {
  vereinbarungsliste
  folge von einzelnen anweisungen
  }

sein.

Die Steueranweisungen zur Darstellung von Steuerstrukturen können in folgende Anwendungsgruppen unterteilt werden:

° die **if-else** Anweisung zur bedingten Steuerungsübergabe,
° die **do**-**while** Anweisung zur Schleifenbildung,
° die **while** Anweisung als 2.Variante zur Schleifenbildung,
° die **switch** Anweisung zur Mehrwegeauswahl,
° die universelle Schleifenanweisung **for,**
° die Anweisungen **break**, **continue** und **goto** zur unbedingten Steuerungsübergabe

4.1. Die bedingte Anweisung

Die bedingte, alternative Anweisung ist als **if**-**else** Anweisung realisiert.

Allgemeine Form:
if ***(ausdruck) anweisung_1***
oder
if ***(ausdruck) anweisung_1***
else ***anweisung_2***

Der Ausdruck ***ausdruck*** wird berechnet und als Verzweigungsbedingung bewertet. Ist das Ergebnis ungleich 0 (true), so wird ***anweisung_1*** ausgeführt. Wenn das Ergebnis gleich 0 (false) ist, so wird, falls vorhanden, ***anweisung_2*** im **else**-Zweig ausgeführt, bzw. die nachfolgende Anweisung.

Hinweis:

° beliebige Schachtelungen von **if**-**else** Konstruktionen sind erlaubt
° ein eventuell folgender **else**-Zweig wird dem letzten **else**-losen **if**-Zweig zugeordnet

Programmbeispiel 19:

```
...
 4  /* Das Maximum von drei vorgegebenen Konstanten ermitteln     */
 5   float wert1 = -23.45, wert2 = 23.45, wert3 = 0, maximum;
 6   if (wert1 > wert2)
 8   if (wert1 > wert3)  maximum = wert1;
 9   else            maximum = wert3;
10   else
11    if (wert2 > wert3) maximum = wert2;
12    else           maximum = wert3;
13    /* mittels logischer Operatoren elegantere Loesung              */
15    if (wert1 > wert2 && wert1 > wert3)    maximum = wert1;
16    else
17    if (wert2 > wert1 && wert2 > wert3) maximum = wert2;
18    else                        maximum = wert3;
19    /* maximum wird der Wert 23.45 zugewiesen                       */
 ...
No errors detected
```

4.2. Die Fallunterscheidung switch

Die **switch** Auswahl gestattet eine indizierte Verzweigung eines Programms. Abhängig vom Index (*ausdruck*) wird das Programm in dem entsprechenden Befehlsblock fortgesetzt.
Die Fallauswahl **switch** ist eine Erweiterung der Alternative auf mehr als zwei Fälle.

Allgemeine Form:

```
switch (ausdruck)
{
case konstanter_ausdruck_1  anweisungsliste_1
 case konstanter_ausdruck_2  anweisungsliste_2
...
 case konstanter_ausdruck_n  anweisungsliste_n
 default  anweisungsliste
}
```

Es wird der Ausdruck ***ausdruck*** mit den konstanten Ausdrücken ***konstanter_ausdruck_n*** verglichen und bei Gleichheit wird die Anweisungsliste ***anweisungsliste_n*** ausgeführt. Wird in keiner **case**-Klausel Gleichheit gefunden, so wird, falls vorhanden, die **default**-Klausel ausgeführt. Das Verlassen einer **case**-Klausel muss mit geeigneten Anweisungen (z.B. **break**) selbst organisiert werden.

Hinweis:

- das Resultat des Schalterausdrucks ***ausdruck*** muss ein **int** Wert sein
- ***konstanter_ausdruck_n*** muss eine **int**-Konstante sein; bzw. der Ausdruck muss zur Compilerzeit berechenbar sein
- die **default**-Klausel ist nicht zwingend vorgeschrieben

Programmbeispiel 20:

```
...
 4  /* Demonstration des Ablaufs einer switch - Anweisung                */
 5  /* ohne vorzeitiges Verlassen der Anweisung                          */
 6   int variable = 1,flag = 3;
 7   switch(flag)
 8   {
10   case 1:;
11   case 2:  variable ++;
12   case 3:  variable ++;
13   case 4:  variable ++;
14   default: variable --;
15   }
16    /* variable hat nach Verlassen der switch-Anweisung den Wert 2     */
```

4.3. Programmschleifen

Zur Bildung von Programmschleifen sind drei Möglichkeiten vorhanden.

4.3.1. Die Abweisschleife - die while Anweisung

Bei der **while**-Anweisung erfolgt der Test der Abbruchbedingung am Schleifenanfang.

Allgemeine Form:
while *(ausdruck) anweisung*

Die **while** - Kontrollstruktur sorgt dafür, dass der Anweisungsblock ***anweisung*** solange wiederholt wird, solange der Ausdruck ***ausdruck*** ungleich 0 ist. Ist er gleich 0, wird die **while-**Struktur verlassen.

Hinweis:

- bei jeder Wiederholung wird ***ausdruck*** neu berechnet und ausgewertet
- ist Ausdruck schon bei Eintritt in die **while**-Anweisung gleich 0, wird die **while**- Anweisung sofort verlassen
- mittels **break** kann die **while**-Anweisung vorzeitig verlassen werden
- mittels **continue** wird vorzeitig die Wiederholung der **while**-Anweisung eingeleitet

Programmbeispiel 21:

```
Line#  Source Line                          Microsoft C Compiler Version 5.10

     1   /* Berechnung der Fakultaet einer ganzen Zahl mittels der            */
     2   /*  while Anweisung                                                  */
     3   main()
     4   {
     5   double fakultaet = 1;
     6   int ganz_wert = 170; /* größter möglicher Wert zur Berechnug der     */
     7   /* Fakultaet fuer double Werte                                      */
     8   /* int ganz_wert = 1754;  groeßter möglicher Wert zur Berechnug der  */
     9   /* Fakultaet fuer long double Werte                                 */
    10   while(ganz_wert)
    11   fakultaet *= ganz_wert--;
    12
    13   /* Erg. 170! = 7.257415615308e306       fuer double Werte           */
    14   /* 171! fuehrt zur Ausschrift -- Floating point error : Overflow    */
    15   /* Erg. 1754! = 1.97926189010501005e4930 fuer long double Werte     */
    16   /* 1755! fuehrt zur Ausschrift -- Floating point error : Domain     */
No errors detected
```

4.3.2. Die Nichtabweisschleife - die do Anweisung

Bei der **do**-Anweisung erfolgt der Test der Abbruchbedingung am Schleifenende nach jedem Durchlauf.

Allgemeine Form:
do ***anweisung***
while ***(ausdruck)***

Der Ausdruck ***ausdruck*** wird nach Abarbeitung der Anweisung ***anweisung*** berechnet und solange ausgeführt, solange der Ausdruck ungleich 0 (true) ist. Ist er gleich 0 (false), so wird die **do**-Anweisung verlassen.

Hinweis:

- bei jeder Wiederholung wird ***ausdruck*** neu berechnet und ausgewertet
- ist Ausdruck schon bei Eintritt in die **while**-Anweisung gleich 0, wird die Anweisung einmal ausgeführt
- mittels **break** kann die **do**-Anweisung vorzeitig verlassen werden
- mittels **continue** kann vorzeitig zur Wiederholung der **do**-Anweisung übergegangen werden

Programmbeispiel 22:

```
...
Line#  Source Line                          Microsoft C Compiler Version 5.10

     1  /* Berechnung der Fakultät einer ganzen Zahl mittels der          */
     2  /* do-while Anweisung                                             */
     3  main()
     4  {
     5  double fakultaet = 1;
     6  int ganz_wert = 170; /* groeßter moeglicher Wert zur Berechnug der        */
     7  /* Fakultaet fuer double Werte                                    */
     8
     9  do
    10  fakultaet *= ganz_wert;
    11  while(--ganz_wert != 0);
    12
    13  /* Ergebnis 170! = 7.257415615308e306  für double Werte          */
    ...
No errors detected
```

4.3.3. Die verallgemeinerte abzählbare Schleife, die for Anweisung

Die allgemeinste Form der Schleifenbildung ist durch die **for**- Anweisung realisiert.

Allgemeine Form:
for ***(ausdruck_1; ausdruck_2; ausdruck_3)***
anweisung

Nach der Initialisierung (***ausdruck_1***) erfolgt die Überprüfung der Bedingung (***ausdruck_2***). Ist des Ergebnis gleich 0, wird die Schleife sofort verlassen, andernfalls die Anweisung ***anweisung*** ausgeführt. Nach Ausführung von ***anweisung*** wird ***ausdruck_3*** berechnet und ***ausdruck_2*** wiederum ausgeführt.

Hinweis:

- ***ausdruck_1*** dient der Initialisierung der Laufvariablen, der Anfangsausdruck
- ***ausdruck_2*** ist die Abbruchbedingung. Ist ***ausdruck_2*** gleich 0 -> Abbruch, ist er ungleich 0 -> Abarbeitung ***ausdruck_3***
- ***ausdruck_3*** berechnet den nächsten Wert der Laufvariablen
- mittels **break** kann die **for**-Anweisung vorzeitig verlassen werden
- mittels **continue** kann vorzeitig zur Wiederholung der **for-**Anweisung übergegangen werden, wobei vorher ***ausdruck_3*** neu berechnet wird

Programmbeispiel 23:

```
.....
 1  /* Nach dem Mathematiker Euklid werden Zahlen, die aus der Summe aller    */
 2  /* ihren echten Teiler besteht, als vollstaendige Zahlen bezeichnet.      */
 3  /* Dieses Beispielprogramm sucht alle vollstaendigen Zahlen in einem      */
 4  /* Intervall von 0 bis 10000                                             */
 5  main()
 6  {
 7  register int reg_i,reg_j;
 8  int intervall = 10000, vollzahl, summe;
 9  for (reg_i = 3; reg_i <= intervall; reg_i++)
10 {
11 summe = 1;
12 for (reg_j = 2; reg_j <= reg_i-1; reg_j++)
13 if (reg_i % reg_j == 0)   summe += reg_j;
14 if (reg_i == summe)     vollzahl = reg_i;
15 }
16 /* Fuer dieses Intervall werden die Zahlen 6, 28, 496 und 8128 als       */
17 /* vollstaendige Zahlen ermittelt                                        */
18 }
No errors detected
```

4.4. Die Unterbrechungsanweisung - die break Anweisung

Die **break**- Anweisung bewirkt das Verlassen der innersten umfassenden Schleife.
Allgemeine Form:
break;

Hinweis:

- Verlassen von inneren Schleifen für **for**-,**do**- und **while**- Anweisungen
- Verlassen von **switch**-Anweisungen

Programmbeispiel 24:

```
Line#  Source Line                    Microsoft C Compiler Version 5.10

  1  /* Demonstration des Ablaufs einer switch - Anweisung          */
  2  /* mit vorzeitigem Verlassen der Anweisung                     */
  3  main()
  4  {
  5  int variable = 1;
  6  int flag = 3;
  7  switch(flag)
  8  {
  9  case 0:
 10 case 1:; break;
 11 case 2:  variable ++; break;
 12 case 3:  variable ++; break;
 13 case 4:  variable ++; break;
 14 case 5:  variable ++; break;
 15 default: variable --;
 16  }
 17  /* variable hat nach Verlassen der switch-Anweisung den Wert 2       */
No errors detected
```

4.5. Die Fortsetzungsanweisung - die continue Anweisung

Die **continue** -Anweisung bewirkt die Neuinitialisierung der innersten umfassenden Schleife.

Allgemeine Form:
continue;

Hinweis:

- Beginn der nächsten Iteration der umgebenden Schleifen für **for**-,**do**- und **while**- Anweisungen
- bei **for**-Anweisungen wird vorher ***ausdruck_3*** berechnet
- bei **while** - und **do-while**-Anweisungen wird vorher der **while**-Ausdruck (Schleifentest) durchgeführt
- **continue** wirkt nicht in s**witch**-Anweisungen

Programmbeispiel 25:

```
                                                    PAGE   1
                                                    12-28-90
                                                    19:04:09

Line#  Source Line                  Microsoft C Compiler Version 5.10

     1  /* Es soll die Summe der Formel 1:(i*i - 1) fuer den Wertebereich          */
     2  /* (-3 <= i <= 3 )   berechnet werden                                     */
     3  main()
     4  {
     5  float summe = 0, i;
     6  for ( i = -3; i <= 3; i++)
     7  {
     8  if (i == -1 || i == 1) continue; /* Division durch 0 ausschließen    */
     9  summe +=  1.0 / (i * i - 1);
    10  }
    11  /* Das Ergebnis betraegt -0.0833333                                       */
...
No errors detected
```

4.6. Die Sprunganweisung - die goto Anweisung

Die **goto** -Anweisung bewirkt einen unbedingten Sprung zu der durch ***marke:*** markierten Anweisung.

Allgemeine Form:
goto *marke;*
marke:

Die Anwendung der **goto**-Anweisung ist eigentlich nur dann sinnvoll, um abhängig von einer Bedingung aus tieferen Schachtelungsstufen nach außen zu gelangen.

Hinweis:

- jede Anweisung kann markiert sein
- für Marken gelten die gleichen Regeln der Namensbildung wie für Identifikatoren
- Marken werden nicht vereinbart; sie sind durch den Doppelpunkt hinreichend charakterisiert
- Marken müssen sich im gleichen Block wie die **goto**-Anweisung befinden

5. Zeiger

Unter einem Zeiger (pointer) versteht man ein Datenobjekt, das die Adresse eines Speicherbereichs enthält, in dem der Wert einer Variablen gespeichert ist. Neben der Adresse verfügt der Zeiger noch über ein Längenattribut, das die Länge des Datenobjektes in Bytes spezifiziert, auf das der Zeiger zeigt.
Der Indirektoperator (*****) spezifiziert die angegebene Variable als Zeigervariable, während der Adressoperator (**&**) die Adresse des Objekts bildet, die der Zeigervariablen zugewiesen werden kann.

Man sagt, ein 'Zeiger zeigt auf ein Objekt', wenn
zeiger = &objekt;
gilt.

5.1. Zeigervariable

Zeigervariable werden zu Beginn eines Blockes vereinbart, wobei der Indirektoperator (*****) vor dem Identifikator die Variable als Zeigervariable spezifiziert.
Allgemeine Form:
typ zeigerzusatz *identifikator;
wobei ***zeigerzusatz*** aus den Schlüsselwörtern **near**, **far** oder **huge** bestehen kann.

Erläuterungen:

° ***typ*** spezifiziert den Typ des Datenobjektes, deren Adresse der Zeiger aufnehmen soll
° ***zeigerzusatz*** bezieht sich auf das verwendete Datenmodell
- kein Zeigerzusatz verwendet den 'normalen' Zeiger; d.h. der Zeiger kann jede Speichestelle innerhalb eines Segmentes adressieren
- Zeigerzusatz **·near** wirkt wie kein Zeigerzusatz; man kann sich aber bei großen Speichermodellen (**compact, large, huge**) einen 'nahen' Zeiger erzwingen
- Zeigerzusatz **far** oder **huge** kann auf Objekte außerhalb eines Segmentes zugreifen (z.B. Bildschirmspeicher)
- **huge** - Zeiger können miteinander verglichen werden
- in den großen Speichermodellen **compact**, **large** und **huge** sind die Zeiger immer **far**, solange kein anderer Zeigerzusatz angegeben ist (Näheres zu Zeigerzusätzen siehe Punkt 5.4)

Hinweis:

° Zeigermodifikationen sind nicht portabel. Wenn der Einsatz des Programms auf anderen Rechnern unter anderen Compilern geplant ist, sollte man auf Zeigerzusätze verzichten und stattdessen eines der großen Speichermodelle verwenden.
° das Zeigerkonzept kann nicht auf Variable der Speicherklasse **register** angewendet werden, da von Registern keine Adressen gebildet werden können

5.2. Zeigeroperationen - Adressrechnungen

Unter Adressrechnungen werden Operationen mit den Adressen verstanden, die in den Zeigervariablen gespeichert sind.

Hinweis:

° alle arithmetischen Operationen werden unter Beachtung der Längenattribute der Zeiger durchgeführt; der arithmetische Operand ist eine Objektlänge (**sizeof*(objekt)***)

Folgende Operationen mit Zeigern sind erlaubt:

1. Die Addition einer ganzen Zahl zum Zeigerwert; der Zeigerwert wird um eine ganze Zahl von Datenobjekten vorwärts gerückt; im Fall ***zeiger++*** wird der Zeiger inkrementiert, um auf das nächste Objekt zu zeigen.
   zeiger++ bzw. ***zeiger = zeiger + 1*** bzw. ***zeiger += 1***
   bedeutet
   zeiger + 1 -> ***zeiger + (1 *** **sizeof*(objekt))***
2. Die Subtraktion einer ganzen Zahl vom Zeigerwert; der Zeigerwert wird um eine ganze Zahl von Datenobjekten rückwärts gesetzt.
   zeiger-- bzw. ***zeiger = zeiger - 1*** bzw. ***zeiger -= 1***
   bedeutet
   zeiger - 1 -> ***zeiger - (1 *** **sizeof*(objekt))***
3. Subtraktion von zwei Zeigern; Die Differenz ist die Anzahl der Datenobjekte, die sich zwischen den Zeigern befindet.
4. Vergleich von Zeigern; (Identität, Relationsbetrachtungen)
5. Wertzuweisung des Zeigerwertes **NULL**; die Zeigerzuordnung wird aufgehoben.
6. Vergleich eines Zeigers mit **NULL**; besitzt ein Zeiger den Wert **NULL**, so zeigt der Zeiger auf kein Objekt.

Folgende Operationen mit Zeigern sind **nicht** erlaubt:

1. Zwei Zeiger können nicht addiert, nicht multipliziert, nicht dividiert, nicht verschoben oder mit logischen Operationen verändert werden. Selbstverständlich sind Operationen derart ausführbar: ****zeiger = *zeiger * *zeiger***. Hier werden lediglich zwei Variableninhalte miteinander multipliziert.
2. Es können keine Operationen mit **double**- oder **float**-Werten ausgeführt werden.

5.3. Zeigerinitialisierungen

Initialisierungen von Zeigern geschieht mit Adressen vorher definierter Objekte.

Allgemeine Form:
typ zeigerzusatz *identifikator = &objekt;

Programmbeispiel 26:

```
...
 6  /*  Einfache Demonstration zur Arbeit mit Zeigern          */
 7  int quell[3]={2000,3000,4000},ziel;
 8  int *ptr = quell;          /* Zeigerinitialisierung        */
 9  /* Inhalt von ptr:  *ptr = 2000                            */
10  /* Adresse von ptr:  ptr = DS:FFD0                         */
11
12  ptr +=2; /* ptr 2 Datenobjekte weiter, quell[2]            */
13  /* Inhalt von ptr:  *ptr = 4000                            */
14  /* Adresse in ptr:   ptr = DS:FFD4                         */
15
16  ziel = *ptr; /* Inhalt von ziel = 4000                     */
17
18  ptr --; /* ptr ein Datenobjekt zurueck,quell[1]            */
19  /* Inhalt von ptr:  *ptr = 3000                            */
20  /* Adresse in ptr:   ptr = DS:FFD2                         */
21
22  *ptr = 5000; /* Wertzuweisung ueber ptr                    */
23  /* Inhalt von ptr:  *ptr =  5000                           */
24  /* quell[1] = 5000                                         */
No errors detected
```

5.4. Zeigerzusätze

Die Sprache C bietet Zeigerzusätze, die aus der besonderen Speicherverwaltung des INTEL- Prozessors 80x86 resultieren. Normalerweise legt C den Typ verwendeter Zeiger über das verwendete Speichermodell fest. Durch die Verwendung entsprechender Modifizierer bei der Deklaration der Zeiger können Zeigertypen auch unabhängig vom Speichermodell festgelegt werden.

Folgende Zeigerzusätze sind möglich:

- **near** Zeiger stellen nur einen Offset (16 Bit) dar und sind deshalb vom jeweils verwendeten Segmentregister abhängig. Für die kleinen Speichermodelle ist das der Normalwert.
- **far** Zeiger enthalten eine vollständige Adresse im Format Segment:Offset (32 Bit). Es wird dadurch eine Adressierung außerhalb eines Segments möglich. In den großen Speichermodellen sind die Zeiger immer **far**, wenn nicht definitiv **near** verlangt wird.
- **huge** Zeiger werden ebenfalls wie **far**-Zeiger dargestellt. Sie liegen immer in normalisierter Form vor, d.h. jede absolute Speicheradresse läßt nur eine einzige Kombination von Segment:Offset zu. (Identitätsvergleich)

Programmbeispiel 27:

```
...
3  /* Das verwendetet Speichermodeml ist 'medium'. Die Standardzeiger        */
4  /* haben eine Laenge von 16 Bits                                          */
5  int *ptr1;              /*  ein near-Pointer, Laenge 16 Bits             */
6  int near *ptr2;         /*  ein near-Pointer, Laenge 16 Bits             */
7  int far *ptr3;          /*  ein far-Pointer,  Laenge 32 Bits             */
8  int huge *ptr4;         /*  ein huge-Pointer, Laenge 32 Bits             */
9  }

main  Local Symbols

Name              Class   Type          Size   Offset  Register
ptr4. . . . . auto                      -000c
ptr3. . . . . auto                      -0008
ptr2. . . . . auto                      -0004
ptr1. . . . . auto                      -0002
No errors detected
```

5.5. Dynamische Speicherverwaltung

Alle bisher kennengelernten Variablen sind statisch, d.h. sie werden schon beim Programmstart einer festen Speicheradresse zugeordnet. Die Anzahl der Elemente eines Feldes muss bereits zu diesem Zeitpunkt definitiv angegeben werden und kann zu einem späteren Zeitpunkt nicht mehr verändert werden.
Eine besonders sinnvolle Anwendung der Zeigerverarbeitung besteht in der dynamischen Speicherverwaltung. Die gesamte dynamische Speicherverwaltung beruht auf Zeigern. Wird zu einem bestimmten Zeitpunkt eine Speicherreservierung vorgenommen, so wird über geeignete Funktionen die Adresse des reservierten Speicherbereichs mitgeteilt, die natürlich über Zeiger weiter verarbeitet werden kann.

Folgende Funktionen können zur dynamischen Speicherzuordnung herangezogen werden:

- **malloc*(groesse)*** wird zur Anforderung nicht belegten Speicherplatzes benutzt. **malloc** reserviert einen Bereich bestimmter Groesse und liefert die Startadresse dieses Bereichs.
- **calloc*(anzahl, groesse***) ist eine Erweiterung von **malloc**. Ein Speicherbereich wird ***anzahl*** mal ***groesse*** belegt und im Gegensatz zu **malloc** mit Nullen initialisiert und liefert die Startadresse dieses Bereichs.
- **realloc*(zeiger,groesse)*** ändert die Größe eines dynamisch belegten Speicherbereichs auf den neuen Wert ***groesse***.

Hinweis:

- Alle Funktionen reservieren Speicherbereich auf dem C-Speicherheap. Kann der angeforderte Speicher nicht zur Verfügung gestellt werden, wird stattdessen **NULL** zurückgegeben.
- Alle Funktionen liefern als Funktionswert einen sog. **void**-Zeiger, d.h. er kann ohne cast- Operator direkt zugewiesen werden.
- Die Art des Zeigers (**near** oder **far**) wird automatisch gesetzt.
- Nicht mehr benötigter Speicherbereich sollte mittels der Funktion **free*(zeiger)*** freigegeben werden.

Programmbeispeil 28:

```
...
 1  /* Einfache Demonstration der Arbeit mit dynamischer Speicherverwaltung */
 2  #include <malloc.h>   /* Definition der Speicherverwaltungsfunktionen       */
 3  /* In TURBOC #include <alloc.h>                                            */
 4  #include <stdlib.h>
 5  #include <stdio.h>
 6  main()
 7  {
 8  long *quadrate,i;
 9  quadrate =  calloc(100 , sizeof(long));
10  /* Es wird die Adresse eines Feldes aus dem C Speicherheap                 */
11  /* zur Aufnahme von 100 long-Werten  uebergeben                            */
12  if (quadrate == NULL) exit(1);
13  for (i = 1; i <= 100; i++)
14  {
15  /*quadrate = i * i ;   /* Es werden die Quadrate von 1 bis 100             */
16  /* in diesem Feld abgelegt                                                 */
17  quadrate++;
18  }
19  for (i = 1; i <= 100; i++)
20  {
21  --quadrate;          /* Feld rueckwaerts lesen                             */
22  /* Ausgabe der Quadrate von 1 bis 100                                      */
23  }
24  }
```

```
Global Symbols

Name                 Class   Type              Size   Offset

calloc. . . . . .extern  near function      ***     ***
exit. . . . . extern  near function      ***     ***
main. . . . . global  near function      ***     0000
No errors detected
```

6. Mehrdimensionale Felder

In der Sprache C sind mehrdimensionale Felder möglich. Die Definition eines Feldes siehe Punkt 2.1.3 Die Anordnung der Feldelemente erfolgt zeilenweise; d.h., beim Übergang zum nächsten Feldelement ändert sich der letzte Index am schnellsten, da auf die Elemente in der Reihenfolge ihrer Speicherung zugegriffen wird.
Die Speicherung der Feldelemente erfolgt lexikographisch.
In der Sprache C ist ein mehrdimensionales Feld per Definition tatsächlich ein eindimensionales Feld, dessen Elemente selbst Felder mit gleichen Attributen sind. Deshalb werden die Dimensionen und die Indizes jeweils getrennt in eckigen Klammern angegeben.

Allgemeine Form der Typvereinbarung eines mehrdimensionalen Feldes:
speicherklasse typ identifikator [1.feld][2.feld]...[n.feld];

Hinweis:

- Die Bezugnahme auf einzelne Feldelemente ist über mehrere Indizes möglich
- Im Gegensatz·zu anderen problemorientierten Programmiersprachen werden einzelne Indizes in der Form ***array [i] [j]...[n]*** angesprochen
- Jede Indexoperation entspricht einer Zeigeroperation, so daß man das Verhalten von Indizes vollständig mit Hilfe von Zeigern beschreiben kann
- In C sind nur Operationen für einfache Datentypen definiert, d.h, es gibt keine Feldoperationen. Notwendige Feldmanipulationen müssen mit geeigneten Anweisungen für einzelne Feldememente, z.B. die **for**-Anweisung durchgeführt werden
- der C-Compiler übt keinerlei Kontrolle über die angegebene Anzahl der Dimensionen sowie über deren Indizes aus

Programmbeispiel 29:

```
                                                        PAGE 1
                                                      01-15-91
                                                      22:22:55

Line#  Source Line                  Microsoft C Compiler Version 5.10

    1  #include <math.h> /* Aufbau einer Einheitsmatrix              */
    2  #define ZEILE  10
    3  #define SPALTE 10
    4  main()
    5  {
    6  register int i,k;
    7  float einh_matrix[ZEILE][SPALTE] ;
    8  for (i = 0; i < ZEILE; i++)
    9  for (k = 0; k < SPALTE; k++)
   10  einh_matrix [i][k] = (i == k);  /* Hauptdiagonale = 1, Rest = 0    */
   11  }
No errors detected
```

6.1. Zeiger auf Felder

Eine sehr sinnvolle Varinte des Einsatzes von Zeigern ist Die Ver arbeitung von Feldern sowie deren Feldememente. Zeiger können auf den Feldanfang und auf einzelne Felmente gesetzt werden. Die Adresse der Feldememente wird über den Adressoperator **&** gebildet:
pointer = &array[0]; bzw. ***pointer = &array[i];***

Hinweis:

- Soll auf Feldelemente ein schnellerer Zugriff erzielt werden, sollte dies über Zeiger und nicht über die Feldindizes erfolgen.

Begründung:
Kommt ein Feldindex in einem Ausdruck vor, so wird er vom Compiler grundsätzlich in einen Zeiger auf das erste Feldelement, die Startadresse des Feldes, transformiert. Anschließend wird der Abstand des gewünschten Feldes zum Feldanfang hinzugezählt. Jede Bezugnahme auf Feldelemente wird intern über Zeiger realisiert. Der Zeiger weist direkt auf das benötigte Feldelement, so daß direkt und damit schneller zugegriffen werden kann.

Es besteht somit ein unmittelbarer Zusammenhang zwischen der Indizierung von Feldern und der Zeigerarithmetik. Das Setzen eines Zeigers auf das erste Element
pointer = &array[0]
bzw. an die Anfangsadresse des Feldes
pointer = array;.

Der Adressoperator ist in diesem Fall nicht erforderlich.
Falls ***pointer*** auf irgendein Feldemement von ***array*** zeigt, so bezieht sich der Ausdruck
pointer [1]
auf das nächste Feldemement. Allgemein bedeutet der Ausdruck
pointer + i bzw. ***pointer - i***
die Adresse des folgenden bzw. des vorhergehenden Feldemements.

Schlußfolgerungen:
Zeigt ***pointer*** auf ***array*** , so liefert der Ausdruck ***pointer + i***
die Adresse ***&array[i],***
somit ist der Ausdruck ****(pointer + i)***
nur eine andere Notation für ***array[i].***

Es gilt die einfache Beziehung
pointer = &array[i] ist analog zu ***pointer = array + i.***

Programmbeispiel 30:

```
Line#  Source Line                        Microsoft C Compiler Version 5.10

    1  /*  Einsatz von Zeigern zur Textverarbeitung                    */
    2  #include <string.h>
    3  char txt[]="Das ist ein Beispiel zur Verarbeitung von Texten";
    4  main()
    5  {
    6  char *ptr;
    7  char subtxt[9];
    8  ptr = txt + 0;
    9  strncpy(subtxt,ptr + 12,8);
    10  subtxt[8] = '\0'; /* subtxt = "Beispiel"                       */
    11  ptr = txt + 13;
    12  strncpy(subtxt,ptr + 12,6);  /* subtxt = "Verarb               */
    13  subtxt[6] = '\0';
    14  }
No errors detected
```

6.2. Initialisierung von Feldern

Entsprechend der Initialisierung einer einfachen Variablen kann auch ein Feld einfacher Variablen initialisiert werden. Die Initialisierung besteht in diesem Fall aus einer Liste von Initialsierungen für de einzelnen Feldelemente.

Allgemeine Form der Initialisierung eines Feldes:
speicherklasse typ identifikator [1.feld][2.feld]...[n.feld] =
{{ init_liste_1 },
{ init_liste_2 },
...
{ init_liste_n}};

Die Initialisierung erfolgt zeilenweise in der Reihenfolge der Anordnung der Feldelemente. Bei der Initialisierung von Feldern ist eine Dimensionsangabe nicht zwingend erforderlich. Die Dimensionsangabe wird aus der Anzahl der Werte der Initialisierungsliste berechnet. Vor allem wird von dieser Möglichkeit bei der Initialisierung von Zeichenketten Gebrauch gemacht.

Hinweis:

- Felder der Speicherklasse **auto** können nicht initialisiert werden
- numerische Felder der Speicherklassen **static** oder **extern** werden implizit mit 0 initialisiert
- werden in der Initialisierungsliste nicht genügend Werte angegeben, so wird bei numerischen Feldern der Rest mit 0 aufgefüllt

Hinweis zu Zeichenketten:

- Zeichenkettenfelder können über Zeichenkettenkonstanten initialisiert werden. dabei können die geschweiften Klammern entfallen
- Zeichenkettenfelder der Speicherklassen **static** oder **extern** werden mit der temporären Länge 0 implizit initialisiert
- Die Initialisierung von Zeichenkettenfeldern erfolgt zweckmäßigerweise in einem Zeigerfeld, das mit den Adressen der Zeichenketten initialisiert wird. Es erfolgt nicht die Zuordnung von Zeichenkettenkonstanten, sondern es wird jedem Zeigerelement die Adresse des Beginns der Zeichenkette zugewiesen

Programmbeispiel 31:

```
...
4  /*  Initialisierung von Feldern                                  */
5  int mtr[10][10]; /* extern mit 0 initialisiert                   */
6  char jahresz[4][9] = {"Frühling","Sommer","Herbst","Winter"};
7  static int e_mat[3][3]={{1},{0,1},{0,0,1}}; /*  Hauptdiagonale = 1  */
8  static int feld[5] ={1,2};
9  int e[3][3]={1,2,3,4,5,6,7,8,9,10};  /* mehr als 9 -> Fehler        */
   ***** b31.c(9) : error C2078: too many initializers
10  char txt[]={'T','e','x','t','\0'};
11  char txt1[]={"Text"};
12  static char *ptrjahr[ = {"Frühling",  /* Init über Zeigerfeld       */
13  "Sommer","Herbst","Winter"};
...
1 errors detected
```

7. Strukturen und Vereinigungen

Eine Struktur (**struct**) ist eine Zusammenfassung von Objekten unterschiedlichen Typs. Der Speicherbereich einer Struktur wird vom Compiler aus der Summe der Speichergrößen der einzelnen Komponenten berechnet.
Eine Vereinigung (**union**) ist eine Erweiterung des Strukturtyps. Sie vereinigt auf einem virtuellen Speicherbereich Komponenten verschiedenen Datentyps; sie nutzen gemeinsam einen Platz im Arbeitsspeicher. Der Speicherbereich einer Vereinigung wird vom Compiler der größten Komponente entsprechend festgelegt.

Allgemeine Form einer Struktur- und Vereinigungsvereinbarung:

struct ***strukturname { folge von typvereinbarungen}***
folge von identifier;
union ***vereinigungsname { folge von typvereinbarungen}***
folge von identifier;

Erläuterungen:

- ***strukturname*** sowie ***vereinigungsame*** sind wahlfreie Namen zur Bezeichnung der Struktur oder Vereinigung
- i***dentifier*** ist der Identifikator für dieDarstellung einer Struktur -oder Vereinigungsvariable
- ***identifier*** kann ein einfacher Variablenname, ein Feldname, ein Zeigername sein

Hinweis:

- in Strukturen und Vereinigungen können weitere Strukturen oder Vereinigungen enthalten sein
- für Vereinbarungen gelten die gleichen syntaktischen Regeln wie für Strukturen

7.1. Definitionen von Strukturen

Werden in einer Strukturvereinbarung keine Identifikatoren angegeben, so wird nur der Strukturtyp und damit, falls vorhanden, der Strukturname definiert, über den auf die Struktur Bezug genommen werden kann. Es erfolgt keine Vereinbarung von Strukturvariablen und somit auch keine Speicherplatzvergabe.
Um eine konkrete Instanz dieses Strukturtyps zu definieren, muß auf den Strukturnamen Bezug genommen werden in der Form
struct ***strukturname folge von identifier;***

Werden in einer Strukturvereinbarung Identifikatoren spezifiziert, so bezeichnen diese Identifikatoren im weiteren Strukturvariablen, deren Speicherplatzanforderung durch den entsprechenden Strukturaufbau realisiert wird.

Hinweis:

- Der von einer Strukturinstanz belegte Speicherplatz ergibt sich nicht zwangsläufig aus der Summe des Speicherbedarfs der zugehörigen Komponenten. Es können hinsichtlich der Ausrichtung auf Speichergrenzen der Strukturkomponenten unbelegte Bytes entstehen.

Fallunterscheidungen:

1. Die Strukturdefinition besitzt einen Strukturnamen sowie einen oder mehrere Identifikatoren
   - es wurde eine konkrete Instanz des entsprechenden Strukturtyps definiert und damit ist eine Bezugnahme über die Identifikatoren auf den konkreten Speichplatz möglich
   - weitere Bezugnahmen auf den Strukturtyp über weitere Identifikatoren ist möglich

2. Die Strukturdefinition besitzt einen Strukturnamen und keinen Identifikatoren
   - es wurde nur der Strukturtyp definiert und damit ist eine sofortige Bezugnahme nicht gegeben
   - eine Bezugnahme auf den Strukturtyp über Identifikatoren ist nur über die Form **struct *strukturname folge von identifier;*** möglich

3. Die Strukturdefinition besitzt keinen Strukturnamen, aber einen oder mehrere Idedifikatoren
   - es wurde eine konkrete Instanz des entsprechenden Strukturtyps definiert und damit ist eine Bezugnahme über die Identifikatoren auf den konkreten Speichplatz möglich
   - weitere Bezugnahmen auf den Strukturtyp über weitere Identifikatoren ist nicht möglich, da ein Bezug auf diesen Strukturtyp nicht stattfinden kann

4. Die Strukturdefinition besitzt weder einen Strukturnamen noch einen Identifikator
   - es wurde nur der Strukturtyp definiert und damit ist eine sofortige Bezugnahme nicht gegeben
   - eine Bezugnahme auf den Strukturtyp über Identifikatoren ist wegen des fehlenden Strukturnamens nicht möglich

Programmbeispiel 32:

```
....
1  /* Demonstration der Fallunterscheidungen   */
2  /* Fall 1                                   */
3  struct datum
4  {
5  int tag;
6  char monat[3];
7  int jahr;
8  } heute, datfeld[10];
9  /* Fall 2                                   */
10 struct adresse
11 {
12 char plz[6], ort[30], str[30];
13 };
14 struct adresse wohnung;
15 /* Fall 3                                   */
16 struct
17 {
18 char name[30], vorname[20];
19 struct datum geb;
20 struct adresse arb_stelle;
21 } mitarb;
22 /* Fall 4                                   */
23 struct {int dummy1, dummy2, dummy3;};
...
Global Symbols
Name              Class   Type              Size   Offset
datfeld . . . . . common  struct/array       80     ***
heute . . . . . common  struct/array         8      ***
main. . . . . global  near function          ***    0000
mitarb. . . . . common  struct/array         124    ***
wohnung . . . . . .common  struct/array      66     ***
...
No errors detected
```

7.2. Bezugnahme auf Strukturvariable und deren Komponenten

Auf die einzelnen Komponenten einer Struktur kann entweder
° über den Variablennamen der Struktur (Identifikator) oder
° über einen Zeiger auf die Struktur
bezug genommen werden.

Bei der Verwendung des Variablennamens der Struktur wird der Komponentenname mittels eines Punktes als Kettungsoperator für Komponenten angehangen.

Allgemeine Form:
strukturvariable . strukturkomponente

Bei der Verwendung eines Strukturzeigers wird über die Pointerzuweisung -> indirekt auf die Komponenten der Struktur zugegriffen.

Hinweise zur Arbeit mit Strukturzeigern:

° Vereinbarung eines Zeigers auf die Struktur
struct *strukturname *strukturzeiger;*

° Zuweisung der Adresse der gewünschten Struktur über den Adressoperator **&**
pointer = &strukturvariable;

° Bezugnahme auf die Strukturkomponenten über den Indirektoperator ***
(*pointer) . strukturkomponente (a) Die Klammer ist wegen der höheren Priorität des Operators **.** gegenüber dem Operator ***** notwendig. Da diese Schreibweise recht aufwendig ist, gilt anstelle (a)
pointer -> strukturkomponente

° Da das Inkrementieren eines Zeigers ***++pointer*** grundsätzlich in Speichereinheiten des Objekttyps erfolgt, bedeutet das bei Strukturzeigern ein Inkrementieren um die Gesamtlänge der Struktur. Die Anwendung von Inkrement- und Dekrementoperatoren auf Strukturzeiger ist nur sinnvoll, wenn diese auf Felder von Strukturen zeigen.

° Da der Vorrang des Operators **->** höher ist als der von **++** und -- , ist bei gemeinsamer Verwendung dieser Operatoren auf genaue Notation zu achten. Im Bedarsfall sind Klammern zu setzen
pointer -> strukturkomponente++
++pointer -> strukturkomponente
++(pointer -> strukturkomponente)
(pointer++= -> strukturkomponente)
(++pointer) -> strukturkomponente

° der Vorrang des Operators liegt unter dem von **->**. Somit bezieht sich bei
pointer -> strukturkomponente
der Operator * nicht auf pointer, sondern auf den Inhalt, worauf strukturkomponente zeigt

Programmbeispiel 33:

```
3   /* Einfache Strukturverarbeitung ohne und mit Zeigern       */
4   struct datum
5   {
6   int tag;
7   char monat[4];
8   int jahr;
9   };
10   struct person
11   {
12   long persnr;
13   char name[10], vorn[10];
15   struct datum geb_datum;
16   };
17   struct person angest[10],freimit, *ptr;
19   /*  Fuellen der Stuktur ohne Zeigenbenutzung                */
20   angest[2].persnr = 4711;
21   strcpy(angest[2].name, "M_ü_ller");
22   strcpy(angest[2].vorn, "Paul");
23   angest[2].geb_datum.tag = 19;
24   strcpy(angest[2].geb_datum.monat, "Jun");
25   angest[2].geb_datum.jahr = 1960;
26   angest[1] = angest[2];          /* Strukturzuweisung          */
27   /* angest[1] = 4711 M_ü_ller Paul 19 Jun 1960                 */
28   ptr =  &angest[3];             /* Pointerverarbeitung         */
29   *ptr = angest[2];
30   ++ptr -> persnr;               /* 4712                        */
31   ++ptr -> geb_datum.tag;
32   /* angest[3] = 4712 M_ü_ller Paul 20 Jun 1960                 */
...
No errors detected
```

7.3. Initialisieren von Strukturen

Strukturen werden in der gleichen Weise wie Felder initialisiert. Bei der Initialisierung werden die verschiedenen Initialisierungswerte in einer Liste niedergeschrieben und in geschweifte Klammern eingeschlossen.

Allgemeine Form der Initialisierung einer Struktur
struct *strukturname identifier = { init-liste };*

Hinweis:

- Strukturen der Speicherklasse **auto** können nicht initialisiert werden
- bei der Initialisierung eines Strukturfeldes sollte jedes Feldemement in geschweifte Klammern eingeschlossen werden.

Programmbeispiel 34:

```
                                                        02-24-91

Line#  Source Line                      Microsoft C Compiler Version 5.10

1  /* Strukturinitialisierungen _ü_ber verschiedene Wege          */
2   struct datum
3   {
4    int tag; char mon[4]; int jahr;
5   } heute = {{24},"Feb",{1991}};
6  main()
7  {
8 static struct datum morgen = {25,"Feb",1991};
...
No errors detected
```

7.4. Vereinigungen

Eine Vereinigung (Verbund) ist eine Variable, die zu verschiedenen Zeiten in derselben Datenadresse Daten verschiedener Datentypen enthalten kann.

Hinweis:

- wie bei Strukturen muß auch hier erst der **union**-Typ deklariert werden, um den Compiler die Anzahl und Typen der Komponenten mitzuteilen
- in einer Vereinigung stellt der Compiler soviel Speicherplatz zur Verfügung, wie der größte Datentyp in der Vereinigung benötigt
- die Elemente einer Vereinigung heißen ebenfalls Komponenten und werden genau wie die Komponenten von Strukturen mit dem Kettungsoperator angesprochen

Programmbeispiel 35:

```
Line#  Source Line                    Microsoft C Compiler Version 5.10

     1  /* Zerlegen eines Int-Wertes in sein High- und Lowbyte      */
     2  union int_wert
     3  {
     4  unsigned int wort;
     5  struct
     6  {
     7  char high_byte;
     8  char low_byte;
     9  } byte;
    10  }
    11  main()
    12  {
    13  static union int_wert wert;
    14  wert.wort = 0xFF37; /* 65335                                 */
    15  printf("\n High: %2x Low: %2x",
    16  wert.byte.high_byte, wert.byte.low_byte);
    17  } /* High: 37  Low: ffff                                     */

main  Local Symbols
Name                 Class   Type           Size   Offset  Register
wert. . . . . . . . . . static  struct/array     2     0000
No errors detected
```

8. Funktionen

Ein Programm besteht aus einer Menge von Funktionsdefinitionen, die in beliebiger Folge angeordnet sein können. Die Gesamtheit der Funktionen eines Programms kann über mehrere Quelltexte verteilt sein, solange eine einzelne Funktion vollständig in einem File enthalten ist. Jedes dieser Files kann separat übersetzt und mit anderen getrennt übersetzten Moduln komplett verbunden werden.
Die Ausführung eines Programms beginnt grundsätzlich bei der **main**- Funktion.

8.1. Überblick und Klassifizierung der Standardfunktionen

Zunächst sei darauf hingewiesen, daß die Sprache C keinerlei Ein- und Ausgabeanweisungen enthält. Aus diesem Grunde wurde eine umfangreiche Standardbibliothek definiert und bereitgestellt, die selbstverständlich neben den Ein- und Ausgabefunktionen weitere, sehr nützliche Funktionen enthält.

Die folgende Aufstellung gibt einen Überblick über die Standardfunktionen.
Sie sind thematisch in verschiedenen Gruppen aufgeteilt:

° Prozeßverwaltungsroutinen
° BIOS- und DOS-Interfaceroutinen
° Dateiverwaltungsroutinen
° Ein- und Ausgabefunktionen
° Fehlerbehandlungs- und Debuggingroutinen
° Prüfungsroutinen
° Mathematische Funktionen
° Speicher- und Stringverarbeitung
° Speicherverwaltungsfunktionen
° Such- und Sortierfunktionen
° Umwandlungsroutinen
° Verwaltung von Systemvariablen
° Textfensterfunktionen

8.2 Definition von Funktionen

Eine Funktion besteht aus einem Funktionskopf und einem Funktionskörper.
Der Funktionskopf legt
° den Namen der Funktion,
° Die Reihenfolge und Typ der formalen Parameter und
° den Typ des Funktionsergebnisses
fest.

Der Funktionskörper ist ein Block mit Vereinbarungs- und Anweisungsteil.

Eine Funktion hat folgenden prinzipiellen Aufbau:

```
speicherklasse typ funktionsname(liste der formalen parameter)
deklaration der formalen parameter
{
 deklarationen
  anweisungen
}
```

Der ANSI-Standard erlaubt zusätzlich die moderne Form:

speicherklasse typ funktionsname (liste der deklaration der formalen parameter)
{
deklarationen
anweisungen
}

Erläuterungen:

- wird keine ***speicherklasse*** spezifiziert, wird **extern** angenommen
- ***typ*** legt den Datentyp des Funktionswertes fest. Möglich sind
  - alle Grundtypen (c**har, int, float, double**)
  - Strukturen und Unions
  - Zeiger auf beliebige Objekte
- der Standardtyp ist **int**
- wird kein Funktionswert geliefert, so wird **void** angegeben
- ***liste der formalen parameter*** kann leer sein, es muß aber in jedem Fall das Klammernpaar () notiert werden
- ***deklarationen*** legen den Datentyp der in der Funktion benutzten Variablen fest (siehe Punkt 2.)
- ***anweisungen*** legen die Steuerung des Programmablaufs mit den bekannten sprachlichen Mitteln fest

Hinweis:

- die Funktion wird verlassen, d.h., die Steuerung wird an die rufende Funktion zurückgegeben, wenn die rechte geschweifte Klammer erreicht wird
- die Funktion wird vorzeitig verlassen, wenn in der **return**- Anweisung ein Ausdruck spezifiziert wird, so dass der Wert des Ausdrucks als Funktionswert an die rufende Funktion übermittelt werden kann

8.3. Aufruf von Funktionen

Ein Funktionsaufruf ist ein Ausdruck, dessen Wert sich aus dem Funktionsergebnis der gerufenen Funktion ergibt.

Allgemeine Form des Funktionsaufrufs:
funktionsname (argumentliste);

Hinweis:

- ist die Argumentliste leer, muß ( ) trotzdem angegeben werden
- ein vorher noch nicht vereinbarter Funktionsname wird implizit als Name einer Funktion deklariert und der Funktionswert der Funktion ist vom Typ **int**
- einfache Variable werden nach der Methode call by value übergeben, d.h., der Wert des Arguments wird berechnet und dem entsprechenden Parameter zugewiesen. Die aufge-rufene Funktion arbeitet mit einer temporären lokalen Kopie eines jeden Arguments
- Felder, Strukturen, Vereinigungen und Funktionen werden nach der Methode call by reference übergeben, d.h., es werden Anfangsadressen übergeben
- man sollte darauf achten, dass der Typ des Arguments mit dem Typ des entsprechenden formalen Parameters übereinstimmt

8.4. Übergabe eines Funktionswertes

Die Abarbeitung einer Funktion kann durch eine **return**-Anweisung beendet werden.

Allgemeine Form:
return *ausdruck*;

Der Wert von ***ausdruck*** wird berechnet und vor der Rückgabe an die rufende Funktion eventuell in den im Funktionskopf festgelegten Typ konvertiert.
Funktionen, die 'keinen' Funktionswert zurückgeben sollen, können ihren Ablauf beenden durch

- die Anweisung **return;**
- durch Erreichen der rechten Klammer **}**

8.5. Rekursiver Funktionsaufruf

Der Aufruf einer Funktion ist rekursiv, wenn sich die Funktion entweder direkt oder indirekt selbst aufruft.

Hinweis:

- rekursive Lösungen sind im Laufzeitverhalten nicht schneller und sparen keinen Speicherplatz, da im Stack eine Reihe von Zwischenlösungen und Adressen abgelegt werden müssen
- rekursive Lösungen sind schneller und leichter zu schreiben

Programmbeispiel 36:

```
Line#  Source Line                          Microsoft C Compiler Version 5.10

   1  /* Rekursiver unktionsaufruf zur Berechnung der Fakultaet einer Zahl */
   2  #include <stdio.h>
   3  float faku(par)
   4  int par;
   5  /* Fakultätsberechnung in rekursiver Form                          */
   6  {
   7  if (par <= 1) return(1.0);
   8  else
   9  eturn(par * faku(par - 1));
  10  }
  11
  12  main()
  13  {
  14  int zahl;
  15  printf("FAkULTÄTSBERECHNUNG: Von welcher Zahl: ??_");
  16  scanf("%d",&zahl);
  17  printf(" Fakult_ä_t = %f", faku(zahl));
  ...
No errors detected
```

8.6. Zeiger auf Funktionen

Zeiger auf Funktionen können u.a. benutzt werden, um Funktionen als Funktionsargumente zu übergeben. Wenn in einem Ausdruck ein Funktionsname vorkommt, der selbst keinen Funktionsaufruf darstellt, so wird dieser Funktionsname als Zeigerwert dieser Funktion interpretiert.

In der Funktion selbst muß der Funktionszeiger deklariert werden:
typ (*funktionsname)( );

Hinweis:

° die Klammern um den Funktionsnamen sind wegen der niedrigeren Prioritiät des Sternchens anzugeben

Programmbeispiel 37:

```
                                                        PAGE 1
                                                      03-11-91
                                                      15:54:58

Line#  Source Line                  Microsoft C Compiler Version 5.10

    1   /* Demonstration des Funktionsaufrufs ohne und mit Zeiger*/
    2   float quadrat(par_q)
    3   float par_q;
    4   {
    5   return(par_q * par_q);
    6   }

quadrat  Local Symbols
Name                Class   Type            Size   Offset  Register
par_q . . . . . . . . . . param                 0004

    7   float kubik(par_k)
    8   float par_k;
    9   {
    10  return(par_k * par_k * par_k);
    11  }

kubik  Local Symbols

Name                Class   Type            Size   Offset  Register
par_k . . . . . . . . . . param                 0004

    12
    13  /* Funktionsdefinition ohne Zeigerargumente                */
    14  ausgabe(par_a,flag)
    15  float par_a;
    16  char flag;
    17  {
    18  if (flag == 'q')
    19  printf("\n Ergebnis = %f",quadrat(par_a));
    20  else
    21  printf("\n Ergebnis = %f",kubik(par_a));
    22  }

ausgabe  Local Symbols

Name                Class   Type            Size   Offset  Register
par_a . . . . . . . . . . param                 0004
flag. . . . . . . . . . . param                 000c

    23  /* Funktionsdefinition mit Zeigerargumente                 */
    24  zeigausg(par_a,ptrfunkt)
    25  float par_a,(*ptrfunkt)();
    26  {
    27  printf("\n Ergebnis = %f",ptrfunkt(par_a));
    28  }
```

```
zeigausg  Local Symbols

Name                 Class   Type            Size   Offset  Register
par_a . . . . . . . . . param                 0004
ptrfunkt. . . . . . . . param                 000c

    29
    30  main()
    31  {
    32  float quadrat(),kubik();
    33  ausgabe(4.0, 'q'); /* konventioneller  Aufruf                  */
    34  ausgabe(4.0, 'k');
    35  zeigausg(4.0, quadrat); /* Aufruf über Zeiger                  */
    36  zeigausg(4.0, kubik);
    37  }

Global Symbols

Name                 Class   Type            Size   Offset

_fac. . . . . . . . . . .  extern  double          8     ***
ausgabe . . . . . . . .global  near function     ***    003c
kubik . . . . . . . . . . global  near function   ***    001c
main. . . . . . . . . . . global  near function   ***    00fa
printf. . . . . . . . . . extern  near function   ***    ***
quadrat . . . . . . . .global  near function     ***    0000
zeigausg. . . . . . . . global  near function   ***    00b6
No errors detected
```

9. Der C - Präprozessor

Die Sprache C bietet bestimmte Spracherweiterungen mittels eines einfachen Präprozessors, der Makrosubstitution, bedingte Übersetzungen und das Einfügen benannter Files durchführt.

Hinweis:

- ° Sprachanweisungen für den Makroprozessor beginnen mit dem Zeichen **#** in Spalte 1
- ° Fortsetzungszeilen werden mit \ als letztes Zeichen der vorangegangenen Zeile definiert
- ° vor der eigentlichen Compilierung wird der gesamte Quelltext nach **#** -Anweisungen durchsucht

9.1. File - Einfügungen

Um zu gewährleisten, daß bei großen Projekten, die aus einer Vielzahl einzelner Quellfiles bestehen, sich alle Quellfiles auf gleiche symbolische Konstanten, Makros und global benötigte Vereinbarungen beziehen können, werden diese Sprachelemente in den sog. Headerfiles zusammengefaßt. In den einzelnen Quelltextfiles muss dann die entsprechende Anweisung zum Einfügen dieses Headerfiles enthalten sein.

Allgemeine Form:
#include <*filename*>

Es wird ein File aus dem Bibliothek-Fileverzeichnis eingefügt.
#include *"filename"*

Es wird ein File aus dem aktuellen Fileverzeichnis des Nutzers eingefügt.

Hinweis:

- ° der eingefügte C - Text kann beliebig sein, insbesondere aus Strukturdefinitionen, Makrodefinitionen, Definitionen symbolischer Konstanten bestehen

9.2. Makrosubstitution, Definitionen symbolischer Konstanten

Eine Definition der Form
#define *symbolische_konstante zeichenfolge*
zieht eine Makrosubstition der einfachsten Art - die Definition einer symbolischen Konstanten nach sich.

Im Quelltext wird ***symbolische_konstante*** durch ***zeichenfolge*** ersetzt. Da die Implementation von **#define** ein Makropräpass ist und kein Teil des eigentlichen Compilers, existieren sehr wenig syntaktische Einschränkungen an ***zeichenfolge***.

Hinweis:

- ° eine symbolische Konstante kann neu definiert werden und kann vorangegangene Defi- nitionen verwenden
- ° für symbolische Konstanten gelten die gleichen Namenskonventionen wie für Identifikatoren

Es ist ebenfalls möglich, Makros mit Argumenten zu definieren. Der Ersetzungstext ist von der Art und Weise des Makroaufrufs abhänig.

Allgemeine Form einer Makrodefinition:
#define *makro_name(parameter_liste) zeichenfolge*

Allgemeine Form des Makrorufs:
makro_name (argument_liste);

Jedes Auftreten des Makrorufs führt zu einer Substition durch die angegebene Zeichenfolge.

Hinweis:

- die öffnende Klammer in der Makrodefinition muß unmittelbar an den Makronamen anschließen
- die Anzahl der Parameter muß mit der Anzahl der Argumente übereinstimmen

9.3. Streichen von Makro-Definitionen

Die Gültigkeit einer Makrodefinition erstreckt sich bis zum Verlassen der entprechenden Funktion oder bis zum Streichen der Makrodefinition.

Allgemeine Form zum Streichen einer Makrodefinition:
#undef *makro_name* bzw.
#undef *symbolische_konstante*

Mittels **#undef** werden die angegebenen Bezeichner aus dem Verzeichnis des Präprozessors gestrichen.

9.4. Bedingte Compilierung

Zur Unterstützung der Portabilität von C-Programmen auf Quelltextniveau können in Abhänigkeit von bestimmten Bedingungen die folgenden Quelltextzeilen des Eingabefiles durch den Präprozessor normal verarbeitet oder übergangen werden. Es können damit verschiedene Teile des Quellcodes zur Übersetzung gelangen oder nicht.

Ein bedingter Anweisungsblock wird durch eine der drei Präprozessoranweisungen
#if *konstanter_ausdruck*
#ifdef *bezeichner*
#ifndef *bezeichner*

eingeleitet und beendet durch
#endif

Die Präprozessoranweisung
#else
erlaubt eine Alternative.

Erläuterungen:
- **#if** prüft den Wert des Konstantenausdrucks auf ungleich 0. Ist der Wert ungleich 0, werden die folgenden Anweisungen bis zum **#else** oder **#endif** generiert.
- **#ifdef** prüft, ob der Bezeichner für den Präprozessor gegenwärtig definiert ist, d.h., ob er in einer **#define**-Steuerzeile angegeben wurde.
- **#ifndef** prüft, ob der Bezeichner für den Präprozessor gegenwärtig undefiniert ist.

Programmbeispiel 38:

```
                                                        PAGE 1
                                                        03-27-91
                                                        00:23:55
Line#  Source Line                    Microsoft C Compiler Version 5.10

  1   /*  Definieren einer symbolischen Konstanten, eines Makros sowie */
  2   /*  einer symbolischen Konstanten f_ü_r eine Ausgabefunktion      */
  3   #define Maximum \
  4   "\nErmitteln des Maximums zweier Zahlen"
  5   #define  Striche \
  6   printf("\n-------------------------------------\n");
  7   #define MAX(x,y) ((x) > (y) ? (x) : (y))
  8   int tab[]={1,34,54,534,454,323,544,232,343,46};
  9   main()
 10   {
 11    printf( Maximum );
 12    #ifdef Striche
 13    Striche;
 14    #else /* Die folgende Anweisung wird in diesem Beispiel          */
 15    /* in den Objektmodul nicht mit aufgenommen                      */
 16    printf("\n-------------------------------------\n");
 17    #endif
 18    printf("%d", MAX(tab[3],tab[5]));
 19    }


Global Symbols

Name                 Class   Type              Size    Offset

main. . . . . . . . . global  near function      ***    0000
printf. . . . . . . . extern  near function      ***    ***
tab . . . . . . . . . . global  struct/array      20    0050
No errors detected
```

10. Dateiverwaltung

Über die Sprache C herrschen hinsichtlich ihrer Anwendbarkeit häufig weitreichende Meinungsunterschiede. Hauptsächlich wird C in Verbindung mit technischen und wissenschaftlichen Problemstellungen erwähnt. Da die Stärke vor allem im kommerziellen Bereich zu suchen ist zeigt die Tatsache, dass viele Softwaresysteme mit universellen Anwendungsgebieten in programmiert wurden.

In der Sprache C steht eine Vielzahl von Ein- und Ausgabefunktionen zur Verfügung, die im Einzelnen natürlich hier nicht besprochen werden können. Es sei auf die umfangreiche Literatur bzw. auf die im folgenden Kapitel gemachte Zusammenstellung der wichtigsten Ein- und Ausgabefunktionen verwiesen.

10.1. Eigenschaften der Dateibehandlung

Eine Besonderheit der Sprache C besteht in der Tatsache, dass jedes externe Gerät als Datei betrachtet wird. In der Standardbibliothek **stdio.h** sind bereits verschiedene Kanäle als Standardkanäle definiert. Das sind im Einzelnen

° die Standardeingabe als **stdin**
° die Standardausgabe als **stdout**
° die Standardfehler als **stderr**
° die serielle Schnittstelle als **stdaux**
° den Drucker als **stdprn**

Diese Standardkanäle wurden als Komponenten eines Strukturbereichs auf dem Strukturtyp FILE, welcher in der **stdio.h** definiert ist,

#define stdin (&_streams[0])
#define stdout (&_streams[1])
#define stderr (&_streams[2])
#define stdaux (&_streams[3])
#define stdprn (&_streams[4])

mit 0 beginnend durchnumeriert. Man spricht hierbei von einem sog. Handle. Um einen Handle einer Datei zu erhalten, muss dieser File eröffnet werden. Die in der **stdio.h** definierten Standardkanäle werden selbstverständlich durch den Nutzer nicht explizit eröffnet.
Auf die somit erhaltene Nummer, den Handle, wird für alle weiter durchzuführenden Ein- und Ausgabefunktionen Bezug genommen.
In C sind die Dateiverwaltungsfunktionen in zwei Kategorien eingeteilt:
° die Basis oder Low-Level-Routinen und
° die gehobenen oder High-Level-Routinen

Die Unterschiede zwischen den systemnah arbeitenden Low-Level-Routinen und den allgemeineren High-Level-Routinen sind vor allem in ihrer Anwendbarkeit zu suchen. Bei der Arbeit mit High-Level-Routinen muß der Anwender sich aber nicht mit Besonderheiten wie zum Beispiel der Steuerzeichenumsetzung befassen. Auf diese Weise arbeitet der Anwender unabhängig vom verwendeten Betriebssystem und produziert damit automatisch portable Programme.

Folgende Tabellen sollen eine Zusammenfassung der wichtigsten Ein- und Ausgabefunktionen angeben. Die Angaben zur Parametrierung kann hier im Einzelnen nicht besprochen werden.

Low-Level-Funktionen zur Dateiverwaltung

Funktion	Erklärung
access(char **file,* int *modus*)	Prüft, ob ***file*** existiert,gelesen, geschrieben werden soll Funktionswert: Prüfung erfolgreich: 0 ansonsten :-1
close(int *handle)*	Schließt geöffneten ***handle*** Funktionswert: erfolgreich 0 ansonsten :-1
creat(char **file,* int *access)*	Legt neuen ***file*** an oder eröffnet besstehenden. Funktionswert: erfolgreich: ***handle*** ansonsten :-1
dup(int *handle)*	Dupliziert einen ***handle*** Funktionswert: erfolgreich: nächsten freien ***handle*** ansonsten :-1
eof(int *handle)*	Prüft, ob Dateiende erreicht ist Funktionswert:Dateiende erreicht: 1 ansonsten: 0 Fehler : -1
filelength(int *handle)*	Dateilänge ermitteln Funktionswert: erfolgreich: **long**-Bytes ansonsten : -1
lseek(int *handle,* long *offset,* int *fromwhere)*	Dateizeige auf neue Position setzen Funktionswert: erfolgreich: Position von Dateibeginn in **long**-Bytes ansonsten : -1L
tell(int *handle)*	Dateizeigeposition ermitteln Funktionswert: erfolgreich: Position von Dateibeginn in **long**-Bytes ansonsten : -1L
open(char **file,* int *access* **int *access*** ***[,*int *permiss])***	Öffnet ***file*** in gewünschter ***access*** Funktionswert: erfolgreich:handle ansonsten :-1
read(int *handle,* void **buf,* *, unsigned nbyte)*	Lies aus File ***handle*** die gewünschte Anzahl von ***nbyte*** und legt sie im Puffer ***buff*** ab Funktionswert: erfolgreich: Anzahl gelesene.Bytes ansonsten :-1
write(int *handle,* void **buf,* unsigned *nbyte)*	Schreibt in File ***handle*** die gewünschte Anzahl von ***nbyte*** Bytes aus Puffer ***buff*** Funktionswert: erfolgreich: Anzahl geschr.Byte ansonsten :-1
unlink(char **file)*	Löscht einen File Funktionswert: erfolgreich: 0 ansonsten: -1

High-Level-Funktionen zu Dateiverwaltung

Funktion	Erklärung
fclose(FILE **stream)	Schließt geöffnete Datei ***stream***
	Funktionswert. erfolgreich: 0 ansonsten :-1
fcloseall(void)	Schließt alle geöffneten Dateien
	Funktionswert: erfolgreich: Anzahl Dateien ansonsten : -1
feof(FILE **stream)	Prüft, ob Dateiende erreicht ist
	Funktionswert: erfolgreich: End of File ansonsten:0; Fehler :-1
fgetc(FILE **stream)	Holt ein ***zeichen*** von ****stream***
	Funktionswert: erfolgreich:gelesenes ***zeichen*** ansonsten : end of file
fgets(char **string, ***laenge,* FILE **stream)**	Liest Zeichenkette in ***string*** mit max. ***laenge*** - 1 Zeichen **int** Funkionswert: erfolgreich: Zeiger auf ***tring*** ansonsten : **NULL**
fopen(char **dateiname, ***modus)***	Öffnet oder erstellt Datei im gewünschten ***modus*** **int** Funktionswert: erfolgreich:Zeiger auf **FILE** ansonsten : **NULL**
fprintf(FILE **stream, **char **formatstring,** **void *argumente)***	formatierte Ausgabe auf geöffnete Datei Funktionswert: erfolgreich: Anzahl ausgegebene Zeichen ansonsten : end of file
fputc(int *zeichen, **FILE **stream)**	Ausgabe ein ***zeichen*** auf ****stream*** Funktionswert: erfolgreich: ausgegebene Zeichen ansonsten : end of file
fputs(char **string, **FILE **stream)**	Ausgabe ***string*** auf geöffnete Datei Funktionswert: erfolgreich: letztes ausgegeben Zeichen ansonsten: end of file
fread(void **buffer, **int *anzahl bytes,*** **int *anzahl sätze,* FILE **stream)***	Liest ***anzahl sätze*** mit Länge ***anzahl bytes*** in ****buffer*** ein Funktionswert: erfolgreich: Anzahl gelesene Einheiten ansonsten: Wert < 0
fscanf(FILE **stream, **void *argumente)***	Formatierte Eingabe in geöffnete Datei **char *formatstring,*** Funktionswert: erfolgreich: Anzahl geles. Argumente ansonsten : end of file
fseek(FILE **stream, **long *offset,* int *whence)***	Dateizeige auf neue Position setzen Funktionswert: erfolgreich: Pos. von Dat.beginn in **long**-Bytes ansonsten : Wert < 0
ftell(FILE **stream)*	Dateizeigeposition ermitteln Funktionswert: erfolgreich: Pos. von Dat.beginn in **long**-Bytes ansonsten : -1L
fwrite(void **buffer, **int *anzahl bytes,*** **int *anzahl sätze*,FILE** **sream)*	Schreibt ***anzahl sätze*** mit Länge ***anzahl bytes*** in ****buffer*** Funktionswert: erfolgreich: Anzahl geschr. Einheiten ansonsten: Wert < 0
ungetc(char *zeichen, **FILE **stream)**	Schreibt ***zeichen*** zurück in den Puffer von ***stream*** Funktionswert: zurückgeschriebe Zeichen
rewind(FILE **stream)	Filepointer auf Dateianfang von ***stream*** Funktionswert: Zeiger auf Dateibeginn

10.2. Ein - und Ausgaberoutinen der Standardbibliotek

Es sollen einige wichtige Ein - und Ausgabefunktionen der Standardbibliothek angegeben werden. Sie lassen sich in folgende Gruppen einteilen:

- Funktionen bzw. Makros zur zeichenweisen Ein - und Ausgabe ohne Formatierung
- Funktionen zur zeilenweisen Ein - und Ausgabe ohne Formatierung
- Funktionen zur formatgesteuerten Ein - und Ausgabe

10.2.1. Zeichenweise Ein - und Ausgabe

Diese Form der Ein - und Ausgabe wird über die Makros

int *zeichen* = getchar( ); bzw. **putchar(int *zeichen);**
int *zeichen* = getc(stdin); bzw. **putc(int *zeichen,*stdout);**

realisiert. Gelesen wird von der Standardeingabe, geschrieben in die Standardausgabe. Sollen die Zeichentransporte nicht über Standarddateien realisiert werden, müssen die adäquaten Funktionen

fgetc(FILE **stream)* bzw. **fputc(int *zeichen,*FILE **stream)***

benutzt werden.
Bei jedem Aufruf wird genau ein Zeichen übergeben und bei Eingabeende **EOF**.

Hinweis:

- **EOF** ist in der st**dio.h** als **-1** definiert und wird über **CTRL Z** erwirkt
- Bei der Eingabe wird der Code der Return-Taste (hex.0D) in ein neue Zeile Zeichen (hex.0A) umgewandelt
- Bei der Ausgabe wird das neue Zeile Zeichen in Return und Neue Zeile (hex.0- und 0A) umgewandelt
- Zwischen den Makros gibt es folgende Zusammenhänge(definiert in der **stdio.h**):

  #define getchar( ) <-> **getc(stdin)** und
  #define putchar(c) <-> **putc((c),stdout).**

- zwischen den Makros und den entsprechenden Funktionen wiederum bestehen folgende Zusammenhänge (definiert in der **stdio.h**):

  #define getc(*f*) <-> ...f**getc(f)**
  #define putc(*c,f*) <-> ...**fputc((c),f)**.

Abgesehen davon, daß **getc** bzw. **getchar** Makros und **fgetc** eine echte Funktion ist, sind beide vollkommen identisch. Die gleiche Aussage gilt selbstverständlich auch für die Ausgabe.

Programmbeispiel 39:

```
                                                        PAGE   1
                                                        03-27-91
                                                        21:47:46

Line#  Source Line                        Microsoft C Compiler Version 5.10

     1  /* Kopieren Eingabe auf Ausgabe mitUmwandlung in Grossbuchst. */
     2  #include <stdio.h>
     3  #include <ctype.h> /* Funktionen islower und toupper definiert      */
     4  main()
     5  {
     6  int zeich;
     7  while ((zeich = getchar()) != EOF)
     8  /* CTRL-Z oder F6 erwirkt EOF (End Of File)                        */
     9  putchar( islower( zeich) ? toupper(zeich) : zeich);
    10  }


Global Symbols

Name                 Class   Type            Size   Offset
_ctype. . . . . .  extern   struct/array      ***    ***
_filbuf . . . . . .extern   near function     ***    ***
_flsbuf . . . . . .extern   near function     ***    ***
_iob. . . . . .    extern   struct/array      ***    ***
main. . . . . .   .global   near function      ***    0000
No errors detected
```

10.2.2. Zeilenweise Ein - und Ausgabe

Diese Form der Ein - und Ausgabe wird über die Funktionen
gets(char **string);* und **puts(char **string);***
realisiert.
Gelesen wird von der Standardeingabe, geschrieben in die Standardausgabe. Sollen die Zeichentransporte nicht über Standarddateien realisiert werden, müssen die adäquaten Funktionen
fgets(charstring,* int *laenge*,FILE **stream);** und **fputs(char **string*,FILE **stream);**
genutzt werden.

gets(...) bzw. **fgets(...)** dient zum Einlesen einer Zeichenkette. Sie wird im Argument ***string*** abgelegt. Nach Eingabe der Enter - Taste, nach Erkennen eines neue Zeile - Zeichens (\n) oder nach ***laenge - 1*** gelesenen Zeichen wird die Eingabe beendet und die Zeichenkette mit der Endekennung \0 versehen. Das neue Zeile - Zeichen wird nicht ersetzt.

puts(...) schreibt den nullterminierten ***string*** in die Standardausgabedatei und beendet die Ausgabe mit einem neue Zeile Zeichen (hex.0A).

fputs(...) schreibt den nullterminierten ***string*** in die mit ***stream*** festgelegte Datei. Ob ein neue Zeile Zeichen ergänzt wird, hängt vom verwendeten Dateityp ab (siehe Abschnitt 10.3.2)

Hinweis:

- Bei dieser Form der Datentransporte handelt es sich nur um Abbildungen von Daten, dh. es können beim Datentransport keine Typumwandlungen erzwungen werden.
- Die Funktion **puts( )** arbeitet mit Standardattributen; d.h. farbige Ausgaben sind nicht möglich. Für farbige Ausgaben und andere Spezialitäten des PC ist die Funktion **cputs( )** vorgesehen.

Programmbeispiel 40:

```
                                                        PAGE 1
                                                      03-27-91
                                                      21:48:05
Line#  Source Line                    Microsoft C Compiler Version 5.10

1   /*    Kopieren Eingabezeile von Tastatut  auf Ausgabebildschirm  */
2   #include <stdio.h>
3   #define BUFFER 128    /* Maximale Grösse einer Tastatureingabe   */
4   main()
5   {
6   char line[BUFFER];
7   while( gets (line) )
8   puts(line);
9   }

main  Local Symbols

Name              Class   Type          Size   Offset  Register
line. . . . . . . . . . . auto                      -0080

Global Symbols

Name              Class   Type              Size   Offset
gets. . . . . . . . . . . extern  near function     ***    ***
main. . . . . . . . . . .global  near function      ***    0000
puts. . . . . . . . . . . extern  near function     ***    ***
No errors detected
```

10.2.3. Formatgesteuerte Ein - und Ausgabe

Werden beim Datentransport Typumwandlungen verlangt, bietet sich die formatierte Ein - und Ausgabe von Daten an. Für die Bearbeitung von Standarddateien stehen zwei wichtige Funktionen zur Verfügung:

scanf (char **formatstring,*void *argumente);* und
printf(char **formatstring,*void *argumente);*

Der Formatstring gibt den Aufbau des externen Datensatzes an, wobei spezifiziert wird, an welche Stelle sich welche Argumente in welchem Datenformat befinden sollen. Die Datenformate der Argumente werden durch Formatelemente beschrieben, die mit Textzeichen beliebig gemischt werden können.
Bei der Eingabe von der Standardeingabedatei **stdin** geben die Formatelemente an, wie die Eingabezeichenkette zu interpretieren ist.
Bei der Ausgabe in die Standardausgabedatei **stdout** geben die Formatelemente die Stelle an, an der Argumente in welchem Format eingefügt werden sollen.
Da die Standardfunktion **printf** sicher eine der meist benötigten Funktionen zu Darstellung von Daten in der Standardausgabe darstellt, soll hier diese Funktion vollständig beschrieben werden.

Der Formatstring aus o.g. Definition hat folgenden allgemeinen Aufbau:

% [flags] [breite] [.genauigkeit] [modifizierer] typ

Die sog. metalinguistischen Konnektoren ***[ ]*** stehen hier für wahlweise zu machende Angaben. Die Formatangaben beginnen grundsätzlich mit einem Prozentzeichen **%.**

Die Pflichtangabe ***typ***, welche den Datentyp des Arguments festlegt und die Art des Ausgabetyps beschreibt, muß durch eine der folgenden Angaben beschrieben werden:

typ	Ausgabeformat	Erklärung
d,i	**signed int**	(dezimal)
o	**unsigned int**	(oktal)
u	**unsigned int**	(dezimal)
x	**unsigned int**	(hex), Ziffern 0..9,a..f
X	**unsigned int**	(hex), Ziffern 0..9,A..F
f	**float/double**	Darstellung: [-]dddd.dddd
e	**float/double**	Darstellung: [-]d.ddd e[+/-]ddd
E	**float/double**	Darstellung: [-]d.ddd E[+/-]ddd erfolgreich: Position von Dateibeginn
g,G	**float/double**	Darstellung wie e bzw. E oder f, abhänig von der Genauigkeit
c	**char**	einzelnes ASCII-Zeichen
s	**char** oder	Zeichenkette bis Nullzeichen bis die Ausgabebreite erreicht ist
%		Ausgabe des Prozentzeichens
p		Zeiger wird in hexdez. Schreibw **near** YYYY Offset far XXXX:YYYY Segment:Offset ausgegeben
n		speichert die bis jetzt ausgegebenen Zeichen im korrespondierenen Argument

Die Bedeutung der wahlweisen Angabe ***flags*** auf die Darstellung der auszugebenen Daten wird in folgender Tabelle gezeigt.

[flag]	Bedeutung/Wirkung	
keine Angabe	rechtsbündig, führende Nullen werden als Leerzeichen dargestellt	
-	linksbündige Ausgabe; Auffüllen rechts mit Leerzeichen	
+	positiven numerischen Werten wird immer das Pluszeichen vorangestellt	
Leerzeichen	nur negatives Vorzeichen vorangestellt	
#	Konvertierung in alternativer Form	
	c,s,d,i,u	keine Wirkung
	o	bei positiven Werten wird Null vorangestellt
	x,X	0x bzw. 0X vorangestellt
	e,E,f	Dezimalp. immer ausgeben
	g,G	wie e,E, jedoch werden Nullen hinter Dezimalpunkt nicht unterdrückt

Die wahlweise Angabe ***breite*** legt die minimale Anzahl von Stellen in der Ausgabe fest. Wird diese Breite unterschritten, füllt **printf** mit Leerzeichen rechts- oder linksbündig auf, in Abhänigkeit von der Flag-Angabe.

[breite]	Bedeutung/Wirkung
ganze Dezimalzahl n	n entspricht der Mindestbreite
0 n	analog n, statt Leerzeichen werden führende Nullen vorangestellt
*	Angabe der Breite aus Argumentliste

Die Genauigkeit der Ausgabe wird durch einen Punkt eingeleitet.

[genauigkeit]	Bedeutung/Wirkung
keine Angabe	Standardeinstellung eine Ausgabestelle für d,i,o,u,x,X sechs Ausgabestellen für e,E,f
.0	bei e,E,f keine Nachkommastellen
.Dezimalzahl n	bei d,i,o,u,x X werden mindestens n Stellen ausgegeben bei e,E,f werden n Nachkommastellen mit eventueller Rundung ausgegeben bei g,G werden n Stellen mit eventueller Rundung ausgegeben bei c ohne Wirkung bei s werden die ersten n Zeichen des Strings ausgegeben
*	Die Genauigkeit wird analog zur Breite in der Argumentliste angegeben

Mit der wahlweisen Angabe ***modifizierer*** lassen sich vom Speichermodell abweichende Größen festlegen. Bei den kleinen Speichermodellen wird standardmäßig der **near-**Pointer, bei großen Modellen der **far-**Pointer als Defaultwerte bei Zeigerausgaben vorausgesetzt.

Bei numerischen Werten wird grundsätzlich die standardmäßig kürzere Form verwendet, also **int** statt **long** und **float** statt **double**.

[modifizierer]	Interpretation/Wirkung
F	**far** - Zeiger Format: XXXX:YYYY Segment:Offset
N	**near** - Zeiger Format: YYYY nur Offset
h	bei d,i,o,u,x,X **short** Werte
l	bei d,i,o,u,x X **long** Werte bei e,E,f,g,G **double** Werte

Hinweis:

- Soll in einem Text ein Prozentzeichen ausgegeben werden, so muß **%%** spezifiziert werden
- Bei der Eingabe ist zu beachten, daß die Argumente Zeiger sein müssen. Bei einfachen Variablen ist, falls kein Zeiger zur Verfügung steht, die Adresse über einen Adressoperator **&** zu spezifizieren. Bei Feldvariablen ist jede Bezugnahme automatisch ein Zeiger.
- Soll die Ein - oder Ausgabe nicht über die Standarddateien, sondern über **FILE**-Pointer realisiert werden, stehen zwei weitere Funktionenzur Verfügung:
  fscanf(FILE ****stream*,char ****formatstring***,void** ***argumente***);** und
  fprintf(FILE ****stream,*char ****formatstring***,void** ***argumente***);** (siehe Punkt 10.1)

- ° Sollen die formatumgewandelten Zeichenketten im Hauptspeicher abgelegt bzw. für die Eingabe im Speicher Zeichenketten vorausgesetzt werden, können folgende Funktionen genutzt werden:
  sscanf(char *_string_,char *_formatstring_,void _argumente_); und
  sprintf(char *_string_,char *_formatstring_,void _argumente_);
  string ist die Zeichenkette, die bei **sprintf(..)** das Ergebnis der Formatumwandlung aufnehmen soll bzw. bei **sscanf(...)** die umzuwandelnde Zeichenkette enthält.
- ° Die Funktion **prints( )** arbeitet mit Standardattributen; d.h. farbige Ausgaben sind nicht möglich. Für farbige Ausgaben und andere Spezialitäten des PC ist die Funktion **cprintf()** mit analoger Parametrierung vorgesehen.

Programmbeispiel 41:

```
                                                        PAGE  1
                                                        04-16-91
                                                        21:38:44

Line#  Source Line                          Microsoft C Compiler Version 5.10

  1  /* Grenzfaelle formatgesteuerter Ausgabe                                  */
  2  main()
  3  {
  4  puts("\n Ausgabemoeglichkeiten der Integerkonstanten 32767 ");
  5  printf("\n%%d=%d  %%6d=%6d  %%06d=%06d  %%6o=%6o  %%X=%X",
  6  32767,   32767,  32767,    32767,   32767);
  7
  8  puts("\n\n Ausgabemoeglichkeiten der Gleitkommakonstanten -12.345e3 " );
  9  printf("\n%%8f=%8f  %%8.3f=%8.3f  %%8.0f=%8.0f    %%.3f=%.3f",
 10  -12.345e3,  -12.345e3,   -12.345e3,    -12.345e3);
 11  printf("\n%%8E=%8E  %%8.3e=%8.3e  %%8.0e=%8.0e  %%.3e=%.3e",
 12  -12.345e3,   -12.345e3,   -12.345e3,   -12.345e3);
 13  printf("\n%%8g=%8g      %%8.3g=%8.3g  %%8.0g=%8.0g  %%.3g=%.3g",
 14  -12.345e3,     -12.345e3,   -12.345e3,   -12.345e3);
 15
 16  puts("\n\n Ausgabemoeglichkeiten der Zeichenkettenkonstante BERLIN ") ;
 17  printf("\n%%s=%s       %%6s=%6s       %%2s=%2s       %%.3s=%.3s",
 18 "BERLIN",      "BERLIN",      "BERLIN",       "BERLIN");
 19  printf("\n%%10s=%10s  %%-10s=%-10s  %%10.3s=%10.3s  %%-10.3s=%-10.3s ",
 20  "BERLIN",   "BERLIN",    "BERLIN",       "BERLIN");
 21  }
...
No errors detected
```

<u>Folgendes Druckbild entsteht</u>:

Ausgabemoeglichkeiten der Integerkonstanten 32767
%d=32767 %6d= 32767 %06d=032767 %6o=77777 %X=7FFF

Ausgabemoeglichkeiten der Gleitkommakonstanten -12.345E3
%8f=-12345.000000 %8.3f=-12345.000 %8.0f= -12345 %.3f=-12345.000
%8E=-1.23450E+04 %8.3e=-1.23e+04 %8.0e=-1.2345e+04 %.3e=-1.23e+04
%8g= -12345 %8.3g=-1.23e+04 %8.0g=-1.2345e+04 %.3g=-1.23e+04

Ausgabemoeglichkeiten der Zeichenkettenkonstante BERLIN
%s=BERLIN %6s=BERLIN %2s=BERLIN %.3s=BER
%10s= BERLIN %-10s=BERLIN %10.3s= BER %-10.3s=BER

10.3. Dateiverwaltung durch Low - und High - Level Funktionen

Die Low- bzw. High Level Verarbeitung wird auch als File - bzw. Stream Verarbeitung bezeichnet. Die Dateiidentifikation bei der File-Verarbeitung besteht aus einem **int**-Wert, den o.g. Handle, während bei der Stream-Verarbeitung die Dateiidentifikation über eine in der stdio.h definierte **FILE**- Struktur zu realisieren ist. Dieser Strukturtyp **FILE** nimmt den sog. Filepointer, eine erweiterte Variante des Handle, auf.

10.3.1 Die File - Verarbeitung, die Low - Level Funktionen

Im Gegensatz zu der Stream Verarbeitung wird bei der Verarbeitung von Files keine Pufferung vorgenommen. Um die Dateiidentifikation, den Handle eines bestehenden Files zu erhalten, muß dieser mittels der Funktion

int open(char *file,int *access [,permiss]*);

mit den in ***access*** definierten Zugriffsrechten und den wahlfreien Möglichkeiten einen File-Sharing-Betrieb eröffnet werden. Die Anzahl der gleichzeitig offenen Dateien ist durch die **FILES**-Angabe in der **CONFIG.SYS** festgelegt. Die recht aufwendige Schreibweise in ***access*** zur Festlegung der Zugriffsrechte für den File wird durch eine einfache numerische Notation, die in der Include-Datei **fcntl.h** definiert ist, erheblich erleichtert. Die folgende Tabelle zeigt den Zusammenhang zwischen den symbolischen Konstanten ***access*** sowie ihren numerischen Werten:

access	Bedeutung	fcntl.h
O_RDONLY	nur Lesezugriff	1
O_WRONLY	nur Schreibzugriff	2
O_RDWR	Lese- und Schreibzugriff 4	
O_CREAT	legt File an und eröffnet mit Zugriffsrecht permiss	0x0100
O_TRUNC	Inhalt gelöscht,Zugriffsrechte bleiben erhalten	0x0200
O_APPEND	Schreiboperation am Fileende	0x0800
O_TEXT	eröffnen im Textmodus, Konvertierung CR/LF nach LF	0x4000
O_BINARY	eröffnen im Binärmodus, keine Konvertierung von CR/LF	0x8000

Hinweis:

- Die numerischen Angaben in der fcntl.h beziehen sich in dieser Tabelle auf TURBOC 2.0
- Um portale Programme zu schreiben, sollte man sich auf die symbolischen Konstanten beziehen

Der Funktionswert der Funktion **open( )** ist die Dateiidentifikation, der Handle. Für alle weiteren Ein- und Ausgabefunktionen wird dieser File-Handler zur Identifikation auf diesen File benutzt.
Anders als bei der Stream-Verarbeitung muß zum Erzeugen eines neuen Files eine dafür bestimmte Funktion benutzt werden:

creat(char *file,int *access*);

Sie arbeitet ananlog wie **open( ).** ***access*** kann dabei einen beliebigen Wert enthalten, es wird immer automatisch der Schreibmodus gewählt.
Zum Schreiben und Lesen existieren nur zwei Funktionen:

read(int *handle*,void **buf*,unsigned *nbyte*);
write(int *handle*,void **buf*,unsigned *nbyte*);

Es wird ein Datenblock der Länge ***nbyte*** von dem File mit der Identifikation ***handle*** gelesen bzw. in das File geschrieben. Der Zeiger zeigt dabei im Hauptspeicher auf den zu verarbeitenden Datenblock.

Programmbeispiel 42:

```
                                                                PAGE 1
                                                              03-27-91
                                                              21:48:22

Line#  Source Line Microsoft C Compiler Version 5.10

  1  /* Dieses Programm liest sich selbst aus dem aktuellen Verzeichnis        */
  2  /*undschreibtesaufdenAusgabebildschirm                                    */
  3  #include <fcntl.h>
  4  #define DATEI "beispiel.42"
  5  #define BUFSIZE 512
  6  main()
  7  {
  8  int handle;
  9  char buffer[BUFSIZE];
 10  handle = open(DATEI,O_RDONLY);
 11  while( read( handle,buffer,BUFSIZE) > 0)
 12  write(1,buffer,BUFSIZE);  /* Handle 1 ist die Standardausgabe             */
 13  close(handle);
 14  }

main  Local Symbols

Name                 Class   Type         Size   Offset  Register
handle. . . . . .auto                        -0202
buffer. . . . . auto                         -0200

Global Symbols

Name                 Class  Type              Size   Offset
close . . . . . .extern  near function    ***    ***
main. . . . . global   near function    ***    0000
open. . . . . extern  near function    ***    ***
read. . . . . .extern  near function    ***    ***
write . . . . . extern  near function    ***    ***
No errors detected
```

Soll ab einer bestimmten Position im File weitergearbeitet werden, steht die Funktion

lseek(int *handle,*long *offset*,int *fromwhere*);

zur Verfügung. Diese Funktion positioniert den Datei-Zeiger des Files ***handle*** an die Position "offset", relativ zum Datei-Beginn, Datei-Ende oder der aktuellen Position, abhänig von ***fromwhere***. Auch hier kann wieder, ähnlich wie bei **open( )**, zwischen zwei Formen der Angabe für ***fromwhere*** gewählt werden:

fromwhere	Bedeutung	stdio.h
SEEK_SET	relativ zum Datei-Beginn	0
SEEK_CUR	relativ zur aktuellen Position	1
SEEK_END	relativ zum Datei-Ende 2	

lseek( ) übergibt die neue Position des Dateizeigers relativ zum Datei-Beginn, gemessen in Byte als **long**-Wert.

Mit der Funktion

long int = tell(int *handle*);

wird die aktuelle Position des Dateizeigers relativ zum Datei- Beginn ermittelt. Die ermittelte Position wird in Byte als **long**-Wert als Funktionswert zur Verfügung gestellt.

Mit der Funktion

close(int *handle*);

wird der File mit der Identifikation ***handle*** geschlossen.

Hinweis:

° Schließt **close( )** eine Textdatei, die zum Schreiben geöffnet wurde, schreibt die Routine vorher noch das softwaremäßige **CTRL Z,** bevor der File dann tatsächlich geschlossen wird.

10.3.2 Die Stream - Verarbeitung, die High - Level Funktionen

Im Gegensatz zu der File-Verarbeitung wird bei der Verarbeitung von Streams eine Pufferung vorgenommen. Die für den Umgang mit Streams vorgesehenen Funktionen befinden sich in der Standardbibliothek. Diese muß mittels **#include <stdio.h>** eingefügt werden.

Um einen Stream zu Erzeugen oder zu Eröffnen, muß die Funktion

filepointer* = fopen(char ****dateiname,int *modus)*;**

benutzen. Als Ergebnis liefert die Funktion einen Zeiger auf eine Datenstruktur vom Typ **FILE**, der in der Standardbibliothek definiert ist. Diese Zeiger, der sog. Filepointer, muß im Programm mit geeigneten Mitteln gespeichert werden, da er für alle weiteren Datenzugriffe benötigt wird.

**FILE ****filepointer*;

Hinweis:

° Der Dateiname muß die Datei genau spezifizieren. Er muß ggf. die Namen des Laufwerks sowie des Subdirektories enthalten: ***laufwerk:\\subdirektorie\\dateiname***

Für den Dateityp **modus** gilt folgende Tabelle:

modus	**Bedeutung**
"r"	Öffnen zum Lesen; Datei muß existieren
"w"	Erzeugung einer Datei zum Schreiben; falls Datei existiert, wird vorheriger Inhalt gelöscht
"a"	Öffnen zum Schreiben an Dateiende;existiert Datei noch nicht, wird sie erstellt und eröffnet
"r+"	Öffnen einer Datei zum Lesen und Schreiben; Datei muß existieren
"w+"	Öffnen einer neuen Datei zum Lesen und Schreiben; falls Datei existiert, wird alter Inhalt gelöscht
"a+"	Öffnen einer Datei zum Lesen und Schreiben an Dateiende; falls Datei existiert, bleibt alter Inhalt erhalten, ansonsten wird sie erstellt

Hinweis:

° Wenn Lesen und Schreiben erlaubt sind, muß mit den Funktionen **fseek( )** oder **rewind( )** zwischen einem Lese - und einem Schreibzugriff der Filepointer zurückgesetzt werden

Der Dateityp ***modus*** kann noch um die nachgestellten Zeichen **"t"** für Textmodus und **"b"** für Binärmodus erweitert werden. Auch hier wird bei Textdateien beim Schreiben **LF** durch **LF/CR** ersetzt und beim Lesen entsprechend **LF/CR** durch **LF,** was bei Binärdateien natürlich nicht der Fall ist. Sollte weder "t" noch "b" angegeben werden, so wird der defaultmäßige Dateimodus der Globalvariablen **_fmode**, die in der **fcntl.h** definiert ist, zugrunde gelegt.

Zum Lesen und Schreiben existieren eine Vielzahl von Funktionen:

fgetc(int *zeichen*,FILE **stream*);
fputc(int *zeichen*,FILE **stream*);
fgets(char **string*,int laenge,FILE **stream*);
fputs(char **string*,FILE **stream*);
fscanf(FILE **stream*,char **formatstring*,void *argumente*);
fprintf(FILE **stream*,char **formatstring*,void *argumente*);

Diese Funktionen zur zeichenweisen, zeilenweisen und formatgesteuerten Ein - und Ausgabe wurden bereits vorgestellt. Nur hier erfolgen im Unterschied zu den obengemachten Ausführungen alle Ein - und Ausgabeoperationen über Streams und nicht über Standardkanäle.

Neu hingegen sind folgende Funktionen:
fread(void **buffer*,int *anzahl bytes*,int *anzahl sätze*,FILE **stream*)
fwrite(void **buffer*,int *anzahl bytes*,int *anzahl sätze*,FILE **stream*)

Es werden Datenblöcke einer bestimmten Länge von den Stream gelesen bzw. in den Stream geschrieben. Dieser Datenblock kann ein Feld, eine Struktur, Felder von Strukturen, einfache Variablen o.ä.. sein. Der Funktion muss ein Zeiger auf den Datenblock, die Größe eines Elementes, die Anzahl der Elemente sowie die Dateikennung übergeben werden. Soll ab einer bestimmten Position in dem Stream geschrieben oder aus ihm gelesen werden, kommen die Funktionen

fseek(FILE **stream*,long *offset,*int whence);
ftell(FILE **stream*);

zur Anwendung.
Die Parametrierung entspricht den Funktionen **lseek()** und **tell(),** allerdings wird hier als Dateiidentifikation der Filepointer übergeben.

Mittels der Funktion

rewind(FILE **stream)*;

kann der Filepointer im Stream auf die Position Null gesetzt werden. Eine eventuell gesetzte Dateiendekennung **EOF** wird durch **rewind()** gelöscht.

Mit der Funktion

fclose(FILE **stream)*;

wird der angegebene Stream geschlossen und vorher alle dazugehörigen Puffer geleert. Die vom System vergebenen Puffer werden wieder freigegeben, während die mit **setbuf** oder **setvbuf** angeforderten Puffer nicht automatisch freigegeben werden.

Programmbeispiel 43:

```
                                                        PAGE   1
                                                        04-16-91
                                                        22:01:21
Line#  Source Line                      Microsoft C Compiler Version 5.10

    1   /*  Dieses Programm erstellt eine Personen-Datei. Wird beim Eroeffnen
    2   **  festgestellt, daa die Datei noch nicht vorhanden ist,wird sie angelegt.
    3   */
    4   #include <stdio.h>
    5   #include <stdlib.h>                 /* Definition atol und atof          */
    6   #define DATEI "person.dat"
    7   struct persondat
    8   {
    9   int persnr   ;
   10   char name[20];
   11   float gehalt;
   12   long tel;
   13   }
   14
   15   main()
   16   {
   17   struct persondat pers,*ptr;
   18   FILE *fp;
   19   char buffer[20];
   20   char wahl;
   21   ptr =  &pers;
   22   fp =fopen(DATEI,"a+");              /*  öffnen zum Erweitern          */
   23   if (fp == NULL)                     /*  open nicht erfolgreich        */
   24   {
   25   printf("\n Dateifehler");
   26   exit(1);                            /*  Abbruch; RC = 1               */
   27   }
   28   puts("\n\n\n\t\t\tBitte die Stammdaten einer Person eingeben ");
   29   while(1)
   30   {
   31   printf("\n Personennummer:  ");
   32   ptr -> persnr = atoi(gets(buffer));  /* konvertieren 'char to int'       */
   33   printf(" Gehalt:  ");
   34   ptr -> gehalt = atof(gets(buffer));  /* konvertieren 'char to float'     */
   35   printf(" Name:  ");
   36   gets(ptr ->name);
   37   printf(" Tel-nr:  ");
   38   ptr -> tel = atol(gets(buffer));     /* konvertieren 'char to long'      */
   39   fwrite(&pers,sizeof(pers),1,fp);
   40
   41   puts("\n Kontrollausdruck \n");
   42   printf("\n Name:  %10s  Pers.Nr: %d ",
   43   ptr -> name, ptr -> persnr);
   44   printf("\n Gehalt: %6.2f DM  Tel.-Nr.: %ld ",
   45   ptr -> gehalt  ,  ptr -> tel);
   46   printf("\n\n Ausgabesatz hat %d Bytes ",sizeof(pers));
   47   printf(" und beginnt bei Adresse %u\n",&pers);
   48
   49   printf("\n Weitere Personendaten eingeben? (j,n)");

                                                        PAGE 2
                                                        04-16-91
                                                        22:01:21
Line#  Source Line                      Microsoft C Compiler Version 5.10

   50   scanf("%c",&wahl);
   51   if (wahl == 'n') break;
   52   else gets(buffer);                  /* Leeren Eingabepuffer          */
   53   }
   54   fclose(fp);
   55   }

main  Local Symbols

Name                 Class   Type      Size   Offset  Register
wahl. . . . . . . . . . auto                    -0038
pers. . . . . . . . . . auto                    -0036
buffer. . . . . . . . . auto                    -0018
ptr . . . . . . . . . . . auto                  -0004
fp. . . . . . . . . . . . auto                  -0002

Global Symbols
```

```
Name                 Class   Type              Size   Offset
atof. . . . . .extern   near function      ***    ***
atoi. . . . . .extern  near function      ***    ***
atol. . . . . .extern  near function      ***    ***
exit. . . . . .extern  near function      ***    ***
fclose. . . . . .extern  near function      ***    ***
fopen . . . . . extern  near function      ***    ***
fwrite. . . . . extern  near function      ***    ***
gets. . . . . extern  near function      ***    ***
main. . . . . .global  near function      ***    0000
printf. . . . . .extern  near function      ***    ***
puts. . . . . extern  near function      ***    ***
scanf . . . . . .extern  near function      ***    ***

Code size = 0180 (384)
Data size = 013a (314)
Bss size  = 0000 (0)
No errors detected
```

11. Verzeichnis der Test- und Beispielprogramme

Quellcode	Größe	ausführbares Progr.	Größe
B01.C	539 Byte	B01.EXE	9722 Byte
B02.C	190 Byte		
B03.C	984 Byte	B03.EXE	10595 Byte
B04.C	85 Byte	B04.EXE	5675 Byte
B05.C	416 Byte	B05.EXE	6247 Byte
B06.C	507 Byte	B06.EXE	5842 Byte
B07.C	159 Byte	B07.EXE	5699 Byte
B08.C	1486 Byte	B08.EXE	10423 Byte
B09.C	279 Byte	B09.EXE	9736 Byte
B10.C	370 Byte	B10.EXE	5671 Byte
B11.C	1042 Byte	B11.EXE	30677 Byte
B12.C	518 Byte	B12.EXE	31878 Byte
B13.C	701 Byte	B13.EXE	9882 Byte
B14.C	617 Byte	B14.EXE	9891 Byte
B15.C	526 Byte	B15.EXE	9792 Byte
B15A.C	852 Byte	B15A.EXE	9821 Byte
B16.C	662 Byte	B16.EXE	9928 Byte
B17.C	311 Byte	B17.EXE	9769 Byte
B18.C	326 Byte	B18.EXE	9752 Byte
B19.C	1012 Byte	B19.EXE	6628 Byte
B20 C			
B21.C	911 Byte	B21.EXE	24960 Byte
B22.C	586 Byte	B22.EXE	24960 Byte
B23.C	977 Byte	B23.EXE	7378 Byte
B24.C	588 Byte	B24.EXE	6644 Byte
B25.C	515 Byte	B25.EXE	25158 Byte
B26.C	1091 Byte	B26.EXE	3286 Byte
B27.C	677 Byte	B27.EXE	6648 Byte
B28.C	1025 Byet	B28.EXE	6848 Byte
B29.C	584 Byte	B29.EXE	27816 Byte
B30.C	595 Byte	B30.EXE	6818 Byte
B31.C	807 Byte	B31.EXE	6996 Byte
B32.C	762 Byte	B32.EXE	3514 Byte
B33.C	1339 Byte	B33.EXE	6942 Byte
B34.C	434 Byte	B34.EXE	6648 Byte
B35.C	475 Byte	B35.EXE	6622 Byte
B36.C	468 Byte	B36.EXE	27506 Byte
B37.C	946 Byte	B37.EXE	25288 Byte
B38.C	770 Byte	B38.EXE	6720 Byte
B39.C	415 Byte	B39.EXE	5722 Byte
B40.C	308 Byte	B40.EXE	6276 Byte
B41.C	1139 Byte	B41.EXE	26504 Byte
B42.C	498 Byte	B42.EXE	4550 Byte
B43.C	1850 Byte	B43.EXE	31258 Byte