MI - C

C - COMPILER

Version 3.18

von Dipl.- Math. G. Kersting / H. Rose

GEWÄHRLEISTUNGSAUSSCHLUSS

Der Käufer des C - Compilers MI - C verzichtet ausdrücklich auf jede Schadenersatzforderung, falls im Zusammenhang mit dem C - Compiler MI - C Verluste oder Ausgaben entstehen, oder er nicht in der Lage ist, den C - Compiler MI - C für bestimmte, gleichgültig welche Zwecke zu verwenden.
Die Autoren behalten sich das Recht vor, an diesem Handbuch und am C - Compiler MI - C Änderungen vorzunehmen, ohne Verpflichtung diese Änderung irgendeiner Person bekanntzugeben.

EINFOHRUNG

C ist eine universelle Programmiersprache, die sich für vielfältige Anwendungen anbietet. Unter anderem ist es möglich maschienennahe effiziente Programme zu erstellen, die sonst in Assembler geschrieben werden müßten. Die Sprache C ist ausführlich beschrieben in:

"THE C PROGRAMMING - LANGUAGE "
BRIAN W. KERNIGHAN DENNIS M. RITCHIE
ENGLEWOOD CLIFFS NEW JERSEY
PRENTICE - HALL 1978

Eine deutsche Übersetzung dieses Buches ist erhältlich:

PROGRAMMIEREN IN C
KERNIGHAN RITCHIE
MÜNCHEN WIEN
CARL HANSER VERLAG 1983

Das Buch enthält u.a. eine Einführung in C. Zum Verständnis sind nur geringe Kenntnisse der Programmierung erforderlich, (z.B. was ist eine Schleife,Variable), die mit der Kenntnis einer anderen Programmiersprache gegeben sind.

Der Compiler MI - C hält sich an die dort festgelegten Definitionen. Eventuelle Unterschiede zu anderen C Compilern, Erweiterungen und Einschränkungen können im Kapitel E nachgelesen werden.
In diesem Handbuch wird zum einen die Handhabung des Compilers MI - C erklärt zum anderen eine kurze Beschreibung der Sprache C gegeben. Auf Dinge von speziellem Interesse wird ab Kapitel E eingegangen. In Kapitel K befindet sich eine alphabetisch geordnete Fehlermeldungsliste mit Erklärungen.

INHALTSVERZEICHNIS Seite

A. EINIGE EIGENSCHAFTEN DES COMPILERS MI - C

Der Compiler MI - C erstellt aus einem in einer Datei vorliegenden C - Quellprogramm eine Datei mit 8080 oder Z80 Assemblercode. Mittels eines Assemblers wird dann daraus eine Datei mit Maschinencode erzeugt. Der zum CP/M System gelieferte Assembler ASM.COM ist ausreichend. Mehr Komfort bietet eine Kombination von Assembler und Linker (speziell MAC80 und L80 von Microsoft), deren Gebrauch vom Compiler besonders unterstützt wird.
MI - C ist benutzerfreundlich, und mit dem Compiler erstellte Programme sind schnell und kurz. Der Compiler gibt optimierten Code aus. Z.B. wird ein Ausdruck aus Konstanten bereits während der Compilation ermittelt, eine Subtraktion mit einer Konstanten in eine Addition, eine Addition mit einer kleinen Konstanten in eine Inkrementoperation verwandelt, eine Variable, die sich schon durch eine vorhergehende Operation in einem Register befindet, möglichst nicht erneut geladen.
Die minimale Größe für ein vollständiges Programm beträgt einige hundert Byte.

Hervorzuheben ist:

- Der Compiler legt eine Fehlerprotokoll - Datei an. Jede Fehlermeldung besteht aus einer wahlweise deutsch- oder englischsprachigen Fehlermeldung und der C - Quellzeile (mit zugehöriger Zeilennummer), in der der Fehler aufgetreten ist. Die genaue Fehlerstelle ist markiert.
- Der Compiler kann auf Wunsch die C - Quellzeilen als Kommentar mit in die Assemblerdatei ausgeben. So kann genau verfolgt werden, welcher Code aus den jeweiligen C - Anweisungen generiert wird.
- Der Vorgang der Übersetzung des Quellprogramms kann am Terminal verfolgt werden. Der Name der gerade übersetzten Funktion wird auf dem Terminal protokolliert. Zusätzlich wird für jeweils 10 Zeilen ein '-' oder 'E' ausgegeben (ein / kein Fehler aufgetreten). Mittels 'control C' kann der Compilerlauf abgebrochen werden. Die Information in den Ausgabedateien bleibt dabei erhalten.
- Ein C - Quellprogramm von 15k Byte Länge wird i.a. in weniger als 3 Minuten (bei 2MHz) übersetzt. Da keine Assemblermacros erzeugt werden, ist die Assemblierungszeit auch sehr kurz.

- Das Quellprogramm kann nach belieben in einzelne Teile zerlegt werden (eine oder mehrere Funktionen und oder Variablendeklarationen), die dann getrennt übersetzt werden können. Wird mit einem Assembler/ Linker gearbeitet, können die Teile auch getrennt assembliert werden. In diesem Fall erzeugt der Compiler automatisch die nötigen EXT, PUBLIC, DSEG und CSEG Anweisungen für den Assembler, damit der Linker die einzelnen Teile zusammenbinden kann.
- Gleitkommazahlen (float,double) sind als gepackte BCD Zahlen dargestellt. Hierdurch entfällt die Ungenauigkeit, die entsteht, wenn dezimale Gleitkommazahlen binär dargestellt werden. Es wird immer mit 13 Stellen Genauigkeit gerechnet.
- Es steht ein Trace zur Verfügung. Während des Laufes des compilierten Programmes werden die Namen der jeweils aufgerufenen Funktionen auf dem Terminal ausgegeben, der Wert einer Anweisung mit Wert (z.B. A= 5;), der Wert eines Testes (z.B. if (A++)), sowie die zur Anweisung gehörige Zeilennummer des Quellprogramms. Über die Tastatur kann die Protokollierung jederzeit ein- und ausgeschaltet werden.
- Es können sowohl Unterprogramme allgemeiner Art als auch direkt lauffähige Programme für CP/M (... .COM -Dateien)) erzeugt werden. Für diesen Fall steht eine Bibliothek mit Ein-/Ausgabefunktionen zur Verfügung.
- Namen von globalen Variablen und Funktionen treten in der Assemblerdatei unverändert auf, was die Fehlerverfolgung vereinfacht und bei Bedarf einfach Manipulationen und Ergänzungen erlaubt.
  Z.B. wird aus 'VAR = 5;' : LXI H,5 bzw. LD HL,5
  SHLD VAR LD (VAR),HL
- Der erzeugte Code kann in einen ROM - Speicher gebracht werden. Der Datenbereich kann mittels der Assembleranweisung 'ORG' oder mit Hilfe des Linkers an eine beliebige freie Stelle des Speichers gelegt werden.
- Der Compiler sowie die übersetzten Programme können durch Interrupts an beliebiger Stelle unterbrochen werden. Die Register A, B, C, D, E, F, H, L dürfen dabei nicht verändert werden. Geschieht die Unterbrechung während eines Aufrufes von BDOS, so sind die dabei gültigen Beschränkungen zu beachten. (BDOS stellt nur einen begrenzten Raum im Stack zur Verfügung. Ausserdem können die Unterprogramme im jeweiligen BIOS Beschränkungen auferlegen.)

B. DIE ARBEIT MIT DEM COMPILER

I. AUFRUF, PROTOKOLLIERUNG, OPTIONEN, TRACE

Ein C - Programm besteht aus einer oder mehreren Funktionen. Der Compiler erzeugt daraus Assemblercode Unterprogramme. Das von anderen Pogrammiersprachen her bekannte Hauptprogramm ist eine Funktion mit dem Namen MAIN. MAIN wird beim Start eines compilierten und assemblierten C - Programms automatisch aufgerufen.
Der Compiler wird gestartet durch:

CC NAME'cr' ('cr' bedeutet RETURN)

Das C Quellprogramm muß in der Datei NAME.C sein. Es werden die Dateien NAME.MAC mit dem Assemblerprogramm und NAME.LST mit den Fehlermeldungen erzeugt.
Wenn andere Optionen als die voreingestellten benötigt werden, können sie vor dem Dateinamen eingeleitet durch / angegeben werden. z.B.:

CC /CS NAME'cr'

Wird gestartet durch CC'cr' ,so fordert der Compiler die benötigte Information an. Diese Form muß angewandt werden, wenn die Quell- oder Zieldateien einen anderen Zusatz als .MAC (.ASM) und .C haben. Es werden solange Eingabedateien angefordert bis nur RETURN eingegeben wird. Wird bei der Ausgabedatei nichts angegeben d.h. 'cr', so wird die Ausgabe am Terminal protokolliert. Während des Laufs des C - Compilers werden die Namen der gerade übersetzten Funktionen auf dem Terminal protokolliert. Hinter dem Namen wird für jeweils 10 fehlerfreie Zeilen ein '-' gesetzt. Erscheint ein 'E' so bedeutet das, daß innerhalb dieser 10 Zeilen ein Fehler entdeckt wurde. '-', denen kein Name vorausgeht, gehören zu externen Definitionen.
Außer der Ausgabe - Datei wird noch eine Fehlerprotokoll - Datei mit gleichem Namen und dem extend .LST erzeugt. Sie enthält die Fehlermeldungen des Compilers mit den dazugehörigen Zeilennummern aus der Quelldatei.
Der Compiler kann durch Eingabe von 'control C' unterbrochen werden. Die Ausgabedateien werden dann vor dem Sprung zum Betriebssystem (WBOOT) geschlossen. Die bis zu diesem Zeitpunkt gewonnene Information geht also nicht verloren.

Folgende Optionen verändern die Compilation:

* Das Programm wird nur auf Fehler untersucht. Es wird kein Assemblercode ausgegeben. Wenn kein Ausgabedateiname angegeben wird, werden die Fehlermeldungen auf dem Terminal protokolliert.

C Das C Quellprogramm wird als Kommentar mit in die Assemblerdatei ausgegeben, sodaß verfolgt werden kann welcher Assemblercode aus den jeweiligen C - Quellzeilen erzeugt wird.

L Es wird keine Fehlerprotokolldatei angelegt. Fehlermeldungen werden mit in die Assemblercodedatei als Kommentar ausgegeben.

T Programmteile die unter dieser Option übersetzt werden, bekommen Information für den Trace mit. (T setzt automatisch die Option X.)

X Eine Anweisung vom Typ Ausdruck hinterläßt ihren Wert im HL Register (beim Typ long in CLPRIM und double in CFPRIM). Benötigt wird diese Option nur, wenn mit #asm .. #endasm Assemblercode eingefügt wird, und dabei der Wert von einem Ausdruck in dem Assemblerteilprogramm verwandt wird.

S Der Wert eines char liegt bei dieser Option zwischen 0 und 255 und nicht wie sonst zwischen -128 und 127. (S bietet sich an, wenn char Variable nicht Zahlen sondern Zeichen darstellen sollen.)

A Diese Option wird benötigt, wenn kein Linker benutzt wird. Die Voreinstellung für die Ausgabedatei ist dann .ASM. Die Ausgabe von externen Bezügen unterbleibt. Außerdem wird eine Labelnummer angefragt (Voreinstellung : 0 ; Siehe Beispiele ohne Linker). Unter der Option A gelten zwei Einschränkungen. Statische interne Variablen dürfen nicht initialisiert werden, und statische Funktionen müssen vor dem ersten Auftreten deklariert werden.

G Die Speicherplatzdefinition für nicht initialisierte Variablen unterbleibt. Im Normalfall wird diese Option nicht benötigt.

. Alle vom Compiler ausgegebenen Namen (Assemblerlabels) werden mit einem Punkt am Anfang ausgegeben. Namenskonflikte mit vom Assembler reservierten Namen treten dann nicht auf. (z.B. Beim MAC80 sind Registernamen reserviert.) Es wird dann XXPMAIN.REL und LIBP.REL verwandt.

0 Die Datei -.LST wird bei fehlerfreier Compilation (0 Fehlern) gelöscht.

: Dem Doppelpunkt muß ein Buchstabe zwischen A und P

einschließlich folgen. Dieser Buchstabe gibt das Laufwerk an, von dem die Quelldatei genommen werden soll. (z.B. A:CC /:B C:BEISPIEL)

D Strings werden unter dieser Option immer im Datenbereich abgelegt. Der Platz für Strings ist dann nicht begrenzt.

W Bei der Option W kann bei einer switch - Anweisung default an beliebiger Stelle stehen. Dadurch vergrößert sich der ausgegebene Code bei jedem switch mit default um 2 Sprünge. Ohne diese Option darf nach default kein case auftreten.

Zum TRACE :
Wenn einem Programmteil Information für den Trace hinzugefügt ist (Option T), so wird beim Programmlauf jeder Funktionsaufruf auf dem Terminal protokolliert. Außerdem wird bei jeder Anweisung 'ausdruck;', 'if(ausdruck)', 'while (ausdruck)', 'for (..;ausdruck;..)'l 'switch (ausdruck)' die Zeilennummer aus dem Quellprogramm und der gerade errechnete Wert ausgegeben. Bei ganzzahligem Typ wird er als Dezimalzahl und als Hexadezimalzahl ausgegeben. Durch Eingabe des Zeichens 'N' von der Tastatur wird der Trace ausgeschaltet. 'X' schaltet ihn wieder ein und bei 'T' werden nur die Funktionennamen protokolliert. Die eingegebenen Zeichen erscheinen auch auf dem Terminal.

II. BEISPIELE ZUR COMPILATION

Es werden nun einige mögliche Wege zur Erstellung eines unter CP/M lauffähigen C-Programms beschrieben. Programme, die auf anderen Betriebssystemen laufen sollen, benötigen andere selbst zu schreibende Ein-, Ausgabe und Initialisierungsunterprogramme.
Bei den folgenden Beispielen bedeuten Kleinbuchstaben die Eingaben vom Terminal und Großbuchstaben die Ausgaben zum Terminal. Nehmen wir an, das C - Quellprogramm ist in der Datei ZB.C .

a. MIT LINKER

Hier wird die Kombination L80/M80 von Microsoft zugrunde gelegt. Bei anderen Produkten wird ähnlich verfahren.
Das Laufzeitsystem und die Ein- Ausgabeprogramme befinden sich in der Bibliotheksdatei LIB.REL. XXXMAIN.REL muß immer beim Linken als erstes angegeben werden.

```
A>cc zb'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983

MAIN    --
OUTSTRIN-
NL      -
;0 FEHLER

A>m80 =zb'cr'

 NO FATAL ERROR(S)

A>180 zb/n,xxxmain,zb,lib/s/e'cr'
```

Nun haben sie eine Datei zb.com erzeugt. Das Programm kann nun durch Eingabe von zb gestartet werden.
Befindet sich die Funktion OUTSTRIN in einer anderen Datei (z.B. um Zeit zu sparen, denn bei Programmänderungen braucht sie dann nicht mehr neu compiliert werden), und soll sie getrennt übersetzt werden, so geht man wie folgt vor. Der Name dieser Datei sei ZBOUT.C .

```
A>cc zb'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983

MAIN    --
NL      -
;0 FEHLER

A>m80 =zb'cr'

 NO FATAL ERROR(S)

A>cc zbout'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983
OUTSTRIN-
;0 FEHLER

A>m80 =zbout'cr'

 NO FATAL ERROR(S)

A>180 zb/n,xxxmain,zb,zbout,lib/s/e'cr'
```

Es folgt als weiteres Beispiel der Dialog am Terminal, wenn sich die Datei ZB.C auf Laufwerk D befindet.

```
---------------------------------------------------------
A>d:cc'cr'

MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983

OPTIONEN ('*'=NUR FEHLER 'T'=TRACE ) ? cs'cr'

AUSGABEDATEI ? zb.src'cr'

EINGABEDATEI ? d:zb.c'cr'

MAIN    --
OUTSTRIN-
NL      -
EINGABEDATEI ? 'cr'

;0 FEHLER
---------------------------------------------------------
```

Mit Hilfe des Linkers (/D:.... bei L80) kann man, auch wenn getrennt compiliert wurde, den Variablenbereich ans Ende des Programms legen (kleinere .. .COM möglich), was sich bei der Benutzung von langen nicht initialisierten Feldern empfiehlt.

b. OHNE LINKER

Als Beispiel wird das Vorgehen mit Hilfe des Assemblers ASM.COM (von Digital Research zum CP/M mitgeliefert) betrachtet.
ZB.C enthält ein C Quellprogramm.
Direkt als erstes muß die Quelle #include CSTART.CC enthalten. In CSTART.CC sind u.a. die Definitionen der Eingangspunkte für das Laufzeitsystem, das sich in LIB.HEX befindet.

Sie führen nun den folgenden Dialog:

```
A>cc /a zb'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983
-----
MAIN    --
OUTSTRIN-
NL      -
;0 FEHLER

A>asm zb.aaz'cr'

A>ddt lib.hex'cr'
DDT VERS 2.2
-izb.hex
-r
NEXT  PC
xxx  0100
-g0

A>save yyy  zb.com
```

xxx - 1 ist die höchste vorkommende Adresse im Maschinenprogramm, aus der yyy errechnet wird. (yyy ist die Anzahl der Pages.)
Das Programm kann nun gestartet werden.

Werden Teile der Bibliothek nicht benötigt (z.B. Disk Ein- Ausgabe), so können sie vom Programm überlagert werden, und dieses dadurch verkürzt werden. Das geschieht dadurch, daß eine von 4 vorhandenen Alternativen bei der Assembler- ORG Anweisung am Anfang der Datei CSTART.CC aktiviert wird.

Besteht das Programm aus mehreren Teilen, die getrennt compiliert werden sollen, so darf nur im ersten Teil #include CSTART.CC stehen. Außerdem muß bei allen Teilen explizit eine Startlabelnummer angegeben werden.
Das geschieht folgendermaßen :

```
-----------------------------------------------------------
A>cc'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983

OPTIONEN ('*'=NUR FEHLER 'T'=TRACE ) ? a'cr'

ANFANGS LABEL : 500'cr'

AUSGABEDATEI ? zbout.asm'cr'
EINGABEDATEI ? zbout.c'cr'

OUTSTRIN-
EINGABEDATEI ? 'cr'
;0 FEHLER
-----------------------------------------------------------
```

Nach der Option A wird die Anfangslabelnummer erfragt. Sie hat folgende Bedeutung :
Der C - Compiler erzeugt Assemblerlabels, die die Form Cxxxxx haben, wobei xxxxx eine Zahl ist (z.B. C200:). Die gleichen Zahlen dürfen nicht in verschiedenen Programmteilen auftauchen, da, wenn ohne Linker gearbeitet wird, alle Teile gemeinsam assembliert werden müssen. Es wird die Anfangszahl angefragt.

Die Dateien ZB.ASM und ZBOUT.ASM müssen nun gemeinsam assembliert werden. z.B.:

```
A>pip zb.asm=zb.asm,zbout.asm'CR'

A>asm zb.aaz'cr'
```

Die Bibliothek liegt auch in Quellform vor. Wenn dort Änderungen vorgenommen werden, oder C - Funktionen als Unterprogramme für andere Programme verwendet werden sollen, so müssen die dort aufgerufenen Bibliotheksprogramme gemeinsam mit diesen Funktionen assembliert werden. Die Datei LIB.HEX wird dann nicht benutzt.

z.B.: ZB.C enthält nicht #include CSTART.CC

```
------------------------------------------------------------
A>cc /a zb'cr'
MI - C  v-3.18   (C) COPYRIGHT  G. KERSTING / H. ROSE, 1983

MAIN    --
OUTSTRIN-
NL      -
;0 FEHLER

A>pip zb.asm=cstart.asm,clib.asm,iofu.asm,zb.asm'cr'

A>asm zb.aaz'cr'

A>load zb'cr'

------------------------------------------------------------
```

Das Programm kann nun gestartet werden.

C. KURZE BESCHREIBUNG DER SPRACHE C

I. NAMEN, SCHLUSSELWORTE, KONSTANTEN, STRINGS, TRENNUNGSZEICHEN, KOMMENTARE

I1 NAMEN

Namen bestehen aus Ziffern, Buchstaben und dem Unterstreichungszeichen "_". Das erste Zeichen eines Namens muß ein Buchstabe sein. Nur die ersten 8 Zeichen eines Namens sind signifikant, er kann aber beliebig lang sein. (d.h. ABCDEFGH5 und ABCDEFGH100 werden nicht unterschieden) Bei Namen werden Klein- und Großbuchstaben unterschieden. (d.h. ABC ist verschieden von abc)

I2 SCHLUSSELWORTE

Schlüsselworte sind Namen, die bereits eine bestimmte Bedeutung haben. Sie dürfen auf keinen Fall anderweitig verwendet werden.
Liste der Schlüsselworte:

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

Bei Schlüsselworten werden Klein- und Großbuchstaben vom Compiler MI - C nicht unterschieden. (INT = int)

I3 KONSTANTEN

I3a GANZZAHLIGE KONSTANTEN

Es gibt drei Arten von ganzzahligen Konstanten:
1. Dezimale, ganzzahlige Konstanten bestehen aus den Ziffern 0 bis 9.
2. Falls die erste Ziffer 0 ist, wird die ganzzahlige Konstante als Oktalzahl aufgefaßt. Die Ziffern 8 und 9 haben dabei den oktalen Wert 10 bzw. 11.
3. Falls die ganzzahlige Konstante am Anfang die Zeichen 0X enthält, wird sie als Hexadezimalzahl aufgefaßt.

a oder A bis f oder F entsprechen den dezimalen Werten 10 bis 15.
Falls der Wert einer dezimalen Konstanten kleiner oder gleich 32767 ist, hat sie den Typ int. Ist ihr Wert größer als 32767, so ist sie vom Typ long. Hexadezimale und oktale Konstanten sind vom Typ unsigned, wenn ihr Wert kleiner oder gleich 65535 ist. Sonst haben sie den Typ long. Eine ganzzahlige Konstante, auf die unmittelbar der Buchstabe l oder L folgt, ist ebenfalls vom Typ long. Der größte Wert einer long Konstanten ist 2147483647.

I3b ZEICHENKONSTANTEN

Eine Zeichenkonstante ist ein in Apostroph eingeschlossenes Zeichen. z.B. 'A', 'B', 'C', '0'.
Der Wert eines solchen Zeichens ist der Wert, den das Zeichen im ASCII - Code hat.

Beispiel
'0' hat den Wert 30H. '1' hat den Wert 31H. 'A' hat den Wert 41H.

Einige nichtdarstellbare Zeichen besitzen eine Ersatzdarstellung. Im folgenden die Ausnahmen und ihre Darstellung als Zeichenkonstante:

Name	Zeichen	Darstellung als Zeichenkonstante
Schrägstrich (backslash)	\	'\\'
Rückwärtsschritt (backspace)	BS	'\b'
Wagenrücklauf (carriage return)	CR	'\r'
Seitenvorschub (form feed)	FF	'\f'
horizontale Tabulation (horizontal tab)	TAB HT	'\t'
neue Zeile (newline)	NL (LF)	'\n'
Apostroph (single quote)	'	'\''

Außerdem kann ein binärer Wert unmittelbar als oktale Konstante angegeben werden. '\zzz' ist Zeichenkonstante,

wobei zzz 1, 2 oder 3 oktale Ziffern sein können. z.B. ist '\0' die binäre Null. Ihr Wert ist 0. (Im Gegensatz zur Zeichenkonstanten '0', die den Wert 30H besitzt.) Zeichenkonstanten sind stets vom Typ int.

I3c GLEITKOMMA (FLOAT) KONSTANTEN

Eine Gleitkommakonstante besteht aus einem ganzzahligen Teil, einem Dezimalpunkt, einem gebrochenen Teil und einem e oder E gefolgt von einem Exponenten mit oder ohne Vorzeichen. Wenigstens eines von beidem: Dezimalpunkt oder E (e) mit Exponent muß vorkommen. Ebenso muß wenigstens eines von beidem ganzzahliger oder gebrochener Teil vorkommen.

Beispiel

3.4567E50	ist Gleitkommazahl
6.0	ist Gleitkommazahl
6	nicht, weder . noch E oder e
.033	ist Gleitkommazahl
.e9	nicht, weder ganzzahliger noch gebrochener Teil

Gleitkomma Konstanten sind stets vom Typ double.

I4 STRINGS

Ein String ist eine Folge von Zeichen, die in Anführungsstriche eingeschlossen ist. Der Wert eines Strings ist ein Pointer, der auf ein Feld zeigt, das die zwischen den Anführungsstrichen stehenden Zeichen enthält gefolgt von einer binären Null. Die Ersatzdarstellungen sind dieselben wie bei den Zeichenkonstanten. (z.B. \B für Backspace) Nicht sinnvoll ist \0, da hierdurch der String beendet würde. Falls im String selbst das Zeichen " vorkommen soll, muß ihm ein \ vorangehen!
Beispiel "DIES IST EIN STRING"

I5 TRENNUNGSZEICHEN

Trennungszeichen (Leertaste (blank), Tab, Neue Zeile (new line)) werden ignoriert.

I6 KOMMENTARE

Kommentare müssen mit /* beginnen und werden mit */ beendet. Kommentare können auch über mehrere Zeilen gehen.
Beispiel /* DIES IST EIN KOMMENTAR */

II. DEKLARATIONEN

Der C - Compiler kann nur Objekte verarbeiten, die vorher deklariert wurden. (Ausnahme: Ein unbekannter Name, auf den '(' folgt, wird automatisch als Name einer int Funktion definiert.) Mit Objekt ist ein Bereich im Speicher gemeint. Deklarationen sehen allgemein so aus:
Speicherklasse Typname Liste_von_zu_deklarierenden_Namen_
,_die_durch_Kommata_getrennt_sind ;
Wenigstens eines von beidem Speicherklasse oder Typ muß in einer Deklaration vorhanden sein. Fehlt die Angabe eines Typs, so wird automatisch int angenommen.

Beispiel

```
      int VA, VB, VC;        Typ   3 Objekte  ;
      static  int VD;        Speicherklasse Typ  1 Objekt ;
```

Zu den Namen können die Zeichen [], (), * treten, wodurch eine theoretisch unbegrenzte Anzahl abgeleiteter Typen definiert werden kann.
[] bedeutet Feld vom Typ ...
() bedeutet Funktion, die Typ ... zurückliefert
* bedeutet Pointer auf den Typ ...
Außerdem besteht noch die Möglichkeit mittels struct ... { } oder union ... { } solche Objekte zu einer Einheit zusammenzusetzen.
Hier einige Beispiele:

Beispiele

```
int VA;          die Variable VA ist vom Typ int
int *VA;         VA ist Pointer auf eine Variable vom Typ
                 int, *VA bezieht sich auf diese Variable
                 und VA enthält die Adresse von *VA
int VA();        VA ist eine Funktion ohne Parameter,
int MAX(A,B);    MAX  ist  eine Funktion mit  Parametern,
 {return         die  einen Wert vom Typ int  zurücklie-
 A<B ? B:A;}     fern
long (*VA)();    VA ist ein Pointer auf eine Funktion, die
                 einen Wert vom Typ long zurückliefert
char VA[3];      VA  ist ein Feld,  das drei Elemente  vom
                 Typ char enthält ( VA[0], VA[1], VA[2] )
int  *VA[3];     VA  ist ein Feld,  das drei  Pointer  auf
                 Werte vom Typ int enthält
int  (*VA)[3];   VA ist ein Pointer,  auf ein  Feld,  das
                 drei Werte vom Typ int enthält

int Q[2][3][4];  Q ist ein dreidimensionales Feld (2x3x4)
```

```
long **VLP          ist ein Pointer auf einen Pointer auf
                    ein Objekt vom Typ long.
struct CORNELIA *( *( *VC ) [3] ) ();
                    VC ist Pointer auf ein Feld, das aus
                    drei Pointern auf Funktionen, die Poin-
                    ter auf eine Struktur vom Typ CORNELIA
                    zurückliefern, besteht.
-------------------------
```

Die Klammern in den Beispielen sind nötig, da die Modifikatoren verschiedene Priorität haben. () und [] binden stärker als *. Zu lesen ist eine solche Deklaration von innen, also vom Namen aus nach außen.

Die Speicherklasse eines Objektes macht Aussagen über die "Lebensdauer" dieses Objektes. (Existiert es nur während eines Funktionsaufrufes oder während des gesamten Programmes?)
Der Typ eines Objektes macht Aussagen über die Länge des Speicherplatzes und darüber wie das darin enthaltene Bitmuster zu interpretieren ist. (Wieviele Byte müssen für Werte, die das Objekt annehmen kann, reserviert werden? Soll FFFFH als -1 oder als 65537 interpretiert werden?)

II1 SPEICHERKLASSEN

Es gibt fünf Speicherklassen: auto, register, static, extern, typedef.
Außerhalb von Funktionen erklärte Objekte sind externe Objekte. Innerhalb von Funktionen erklärte Objekte sind interne Objekte.

EXTERNE OBJEKTE

Externe Objekte sind entweder im ganzen Programm oder nur in einem getrennt compilierten Programmteil bekannt. Letztere wollen wir statisch nennen. Sie werden durch die Angabe der Speicherklasse static in der Deklaration erklärt. Fehlt die Angabe einer Speicherklasse in der Deklaration, so ist dieses Objekt im ganzen Programm bekannt. Wir wollen diese Objekte globale Objekte nennen. Tritt das Wort extern bei der Erklärung eines Objektes auf, so wollen wir von einer Definition und nicht von einer Deklaration reden, da hier kein Objekt deklariert wird, sondern nur der Typ eines Objektes festgestellt wird und gesagt wird, daß die Deklaration an anderer

Stelle erfolgt. Speicherplatz für ein Objekt wird nur bei der Deklaration, nicht aber bei einer Definition angelegt. Definitionen ein und desselben externen Objektes dürfen an verschiedenen Stellen eines Programmes auch innerhalb von Funktionen auftreten. Deklarationen müssen genau einmal geschehen.
Jedes Objekt muß vor dem ersten Auftreten definiert sein. Wenn es in verschiedenen, getrennt compilierten Programmteilen benutzt wird, muß es in jedem davon definiert und in einem deklariert werden. Der Speicherplatz für ein externes Objekt wird bei der Deklaration einmal angelegt und ist während des ganzen Programmlaufs vorhanden.

INTERNE OBJEKTE
Interne Objekte können temporär oder statisch sein. Statische Objekte erklärt man durch die Angabe von static in der Deklaration. Sie werden genauso wie externe statische Objekte angelegt, mit der Einschränkung, daß ihre Namen nur in der Funktion (bzw. in dem Block, in der (dem) sie deklariert wurden bekannt sind. Auch ihr Speicherplatz ist während des gesamten Programmlaufs vorhanden.
Temporäre Objekte werden durch die Angabe der Speicherklasse auto oder register in der Deklaration erklärt. Ihre Namen sind nur in der Funktion (bzw. in dem Block), in der (dem) sie deklariert wurden bekannt. Bei den bis jetzt vorgestellten Objekten ist der Speicherplatz während des gesamten Programmlaufes vorhanden. Bei temporären Objekten ist das anders. Sobald die Funktion (bzw. der Block), in der (dem) sie deklariert wurden verlassen wird, existiert ihr Speicherplatz nicht mehr. Er wird bei jedem dieser Funktionsaufrufe (bzw. bei Eintritt in diesen Block) neu angelegt. register ist das gleiche wie auto. Dem Compiler soll durch register nur ein Hinweis gegeben werden, den Zugriff auf ein häufig benutztes Objekt zu optimieren. Temporäre Objekte dürfen nur innerhalb von Funktionen erklärt werden!

Falls bei der Deklaration keine Speicherklasse angegeben wird, wird bei einer Deklaration innerhalb von Funktionen die Speicherklasse auto, bei einer Deklaration außerhalb von Funktionen ein globales Objekt angenommen.

Sollen mehrere Dateien getrennt compiliert werden, die mit den gleichen Objekten arbeiten, so müssen diese Objekte extern sein. Variablen als static zu deklarieren bietet sich an, wenn man sie nur von bestimmten Funktionen aus verändern und anderen Programmteilen keine

Zugriffsmöglichkeit einräumen möchte. Falls eine Funktion rekursiv aufgerufen wird, sollte man mit der Deklaration von internen static Variablen in dieser Funktion vorsichtig sein, da bei jedem erneuten Aufruf der Funktion der alte Wert überschrieben wird. Vorzugsweise sollte man bei rekursiven Funktionen Objekte der Speicherklasse auto verwenden.

Mit typedef wird kein Speicherplatz angelegt, sondern nur ein Name bestimmt, der später anstelle eines Typs (Schlüsselwortes) benutzt wird. An einem Beispiel sieht man am besten, was Typedef macht:

Beispiel
statt
```
int IVAR;
char NAME[9];
```

ist nach
```
typedef char NAMTYP[9];
typedef int PRIM;
```

ebenso zulässig
```
PRIM IVAR;
NAMTYP NAME;
```

Beispiel zu II DEKLARATIONEN

Beispiel
Datei BSP:

```
static VG;
int VA;
float VB[8];
extern int VF;

FUNKT()
{
int VC;
extern int VH;
static int VD;
auto char VE;
     ...

}
```

VG ist eine externe statische Variable, auf sie kann innerhalb der Datei BSP jede Funktion zugreifen. VA und VB sind externe Variablen, auf die auch von anderen Dateien aus zugegriffen werden kann. VF ist eine Variable, für die bereits in einer anderen Datei Speicherplatz angelegt wurde. In dieser anderen Datei ist VF eine externe Variable. Hier wird durch extern int nur auf die Existenz und den Typ von VF hingewiesen. Da bei VC keine Speicherklasse angegeben wurde, hat VC ebenso wie VE die Speicherklasse auto. Jedesmal, wenn FUNKT aufgerufen wird, werden die Speicherplätze für VC und VE erneut angelegt und beim Verlassen der Funktion FUNKT "vergessen". Der für VD angelegte Speicherplatz bleibt bestehen.
extern int VH innerhalb der Funktion FUNKT gibt bekannt, daß außerhalb der Funktion FUNKT eine int Variable mit Namen VH existiert. (VH muß in einer Datei deklariert sein.) VC, VD und VE sind nur in FUNKT bekannt, VG, VH, VA, VB und VF in der gesamten Quelldatei und auf VA, VB, VH und VF können auch andere Dateien zugreifen.

II2 TYP UND ARITHMETISCHE UMWANDLUNGEN

Man unterscheidet die folgenden Typen von Variablen:

char	Zeichen 1 Byte lang, wird bei Bearbeitung in int umgewandelt
int	Integer (ganze) Zahl, 2 Byte lang, das höchste Bit wird als Vorzeichen interpretiert
short	genau so wie int
long	doppelt genaue int, 4 Byte lang
unsigned	ganze Zahl,2 Byte lang
float	Gleitkommazahl, 4 Byte lang, wird beim Bearbeiten in double gewandelt
double	doppelt genaue float, 8 Byte lang

long oder long int, unsigned oder unsigned int, long float oder double haben jeweils die gleiche Bedeutung.
Falls bei einer Deklaration eine Speicherklasse angegeben wurde, kann die Angabe eines Typs entfallen. Dann wird angenommen, der Typ sei int. (static A entspricht static int A).

ARITHMETISCHE UMWANDLUNGEN
Viele Operatoren verursachen Umwandlungen.

Beispiel
...
int VARA;
float VARB;
VARA = VARA * VARB;
Zunächst wird der Wert von VARB von float nach double gewandelt, da mit doppelter Genauigkeit gerechnet wird. Der Wert aus VARA wird nach den unten angegebenen Regeln von int nach double gewandelt. Das Ergebnis ist vom Typ double. Da VARA vom Typ int ist, wird das Ergebnis vom Typ double nach int gewandelt und dann VARA zugewiesen. Hierbei ist zu beachten, daß bei der Umwandlung nicht geprüft wird, ob die umzuwandelnde Zahl für den neuen Typ zu groß oder zu klein ist.

Liste der automatischen arithmetischen Umwandlung:
- Jeder Operand vom Typ char wird nach int und jeder Operand vom Typ float wird nach double gewandelt.
- Falls einer der Operanden vom Typ double ist, wird der andere nach double gewandelt. Das Ergebnis ist ebenfalls vom Typ double.
- Anderenfalls: Ist einer der beiden Operanden vom Typ long, so wird der andere ebenfalls nach long gewandelt. Das Ergebnis ist ebenfalls vom Typ long.
- Anderenfalls: Ist einer der Operanden vom Typ unsigned, wird auch der andere nach unsigned gewandelt. Das Ergebnis ist vom Typ unsigned.
- Anderenfalls: Beide Operatoren müssen vom Typ int sein. Das Ergebnis ist vom Typ int.

II3 FELDER, POINTER, STRUCT UND UNION

Ein Feld besteht aus Objekten gleichen Typs. Die Länge des Feldes, d.h. die Anzahl der in diesem Feld zusammengefaßten Objekte, wird in der Deklaration in eckigen Klammern hinter den Feldnamen geschrieben.
Speicherklasse Typ Feldname [Feldlänge];

Beispiel
int ZAHLEN [30] ;
das Feld ZAHLEN besteht aus 30 Objekten vom Typ int.

Auf die einzelnen Objekte wird durch Feldname [entsprechender Index] zugegeriffen. Wichtig: die Indizierung be-

ginnt bei 0 und hört mit Feldlänge - 1 auf.
Im Beispiel ist das erste Objekt durch ZAHLEN [0], das zweite durch ZAHLEN [1], ..., das letzte durch ZAHLEN [29] zu erreichen.
Feldname alleine zeigt auf das erste Objekt des Feldes.
im Beispiel: ZAHLEN ist dasselbe wie & ZAHLEN [0].
Man kann die einzelnen Feldelemente statt durch Feldname[i] ebenso durch *(Feldname + i) erreichen.
im Beispiel: ZAHLEN[i] = *(ZAHLEN + i)
Ebenfalls gleichbedeutend ist:
ZAHLEN + i und & ZAHLEN [i].
Ein Pointer kann ähnlich wie ein Feldname behandelt werden. Der hauptsächliche Unterschied ist der, daß bei der Deklaration eines Pointers kein Speicherplatz für die Objekte, auf die der Pointer zeigt, freigehalten wird, sondern nur eine "Variable" deklariert wird, die eine Adresse enthalten kann. Deshalb muß einem Pointer vor der Benutzung ein Wert zugewiesen werden.
Falls man einen Pointer PZAHLEN erklärt
int *PZAHLEN;
und ihm die Anfangsadresse des Feldes zuweist
PZAHLEN = ZAHLEN;
so kann man auch hier die einzelnen Objekte des Feldes gleichermaßen durch PZAHLEN[i] oder *(PZAHLEN + i) erreichen, was gleichbedeutend mit ZAHLEN[i] oder *(ZAHLEN + i) ist.
Allgemein kann jeder Feld und Indexausdruck durch einen Pointer + Offset ausgedrückt werden.
Zu beachten ist dabei, daß ein Pointer eine Variable ein Feldname aber eine Konstante ist. Somit sind Ausdrücke wie PZAHLEN++ (meint das nächste Objekt des Feldes Zahlen) PZAHLEN += 29 (meint das letzte Objekt)
erlaubt, aber ZAHLEN = PZAHLEN oder ZAHLEN++ unzulässig.

Ein Feld kann keine Funktionen enthalten. Pointer dagegen können auch auf Funktionen zeigen.

Beispiel

```
fu()
{ int (*fupoi)();
   ...
  fupoi = fu;
  (*fupoi)(); }
```

In der letzten Zeile wird die Funktion fu rekursiv aufgerufen.

Ein mehrdimensionales Feld wird als Feld von Feldern aufgefaßt, deren Dimension um 1 kleiner ist als die Dimension des Feldes selber.

Beispiel
```
int Q[2][3][4];
```
Q ist ein dreidimensionales Feld (2x3x4). Q besteht aus zwei zweidimensionalen Feldern, die wiederum bestehen je aus drei eindimensionalen Feldern, jedes der eindimensionalen Felder besteht aus 4 ganzzahligen Werten.

```
Q[i]       ist zweidimensionales Feld
Q[i][j]    ist eindimensionales Feld
Q[i][j][k] ist Wert vom Typ int
```

Beim linearen Durchlaufen des Feldes läuft der letzte Index am schnellsten. d.h. im Beispiel würde auf das Feld Q der Reihe nach so zugegriffen:

```
Q[0][0][0], Q[0][0][1], Q[0][0][2], Q[0][0][3],
Q[0][1][0],    ...                  Q[0][1][3],
Q[0][2][0],    ...                  Q[0][2][3],
Q[1][0][0],    ...                  Q[1][0][3],
Q[1][1][0],    ...                  Q[1][1][3],
Q[1][2][0], Q[1][2][1], Q[1][2][2], Q[1][2][3]
```

STRUKTUREN

Eine Struktur ist einem eindimensionalen Feld ähnlich. Im Gegensatz zu einem Feld können aber bei einer Struktur die Elemente verschiedenen Typ haben. Außerdem geschieht der Zugriff nicht über einen Index sondern über einen Namen.

Beispiel
```
struct ADRESSE   {
                 int POSTLZ;
                 char ORT[20];
                 char STR[18];
                 int NR;
                 } ;
```

Eine Strukturdeklaration beginnt mit struct, es folgt ein Name (hier: ADRESSE), der die Struktur benennt. Danach werden eingeschlossen in Klammern {} die Elemente definiert.
Ein Ausdruck struct Strukturname kann nun genauso als Typ

verwandt werden wie z.B. int oder char.
Durch struct ADRESSE { .... } CORNELIA ; hat man ein Objekt vom Typ Struktur ADRESSE erklärt. CORNELIA hat 4 Elemente und eine Länge von 42 Byte. CORNELIA.POSTLZ bezeichnet das erste Element von CORNELIA und kann genauso verwandt werden wie eine Variable vom Typ int. CORNELIA.ORT ist ein Feld aus Zeichen, von denen CORNELIA.ORT[0] das erste ist.

Besonderheit beim Zugriff über Pointer:

Beispiel

```
struct ARTIKEL {
                  int NR;
                  float PREIS;
                  char BEZEICH[30];
                  int LAGER;
                } * EINKAUF;
```

EINKAUF ist ein Pointer auf eine Struktur mit Namen ARTIKEL. Statt (*EINKAUF).PREIS ist es ebenso möglich EINKAUF -> PREIS zu schreiben (Pfeil = minus größer). Jedesmal ist das Element PREIS der Struktur, auf die der Pointer EINKAUF zeigt, gemeint.

Eine Struktur kann als Element auch eine weitere Struktur haben.
Es ist möglich, daß eine Struktur einen Pointer auf ein Objekt vom Typ der gerade definierten Struktur enthält. Dadurch ist es möglich verkettete Listen zu erzeugen.
Strukturen mit dem gleichen Namen und Funktionen sind als Elemente einer Struktur nicht zugelassen (wohl aber Pointer darauf).

UNION
Einer Struktur ähnlich ist union. Bei einer Struktur liegen die Speicherplätze der einzelnen Elemente hintereinander. Bei einer union hingegen befinden sich die einzelnen Elemente auf dem gleichen Speicherplatz. Die Länge einer union ist also die Länge ihres längsten Elementes. Der Zugriff auf einzelne Elemente erfolgt genauso wie bei Strukturen.

```
--------------------------
Beispiel
 .....
union A {int I; char *C; float F; } UVAR;
 .....
switch (TYPE)
{
case 1: UVAR.I = 4;
        break;
case 2: UVAR.F = 4.004;
        break;
default:
        UVAR.C = "FEHLER";
 }....
--------------------------
```

II4 INITIALISIERUNG

Bei der Deklaration ist es möglich, den deklarierten Objekten einen Anfangswert zuzuweisen. z.B. int I = 8; allgemein sieht eine Initialisierung so aus:

zu initialisierendes Objekt = AUSDRÜCK
= { INITIALISIERUNGSLISTE }
= { INITIALISIERUNGSLISTE , }

wobei die INITIALISIERUNGSLISTE so aussieht:

AUSDRUCK
INITIALISIERUNGSLISTE, INITIALISIERUNGSLISTE
{ INITIALISIERUNGSLISTE }

Bei der Initialisierung von statischen und von externen Objekten dürfen die Ausdrücke nur konstante Ausdrücke sein bzw. Adressen von zuvor deklarierten Objekten. Zu diesen Adressen darf ein konstanter Ausdruck addiert bzw.von diesen Adressen darf ein konstanter Ausdruck subtrahiert werden. Variablen der Speicherklasse auto können mit beliebigen Ausdrücken (konstanten Ausdrücken, Aufrufen von bereits deklarierten Funktionen oder Variablen) initialisiert werden. Sie werden jedesmal bei Eintritt in den Block, in dem sie deklariert wurden, neu vorbesetzt. Alle anderen Objekte werden nur einmal bei Programmbeginn initialisiert.
Falls Objekte nicht vorbesetzt werden, enthalten sie irgendeinen Anfangswert.
Felder und Strukturen der Speicherklasse auto und union dürfen nicht vorbesetzt werden!
Wenn das vorzubesetzende Objekt ein Feld oder eine Struktur ist, sind die Anfangswerte in {} Klammern einge-

schlossen. Falls weniger Anfangswerte als Feldelemente angegeben werden, werden die restlichen Feldelemente mit 0 vorbesetzt.
Wie man durch besondere Anwendung der Klammerung {} bestimmte Teile eines Feldes vorbesetzen kann, sieht man an den folgenden Beispielen.

Beispiele

a.) static int J[5][4] = { {1,2,3,4} , {11,12,13,14} };

J ist ein zweidimensionales Feld, das aus 5 Elementen besteht. Jedes Element ist ein eindimensionales Feld, das aus 4 Elementen besteht. Jedes dieser Elemente ist vom Typ int. Die beiden ersten der fünf Elemente, aus denen J besteht, sind mit den angegeben Werten vorbesetzt. Der Rest wird mit 0 vorbesetzt. d.h.

```
J[0][0] =  1, J[0][1] =  2, J[0][2] =  3, J[0][3] =  4,
J[1][0] = 11, J[1][1] = 12, J[1][2] = 13, J[1][3] = 14,
J[2][0] =  0,     ...
J[3][0] =  0,     ...
J[4][0] =  0,     ...                         J[4][3] = 0
```

dieselbe Vorbesetzung erreicht man auch durch:

```
static int J[5][4] = { 1, 2, 3, 4, 11, 12, 13, 14 };
```

aber durch

b.) static int J[5][4] = { {1}, {2}, {3}, {4}, {9} };

wird von jedem Element, aus dem J besteht, das erste Element mit den angegebenen Werten vorbesetzt und der Rest mit 0.

```
also: J[0][0] = 1, J[0][1] = 0, ...
      J[1][0] = 2, J[1][1] = 0, ...
      J[2][0] = 3, J[2][1] = 0, ...
      J[3][0] = 4, J[3][1] = 0, ...
      J[4][0] = 9, J[4][1] = 0, ...
```

Allgemein gilt, daß die Klammerung so ausgewertet wird: Wenn die in {} eingeschlossenen Anfangswerte noch weiter mit Hilfe von Klammern {} gruppiert sind, so wird mit den in weiteren Klammern eingeschlossenen, durch Kommata getrennten Anfangswerten zunächst das entsprechende Element des Feldes mit den in Klammern zusammengefaßten Werten vorbesetzt. Falls das Element aus mehr Elementen besteht als Werte angegeben wurden, werden die restlichen Elemente des Elementes mit 0 vorbesetzt. Anschließend wird mit den nächsten in Klammern eingeschlossenen Werten und dem nächsten Feldelement die Initialisierung fortgesetzt. (wie in Beispiel b.))
Falls die in {} Klammern eingeschlossenen Anfangswerte

nicht durch weitere Klammern zusammengefaßt werden, werden der Reihe nach die terminalen Elemente des Feldes mit diesen Werten vorbesetzt. (wie in Beispiel a.))
Allgemein gilt, das keine "überschüsssigen" Werte angegeben werden dürfen.
Ein Feld vom Typ char kann durch einen String vorbesetzt werden.

Beispiel
```
char W[] = "COMPILER";
```

Beispiel
Die Struktur CORNELIA aus II3 könnte so initialisiert werden:
```
struct ADRESSE CORNELIA= {4443,"Schuettorf","Nordhof",6};
```
oder auch {{4443},{"Schuettorf"},{"Nordhof"},{6}}.
Falls bei einer Felddeklaration mit Initialisierung die Angabe der Feldlänge unterbleibt, wird ein Feld definiert, das genauso viele Elemente hat, wie in der Initialisierung Werte vorhanden sind.

Beispiel
```
static int K[] = { 3, 6, 9 };
```
K ist ein Feld mit drei Elementen vom Typ int.

III AUSDRUCKE

III1 PRIMÄRE AUSDRUCKE

Primäre Ausdrücke sind:

	Beispiele:
a Name, der nach den Konventionen in II deklariert wurde	nach int VA; VA
b Konstante	6, 'S', '0'
c String	"TEXT"
d (Ausdruck)	(5+6), ('H')
e Primärer Ausdruck[Ausdruck] (Feldelement)	nach char P[10]; P[2+4]
f Primärer Ausdruck (Ausdruck-liste) Ausdruckliste kann entfallen (Funktionsaufruf)	MAX(A,B)
g Primärer Lvalue.Name	CORNELIA.STR
h Primärer Ausdruck -> Name (Element einer Stuktur oder Union)	EINKAUF->PREIS

Eine Ausdruckliste kann aus einem oder mehreren Ausdrücken bestehen, die durch Kommata getrennt sind.
Ein Lvalue ist ein Ausdruck, der sich auf ein Objekt bezieht. Z.B. der Name einer Variablen oder ein Ausdruck der Form *Pointer. Faustregel:
Ein Lvalue ist etwas, das links von einem Gleichheitszeichen stehen kann, dem man also etwas zuweisen kann. Die Zahl 3 ist kein Lvalue, sie hat einen festen Wert, man kann ihr nichts zuweisen.
Unbekannte Namen, denen eine Klammer (folgt, werden automatisch als Namen einer int Funktionen definiert.

III2 AUSDRUCKE MIT OPERATOREN

a Monadische Operatoren

Monadische Operatoren sind solche, die auf nur ein Element wirken. Die hier aufgelisteten Operatoren haben alle die gleiche Priorität. Die Abarbeitung erfolgt von rechts nach links. (d.h. bei - ~ A; wird zuerst ~ und dann - ausgeführt)

! AUSDRUCK — logisches Nicht
Das Ergebnis von ! AUSDRUCK ist 1, falls AUSDRUCK = 0 ist, sonst 0. Der Typ ist int.

~ AUSDRUCK	Einerkomplement AUSDRUCK muß ganzzahlig sein und wird bitweise komplementiert ~ FFF9 = 0006
- AUSDRUCK	Minus AUSDRUCK wird negiert. VA = -3;
-- LVALUE ++ LVALUE LVALUE -- LVALUE ++	LVALUE wird um 1 erhöht, bzw. um 1 erniedrigt. Welche Auswirkungen es hat, wenn ++ bzw. -- vor oder nach LVALUE steht, sieht man an folgendem Beispiel: X = 5; Y = --X; erst wird der Wert von X um 1 erniedrigt und dann Y zugewiesen. Wert von X = 4 und Wert von Y = 4. X = 5; Y = X-- ; erst wird Y der Wert von X zugewiesen und dann wird X um 1 erniedrigt. Wert von X = 4 und Wert von Y = 5.
& LVALUE	& LVALUE ist Pointer auf LVALUE d.h. & LVALUE ist die Adresse von LVALUE z.B. int A; dann ist &A ein Pointer auf die int Variable A.
* AUSDRUCK	AUSDRUCK muß ein Pointer sein. * AUSDRUCK bezeichnet das Objekt, auf das der Pointer AUSDRUCK zeigt, und ist Lvalue. z.B. int B; * & B = 5; entspricht: B = 5; char F[10]; F[4] entspricht * (F + 4)
sizeof AUSDRUCK	sizeof gibt die Länge von AUSDRUCK in Byte an und ist eine Konstante vom Typ int. z.B. int A; Der Wert von sizeof A beträgt 2. double F[4]; Der Wert von sizeof F beträgt 32.
sizeof (TYP)	sizeof gibt die Länge von TYP in Byte an und ist eine Konstante vom Typ int. z.B.

sizeof (struct ZIC{int *A; char B[7];})
hat den Wert 9. (einmal 2 Byte und 7 mal 1 Byte)

(TYPNAME) AUSDRUCK wird CAST (Typumwandlung) genannt.
z.B. int VA;
sin((float) VA); sin wird aufgerufen mit dem nach float gewandelten Wert von VA

b Dyadische Operatoren

Dyadische Operatoren sind solche, die auf zwei Elemente wirken.
Die folgenden drei Operatoren nennt man multiplikative Operatoren. Sie besitzen die gleiche Priorität. Ihre Abarbeitung erfolgt von links nach rechts. (A/B*C zuerst wird A/B ermittelt und dann das Ergebnis mit C multipliziert.)

AUSDRUCK * AUSDRUCK Multiplikation
AUSDRUCK / AUSDRUCK Division
AUSDRUCK % AUSDRUCK Modulo (z.B. 5 % 3 = 2 Rest bei Division)

Die beiden folgenden Operatoren sind additive Operatoren. Sie haben die gleiche Priorität. Ihre Abarbeitung erfolgt von links nach rechts.

AUSDRUCK + AUSDRUCK Summe
AUSDRUCK - AUSDRUCK Differenz

Pointer werden bei der Addition und Subtraktion gesondert behandelt. Wird zu einem Pointer eine ganze Zahl addiert oder von einem Pointer eine ganze Zahl subtrahiert, so hat der gesamte Ausdruck den gleichen Typ wie der Pointer. Die ganze Zahl wird in beiden Fällen so skaliert, daß der Pointer um soviele Element weiter zeigt, wie die ganze Zahl angibt.

Beispiel:

```
long P[6], *POI;
POI = P;
```

POI + 4 zeigt dann auf das 4. Element im Feld P, also wurden 16 Byte zur Adresse in POI addiert.

---

also (Pointer + 1) zeigt nicht auf eine Stelle die ein Byte höher liegt, sondern auf das nächste Objekt des Feldes.
Falls zwei Pointer POI1 und POI2 auf Objekte vom gleichen Typ zeigen, gibt POI1 - POI2 die Anzahl der Objekte

(gleichen Typs) an, die zwischen den beiden Pointern stehen.

Die folgenden zwei Operatoren heißen shift (schiebe) Operatoren. Sie haben die gleiche Priorität. Ihre Abarbeitung erfolgt von links nach rechts.

AUSDRUCK1 >> AUSDRUCK2
AUSDRUCK1 << AUSDRUCK2

Der Wert der beiden beteiligten Ausdrücke muß ganzzahlig sein. AUSDRUCK2 wird in int gewandelt. Das Ergebnis ist vom gleichen Typ wie AUSDRUCK1.
Beispiel: 9 << 3 = 72
(0000 0000 0000 1001 << 3) = (0000 0000 0100 1000)
Beim nach links Shiften wird 0 nachgerückt. Beim nach rechts Shiften wird, falls das Vorzeichen gesetzt ist, 1 sonst 0 nachgerückt. Beim Shiften eines Ausdrucks vom Typ unsigned wird in beiden Fällen eine 0 nachgerückt.

Die folgenden vier Operatoren nennt man Vergleichsoperatoren. Sie haben die gleiche Priorität. Ihre Abarbeitung erfolgt von links nach rechts.

AUSDRUCK < AUSDRUCK
AUSDRUCK > AUSDRUCK
AUSDRUCK <= AUSDRUCK
AUSDRUCK >= AUSDRUCK

Der Wert eines solchen Vergleiches ist 0, falls der Vergleich falsch ist, 1 sonst. Der Typ des Ergebnisses ist int.
Beispiel:
Der Wert von 4 > 4 ist 0
Der Wert von 4+1 > 4 ist 1
Der Wert von 3 < 2 < 5 ist 1 !!!
(Wert von 3 < 2 ist 0, Wert von 0 < 5 ist 1)
Auch Vergleiche zwischen Pointern sind möglich.

Die beiden folgenden Operatoren heißen Gleichheitsoperatoren.

AUSDRUCK == AUSDRUCK
AUSDRUCK != AUSDRUCK

Falls beide Ausdrücke gleich sind, hat AUSDRUCK == AUSDRUCK den Wert 1 und AUSDRUCK != AUSDRUCK den Wert 0. Das Ergebnis hat den Typ int.
Beispiel: 5 != 5 hat den Wert 0
10 > 8 == -1 < 2 hat den Wert 1

Es folgen 3 bitweise Operatoren, die nach absteigender Priorität aufgelistet sind.

AUSDRUCK & AUSDRUCK Bitweises UND

Die beteiligten Ausdrücke müssen ganzzahlige Werte haben. Die Abarbeitung erfolgt von links nach rechts.

Beispiel: 13 & 7 = 5

(0000 0000 0000 1101) & (0000 0000 0000 0111) = (0000 0000 0000 0101)

AUSDRUCK ^ AUSDRUCK Bitweises exklusives ODER

Die beteiligten Ausdrücke müssen ganzzahlige Werte haben.

Beispiel: 13 ^ 7 = 10

(0000 0000 0000 1101) ^ (0000 0000 0000 0111) = (0000 0000 0000 1010)

AUSDRUCK | AUSDRUCK Bitweises ODER

Die beteiligten Ausdrücke müssen ganzzahlige Werte haben.

Beispiel: 13 | 7 = 15

(0000 0000 0000 1101) | (0000 0000 0000 0111) = (0000 0000 0000 1111)

Die beiden folgenden logischen Operatoren sind ebenfalls nach absteigender Priorität aufgeführt. Beide werden von links nach rechts abgearbeitet.

AUSDRUCK && AUSDRUCK logisches UND

AUSDRUCK1 && AUSDRUCK2 hat den Wert 1, falls AUSDRUCK1 und AUSDRUCK2 beide ungleich 0 sind. Falls AUSDRUCK1 gleich 0 ist, wird AUSDRUCK2 nicht mehr ermittelt, sondern gleich der Wert 0 zurückgeliefert.

Beispiel:

Bei A && PUTCHAR(A) wird PUTCHAR(A) nur aufgerufen, wenn A ungleich 0 ist.

AUSDRUCK || AUSDRUCK logisches ODER

AUSDRUCK1 || AUSDRUCK2 hat den Wert 1, falls wenigstens einer der beiden Ausdrücke AUSDRUCK1, AUSDRUCK2 ungleich 0 ist. Sonst 0. Falls AUSDRUCK1 ungleich 0 ist, wird AUSDRUCK2 nicht mehr ermittelt und der Wert 1 zurückgeliefert.

In beiden Fällen (&& und ||) können die Ausdrücke verschiedenen Typ haben. Das Ergebnis hat den Typ int.

AUSDRUCK ? AUSDRUCK : AUSDRUCK bedingter Operator

Der Wert von AUSDRUCK ? AUSDRUCK1 : AUSDRUCK2 ist AUSDRUCK1, falls AUSDRUCK ungleich 0 ist, sonst AUSDRUCK2

Beispiel: A = B > C ? C : B;

A wird das Minimum von C und B zugewiesen.

```
A = A < 0 ? -A : A
```

falls A < 0 ist, wird A negiert.
Falls keiner der beiden Ausdrücke AUSDRUCK1, AUSDRUCK2 ein Pointer ist, wird nach den Regeln aus Kapitel II2 zu dem gemeinsamen Typ gewandelt. Wenn einer der Ausdrücke AUSDRUCK1, AUSDRUCK2 ein Pointer ist, muß der andere ein Pointer vom gleichen Typ sein oder die Konstante 0.

Zuweisungsoperatoren:	Beispiele:
LVALUE = AUSDRUCK	A = B, Wert von A = B ist B
LVALUE += AUSDRUCK	A += F; entspricht A = A + F;
LVALUE -= AUSDRUCK	A -= 9; entspricht A = A - 9;
LVALUE *= AUSDRUCK	A *= 3+D; entspricht A = A * (3+D);
LVALUE /= AUSDRUCK	A /= G(); entspricht A = A / G();
LVALUE %= AUSDRUCK	A %= 4-H; entspricht A = A % (4-H);
LVALUE >>= AUSDRUCK	A >>= 6; entspricht A = A >> 6;
LVALUE <<= AUSDRUCK	A <<= 3; entspricht A = A << 3;
LVALUE &= AUSDRUCK	A &= 2; entspricht A = A & 2;
LVALUE ^= AUSDRUCK	A ^= B; entspricht A = A ^ B;
LVALUE |= AUSDRUCK	A |= C; entspricht A = A | C;

Der Operator mit der niedrigsten Priorität ist der Komma Operator. Seine Abarbeitung erfolgt von links nach rechts.

AUSDRUCK1 , AUSDRUCK2

Zunächst wird der Wert des Ausdrucks AUSDRUCK1 ermittelt, anschließend der von AUSDRUCK2. Der Wert und Typ von AUSDRUCK1,AUSDRUCK2 ist der Wert und Typ von AUSDRUCK2.

Falls der Kommaoperator innerhalb einer Liste von Funktionsargumenten oder als Feldindex auftritt, also an Stellen, an denen das Komma bereits eine andere Bedeutung hat, müssen Klammern gesetzt werden.

Beispiel

```
for ( i=j=0 ; j <= i || (j=0, ++i < max) ; ++j )
   M1[i][j] = M2[i][j];
```

kopiert das untere Dreieck einschließlich der Diagonalen von Matrix M2 nach M1.

Zusammenfassend eine Tabelle mit allen Operatoren, nach absteigender Priorität geordnet. Operatoren mit gleicher Priorität stehen in einer Zeile.

```
()    []    .    ->
*     &     -    !     ~     ++     --     sizeof    (typname)
*     /     %
+     -
>>    <<
<     >     <=   >=
==    !=
&
^
|
&&
||
? :
=    +=    -=    *=    /=    %=    >>=   <<=   &=    ^=    |=
,
```

III3 KONSTANTE AUSDRÜCKE

Konstante Ausdrücke sind Ausdrücke, deren Wert eine Konstante ist.
An bestimmten Stellen müssen in einem C Programm konstante Ausdrücke stehen:

a bei case C: muß C ein solcher Ausdruck sein.
b bei der Angabe von Feldlängen z.B. char NAME [K], muß die Feldlänge (im Beispiel K) ein konstanter Ausdruck sein.
c nach #if muß ein konstanter Ausdruck stehen
d bei der Initialisierung z.B. bei int A = F muß F ein konstanter Ausdruck sein.

In den Fällen a, b und c sind die folgenden konstanten Ausdrücke erlaubt:

ganzzahlige Konstanten vom Typ int
Zeichenkonstanten
sizeof Ausdrücke

Die genannten konstanten Ausdrücke dürfen noch mit den folgenden Operatoren verknüpft sein:
den monadischen Operatoren - ~
den dyadischen Operatoren + - * / % & | ^ << >>
und dem bedingten Operator ? :

Im Fall d ist außer den für a, b und c angegebenen konstanten Ausdrücken noch Folgendes zulässig:
Der monadische Operator & verknüpft mit einem schon deklarierten statischen oder externen Objekt, also dessen Adresse . Zu dieser Adresse darf eine Konstante addiert bzw. von ihr subtrahiert werden.

Beispiel
z.B. nach static float FF;
ist &FF + 45 bei der Initialisierung erlaubt
(aus + 45 wird, da FF vom Typ float ist + 4 * 45)

Zusätzlich gilt bei MI - C: In den Fällen a und d sind ebenfalls Konstanten vom Typ long und double erlaubt.

IV. ANWEISUNGEN

IV1 if - else Anweisung

Diese bedingte Anweisung sieht allgemein so aus:

```
if ( AUSDRUCK )
   ANWEISUNG1
else
   ANWEISUNG2
```

Der Teil else ANWEISUNG2 kann auch entfallen. Zunächst wird der Wert von AUSDRUCK ermittelt. Falls dieser Wert ungleich 0 ist, wird die Anweisung ANWEISUNG1 ausgeführt. Falls der Wert von AUSDRUCK 0 ist, wird ANWEISUNG1 übersprungen und ANWEISUNG2 ausgeführt oder falls else ANWEISUNG2 nicht vorhanden ist, wird nur ANWEISUNG1 ignoriert und hinter ANWEISUNG1 der Programmablauf fortgesetzt.

Beispiel

```
if ( 0 )
    PRINTF("NULL");
else
    PRINTF("EINS");
```

EINS wird ausgedruckt.

```
if ( VA == VB )
   VC = VA + VB;
else
   VC = 100;
```

wenn VA gleich VB ist, wird VC der Wert von VA+VB zugewiesen wenn VA ungleich VB ist, wird VC der Wert 100 zugewiesen

```
if (X = Y)
  Z = 2 * X;
else
  Z = 0;
```

zuerst wird X der Wert von Y zugewiesen, wenn Y und damit X gleich 0 ist, wird Z 0 zugewiesen. Sonst 2 * X.

Die if - else Anweisung kann auch geschachtelt werden.

Beispiel

```
if ( VARA + VARB )
  {
  if ( VARA + VARB > 0 )
     VARC = VARA + VARB:
  else
     C = - ( A + B );
  }
```

```
else
  C = 100;
```

Bei geschachtelten if - else Anweisungen ist zu beachten, daß sich else stets auf das letzte if, auf das noch kein else gefolgt ist, bezieht. Nur durch Klammerung { } kann man erreichen, daß sich jedes else auch auf das gewünschte if bezieht.

Beispiel

zwischen

```
a.) if ( X + Y )

       if ( X + Y ) 0 )
          Z = X + Y;

    else
        Z = 100;
```

und

```
b.) if ( X + Y )
       {
         if ( X + Y ) 0 )
            Z = X + Y;
       }
    else
       Z = 100;
```

ist ein großer Unterschied. In a.) wird das else dem innersten if zugeordnet.

falls X+Y) 0 ist, wird Z der Wert von X+Y zugewiesen
falls X+Y = 0 ist, wird Z nicht verändert.
falls X+Y (0 ist, wird Z der Wert 100 zugewiesen.

In b.) wird das else dem äußeren if zugeordnet.

falls X+Y) 0 ist, wird Z der Wert von X+Y zugewiesen
falls X+Y = 0 ist, wird Z der Wert 100 zugewiesen
falls X+Y (0 ist, wird Z nicht verändert.

Eine weitere mögliche Schachtelung ist:

```
if (AUSDRUCK1)
  ANWEISUNG1
else
  if (AUSDRUCK2)
    ANWEISUNG2
else
  if (AUSDRUCK3)
    ANWEISUNG3
 . . .
else
  ANWEISUNGi
```

Durch diese Art der Schachtelung, hat man die Möglichkeit mehr als nur 2 Fälle zu unterscheiden. (Im folgenden Beispiel 4)

```
if ( ! C )
  C = 100;
else   if ( C ) 100)
```

```
            C *= 2;
else    if ( C < 100 && C > 0 )
            C *= 3;
else
  C = -C;
```

Die vier Fälle C < 0, C = 0, 0 < C < 100, C > 100 werden unterschieden.

IV2 switch Anweisung

Allgemein sieht die switch Anweisung so aus:

```
switch (AUSDRUCK)
  case konstanter AUSDRUCK1: ANWEISUNGSFOLGE1
  case konstanter AUSDRUCK2: ANWEISUNGSFOLGE2
            . . .
  default:                   ANWEISUNGSFOLGEi
```

default ANWEISUNGSFOLGEi kann entfallen.
Ausdruck muß bei MI - C nicht vom Typ int sein. Die konstanten Ausdrücke AUSDRUCK1, AUSDRUCK2, ... müssen alle verschieden sein, und gleichen Typ wie AUSDRUCK haben! (zulässige Ausdrücke siehe III3)

Bei der switch Anweisung wird zunächst der Wert von AUSDRUCK ermittelt. Er wird mit den nach jedem case folgenden konstanten Ausdruck verglichen. Sobald eine Übereinstimmung festgestellt wird, wird mit der Anweisungsfolge, die dem case, bei dem die Übereinstimmung festgestellt wurde, folgt, fortgefahren und die switch Anweisung nach break verlassen. Falls kein break gesetzt ist, werden die Anweisungen nach jedem der folgenden case ebenfalls ausgeführt (bis break oder return gefunden wird). Falls mit keinem case Übereinstimmung festgestellt wurde, wird die Anweisung nach default ausgeführt oder falls kein default vorhanden ist, wird das Programm nach der Endeklammer } von switch fortgesetzt.

Beispiel

```
MAIN()
{
int I;
I = GCH();
switch (I)
  {
  case '1':   FUNK1();
              break;
  case '2':   FUNK2();
  case '3':   FUNK3();
  case '4':   FUNK4();
              break;
  default:    PRINTF ("FALSCHE EINGABE")
  }
}
```

Falls das Eingabezeichen eine 1 ist, wird die Funktion FUNK1 ausgeführt und die switch Anweisung verlassen. Falls das Eingabezeichen eine 2 ist, werden die Funktionen FUNK2, FUNK3 und FUNK4 ausgeführt und dann die switch Anweisung verlassen. Falls das Eingabezeichen eine 3 ist, werden FUNK3 und FUNK4 ausgeführt und die switch Anweisung verlassen und falls das Eingabezeichen eine 4 ist, wird nur FUNK4 ausgeführt und dann die switch Anweisung verlassen. Falls das Eingabezeichen ungleich 1, 2, 3 oder 4 ist, wird FALSCHE EINGABE ausgedruckt.

IV3 while Anweisung

Allgemein sieht eine while Anweisung so aus:

```
while (AUSDRUCK)
       ANWEISUNG
```

Zunächst wird der Wert von AUSDRUCK ermittelt. Falls dieser Wert ungleich 0 ist, wird ANWEISUNG ausgeführt. ANWEISUNG wird so oft ausgeführt, bis AUSDRUCK gleich 0 ist, dann wird hinter ANWEISUNG fortgefahren.

Beispiel

```
while (1)          ist Endlosschleife (kann nur mit break
{ ... }            return oder goto verlassen werden)
```

```
A = 1;
while ( A <= 25 )
  {
  B = A * A;
  PRINTF ("\R\NDAS QUADRAT VON %D IST %D ",A, B);
  ++A;
  }
```

Die Quadratzahlen von 1 bis 25 werden ausgedruckt.

Mit break (ebenso mit goto bzw. return) kann eine while Anweisung vorzeitig verlassen werden.

Beispiel

```
A = 1;
 while ( A <= 25 )
  {
  B = A * A;
  if ( B > 400)
       break;
  PRINTF ("\R\NDAS QUADRAT VON %D IST %D ",A, B);
  ++A;
  }
```

Nur noch die Quadratzahlen von 1 bis 20 werden ausgedruckt.

IV4 for Anweisung

Die for Anweisung hat die Form:

```
for (AUSDRUCK1; AUSDRUCK2; AUSDRUCK3)
       ANWEISUNG
```

Die for Anweisung ist äquivalent zu:

```
  AUSDRUCK1;
  while (AUSDRUCK2)
       {
       ANWEISUNG
       AUSDRUCK3;
       }
```

In der for Anweisung können die Ausdrücke AUSDRUCK1,... entfallen, aber die Semikolon müssen hingeschrieben werden! Falls der Test (= AUSDRUCK2) fehlt, wird angenommen er sei ungleich 0. Mit for (AUSDRUCK1;;AUSDRUCK3) erhält man eine Endlosschleife. Das Fehlen der Ausdrücke AUSDRUCK1 und AUSDRUCK3 hat keine Folgen. Eine for Anweisung kann ebenfalls durch break, goto oder return vorzeitig beendet werden.

```
--------------------------
Beispiel
for  (I = 0 ;  I < 10 ;  ++I)        die ersten 10 Elemente
     FEL[I] = 9;                     von FEL werden mit 9
                                     besetzt
das Gleiche erreicht man durch:
I = 0;
for ( ; I < 10 ; )
     FEL[ I++ ] = 9;
--------------------------
```

IV5 do - while Anweisung

Allgemein hat die do - while Anweisung die folgende Form:

```
do
  ANWEISUNG
while (AUSDRUCK);
```

Zunächst wird ANWEISUNG ausgeführt, dann wird ermittelt, ob AUSDRUCK ungleich 0 ist. Falls AUSDRUCK ungleich 0 ist, wird wieder ANWEISUNG ausgeführt und wieder ermittelt, ob AUSDRUCK gleich 0 ist. usw. Falls AUSDRUCK gleich 0 ist, ist die Schleife beendet und das Programm wird hinter AUSDRUCK fortgesetzt.

```
--------------------------
Beispiel
I = 0;
do
  { J = I * I * I;
    PRINTF("\N\R%10D hoch 3 ist %D ",I, J)
    ++I;
  }
while (I < 10)
--------------------------
```

0 hoch 3 bis 9 hoch 3 wird berechnet und ausgedruckt.

Die do - while Anweisung kann durch return, goto oder break vorzeitig beendet werden.

IV6 break Anweisung

Mit break können die for, do - while, switch und while Anweisungen vorzeitig abgebrochen werden. Das Programm wird hinter der for, do - while, switch oder while Anweisung fortgesetzt.

IV7 continue Anweisung

Die continue Anweisung bewirkt, das innerhalb von while, do - while und for Schleifen ans Schleifenende gesprungen wird.

Beispiel

```
while ( A >= 100 )
  {
  B += 100;
  if ( B < 1000 )
       continue;
  FELD[I++] -= B;
  A -= 10;
  }
```

Falls A>=100 ist, wird B solange wie B<1000 ist, um 100 erhöht und der Schleifenrest übersprungen. Erst, wenn B>=1000 ist wird auch der Rest der Schleife abgearbeitet, bis die Bedingung A>=100 nicht mehr erfüllt ist.

Allgemein ist die Anweisung continue bei den verschiedenen Schleifen äquivalent zu goto WEITER:

```
while (...)          do              for(..;..;..)
{                    {               {
  ...                  ...             ...
WEITER:  ;           WEITER:  ;      WEITER:  ;
}                    }               }
                     while (...);
```

IV8 return Anweisung

Die allgemeine Form der return Anweisung ist eine der beiden folgenden:

```
return;
return   AUSDRUCK;
```

Im ersten Fall ist der zurückgegebene Wert nicht definiert, im zweiten Fall wird der Wert von Ausdruck zurückgegeben.
Mit return wird eine aufgerufene Funktion verlassen und zur aufrufenden Funktion zurückgekehrt. Eine Funktion, die kein return enthält, wird bis zu ihrer Endeklammer } abgearbeitet und dann automatisch (ohne einen Wert zurückzugegeben) verlassen.

Beispiel

```
f()
{ int A1, A2 A3;
  A1 = MAX2(A2,A3);
  ..... }
```

```
MAX2(A,B)
int A, B;
{
int C;
C = A > B ? A : B;
return C *= 2;
}
```

IV9 label Anweisung

Allgemeine Form der label Anweisung:

NAME:

NAME ist ein Name nach den Konventionen in I1. NAME: darf nur vor einer Anweiung stehen.. Gebraucht werden label Anweisungen zur Erstellung von Sprungzielen für goto Anweisungen, die in der gleichen Funktion stehen müssen.

IV10 goto Anweisung

Die goto Anweisung hat die allgemeine Form:

goto NAME;

NAME muß eine Sprungmarke sein (siehe label Anweisung), die in der gleichen Funktion wie goto steht. goto NAME bewirkt einen Sprung von der goto Anweisung zu der Anweisung, die hinter der Marke NAME steht.
Man kann aus Schleifen oder Blöcken heraus- bzw. in sie hineinspringen. Wenn in Blöcke hineingesprungen wird, enthalten alle dort definierten temporären Variablen einen beliebigen Wert.
Die goto Anweisung sollte mit Vorsicht verwandt werden, da bei hemmungslosem Gebrauch von goto ein Programm unübersichtlich wird und nur schwer oder gar nicht zu warten ist.

Beispiel
Falls man verschachtelte Schleifen verlassen will, ist es mit break nur möglich, die innerste Schleife zu verlassen.

```
while ( ... )
  while ( ... )
       {
       if ( EINGABEFEHLER )
            goto FEHLEING;
       ...
       }
  ...
FEHLEING:  ...
```

IV11 Leere Anweisung

Sie hat die Form:

```
;
```

Beispiel
```
while ( I++ < 30 ) ;
```
I wird solange um 1 erhöht, bis I >= 30 ist.

IV12 Blockanweisung

Oberall dort, wo eine der obigen Anweisungen stehen kann, kann auch eine in { ... } eingeschlossene Folge von Anweisungen stehen. Vor die erste Anweisung können Deklarationen gesetzt werden. Die deklarierten Objekte haben nur innerhalb des Blockes ihre Gültigkeit.

IV13 Ausdruck - Anweisung

Ein Ausdruck gefolgt von einem Semikolon ist eine Ausdruck - Anweisung. Im Normalfall ist dies ein Funktionsaufruf oder eine Zuweisung.

V. EXTERNE DEFINITIONEN

extern = außerhalb von Funktionen
Ein C Programm besteht aus einer Folge von externen Defitionen, genauer von Definitionen von externen Objekten, die hier in Funktionen und andere Objekte, Daten genannt, eingeteilt werden sollen.
Extern definierte Daten und Funktionen können beliebigen Typ haben. Sie müssen aber die Speicherklasse extern oder static haben. Extern definierte Daten und Funktionen sind bis zum Dateiende bekannt.

V1 EXTERNE DATENDEFINITIONEN

Eine externe Datendefinition ist eine Deklaration siehe Abschnitt II. Die Speicherklasse einer solchen Deklaration darf nur extern oder static sein.

V2 EXTERNE FUNKTIONSDEFINITIONEN

Die allgemeine Form einer Funktionsdefinition sieht so aus:

```
  Deklaration Funktionsname (evtl. Parameterliste)
       Parameterdeklaration (falls Parameter vorhanden)
       {
        Funktionskörper (= Folge von Deklarationen
                           lokaler Variablen gefolgt
                           von Anweisungen)
       }
-------------------------
```

Beispiel:

```
       long MOD2(X,Y)
           long X, Y;
           {
            long Z;
            Z = ( X > Y ) ? (X) % (Y) : (Y) % (X) ;
            return  2*Z ;
           }
```

long vor MOD2(X,Y) gibt an, daß die Funktion MOD2(X,Y) einen Wert vom Typ long zurückliefert. hier: 2*Z
Falls kein Typ angegeben wird, wird immer angenommen, daß die Funktion einen Wert vom Typ int zurückliefert. Außer bei Funktionen, die Werte vom Typ int zurückliefern, muß

vor dem ersten Funktionsaufruf die Funktionsdefinition erfolgen.

```
Falsch:              Richtig:              Richtig:
MAIN()             long MFUNK()          MAIN()
{.. X = MFUNK();}  { ... }                 extern long MFUNK()
long MFUNK()       MAIN()                  {.. X = MFUNK();}
{ ... }            {.. X = MFUNK();}
```

Wichtig! Eine Funktion kann keine STRUCT, UNION, Felder oder Funktionen zurückliefern, nur Pointer auf solche.
X und Y sind Parameter, long X, Y; ist die Parameterdeklaration. Wichtig! In der Parameterliste dürfen keine struct, union, Funktionen oder Felder stehen, nur Pointer auf solche sind erlaubt.
Z ist eine lokale Variable vom Typ long.
Der Funktionskörper muß in { } eingeschlossen sein.

Beim Aufruf einer Funktion werden Parameter vom Typ char nach int gewandelt und solche vom Typ float nach double.

VI DIE FUNKTION MAIN

Das von anderen Programmiersprachen her bekannte Hauptprogramm ist eine Funktion mit dem Namen MAIN. Sie wird automatisch beim Start des Programms aufgerufen.
MAIN kann ohne Parameter oder mit 2 Parametern definiert werden. Mit Hilfe der Parameter kann auf den Text hinter dem Programmnamen in der Aufrufzeile zugegriffen werden.
Definiert wird folgendermassen:

```
MAIN.(argc,argv) int argc; char *argv[];
```

argc gibt die Anzahl der Argumente +1 an. argc ist ein Pointer auf ein Feld von Strings. Jedes Element von argv ist ein Pointer auf die zugehörige Zeichenfolge aus der Aufrufzeile. Der erste Pointer zeigt aus Kompatibilitätsgründen immer auf einen leeren String.
z.B.: Die Aufrufzeile ist "A>cc /c beispiel'cr'"
argc ist gleich 3. argv[1] zeigt auf "/c" und argv[2] zeigt auf "beispiel".

VII ANWEISUNGEN AN DEN PREPROCESSOR

Der C Compiler enthält einen Preprocessor, der in der Lage ist, Makros zu ersetzen, bedingte Compilation zu veranlassen und Dateien und Assemblertexte einzufügen. Wichtig! Jede Zeile, die den Preprocessor ansprechen soll muß als erstes Zeichen ein # enthalten. Außer der Mitteilung an den Preprocessor darf die Zeile keine weiteren Zeichen enthalten (werden überlesen!)

VII1 ERSETZUNGEN

Eine Anweisung der Form

#define NAME Ersetzungszeichen

bewirkt, daß überall im nachfolgenden Programm, der Name NAME durch die Zeichenfolge Ersetzungszeichen ersetzt wird.

Beispiel

```
#define   FLAENGE  80
   ...
char FELD [FLAENGE];
```

FELD ist ein Feld der Länge 80 vom Typ char.

Es gibt ebenfalls die Möglichkeit Makros mit Parametern zu erklären.

#define Name(Name,Name,...., Name) Ersetzungszeichen

Wichtig! Name und (müssen unmittelbar aufeinander folgen. Falls zwischen Name und (z.B. ein Leerzeichen steht, wird Name durch (Name, Name, ..., Name) Ersetzungszeichen ersetzt!

Beispiel

```
#define   SUMME(L, M, K)   L + M - ( K )
   ...
aus:    WERT = 3 * SUMME(3 - F, 75, 100 - D);
wird:   WERT = 3 * 3 - F + 75 - ( 100 - D );
```

Durch #undef NAME gilt der Makroeintrag NAME als nicht mehr vorhanden.

Beispiel

```
#undef SUMME
```

falls anschließend WERT = SUMME (1, 2, 3) auftritt, wird angenommen, SUMME sei eine externe Funktion.

VII2_EINFUGEN_VON_DATEIEN

#include Dateiname
bewirkt eine Ersetzung dieser Zeile durch den Inhalt der Datei Dateiname. #include kann nicht geschachtelt werden. Statt Dateiname kann auch "Dateiname" oder ‹Dateiname› stehen. Der Dateiname darf die Zeichen " ‹ › nicht enthalten.

VII3_BEDINGTE_COMPILATION

Es gibt drei Formen der bedingten Übersetzung:

1.
```
#if          konstanter Ausdruck
 ...
#else
 ...
#endif
```

2.
```
#ifdef     NAME
 ...
#else
 ...
#endif
```

3.
```
#ifndef     NAME
 ...
#else
 ...
#endif
```

... bedeuten eine beliebige Folge von Programmzeilen.
#else kann auch entfallen.

Jedesmal wird überprüft, ob die angegebene Bedingung erfüllt ist.
Bei #if gilt die Bedingung als erfüllt, falls der konstante Ausdruck ungleich 0 ist. Bei #ifdef ist sie erfüllt, wenn der Name NAME durch #define bekannt gemacht wurde und bei #ifndef, falls NAME nicht mittels #define erklärt wurde.
Ist die Bedingung erfüllt, so werden die Zeilen bis zum #else (falls vorhanden, sonst bis #endif) compiliert, anderenfalls werden sie ignoriert und die Zeilen zwischen #else und #endif compiliert.
Es ist zulässig #if... zuschachteln.

Beispiel

```
#define     H    6
   ...
#ifdef H
   Zeile 50
    ...
   Zeile 100
#else
   Zeile 102
    ...
   Zeile 110
#ifdef MAX
   Zeile 112
    ...
   Zeile 130
#endif
   Zeile 132
    ...
   Zeile 160
#endif
```

Die Zeilen 50 bis 100 werden compiliert, die Zeilen 102 bis 160 ignoriert.

VII4 ÄNDERN DER ZEILENNUMMER

#line konstanter Ausdruck

bewirkt, daß für Fehlerverfolgungszwecke der ermittelte Wert als neue Zeilennummer genommen wird.

VII5 EINFUGEN VON ASSEMBLERTEXTEN

```
#asm
  Assemblertext
#endasm
```

Der zwischen #asm und #endasm eingeschlossene Assemblertext wird unverändert in die Ausgabedatei übernommen. Vor #ASM sollte möglichst eine Leeranweisung (;) stehen. Andernfalls kann der Text z.B. nach if an einer unerwünschten Stelle eingefügt werden.

Beispiel
Funktion, die das Byte, das an einem Inputport anliegt, als Funktionswert zurückliefert.

```
INPUT (PORT)
int PORT;
{
#asm
 POP DE
 POP BC
 IN L,(C)
 LD H,0
 PUSH BC
 PUSH DE
#endasm
}
```

Ein weiteres Beispiel:

```
while ( I+5 < K )
{
#asm
  ...
#endasm
   FUNK (I++);
}
```

Der eingefügte Assemblertext und der Funktionsaufruf von FUNK liegen beide innerhalb der while Schleife, weil #asm nicht als Anweisung angesehen werden kann, sondern der nachfolgende Assemblertext unabhängig von der übrigen Codegenerierung unmittelbar in die Ausgabedatei eingefügt wird.

Ohne genaue Kenntnis des Compilers ist das Ergebnis des nächsten Beispiels nicht absehbar:

```
while ( (I + 5) < K
#asm
  ...
#endasm
      )
  FUNK (I++);
```

VIII. REGELN FÜR DEN GELTUNGSBEREICH VON OBJEKTEN

In C hat man die Möglichkeit, den Quelltext eines Programmes in mehreren Dateien zu halten und diese getrennt zu compilieren. Man muß Variablen, die in mehreren Dateien gebraucht werden, in einer Datei so deklarieren, daß ein Zugriff von anderen Dateien aus möglich ist. Man muß sie also außerhalb von Funktionen deklarieren. (externe Variablen)

Interne Variaben haben ihre Gültigkeit solange, wie der Block existiert in dem sie deklariert wurden. Gleichnamige Objekte aus äußeren Blöcken oder gleichnamige externe Objekte werden für die Dauer dieses Blockes überdeckt und sind danach unverändert. Externe Objekte gelten vom Ort ihrer Definition bis zum Dateiende. Externe static deklarierte Objekte sind nur in dem Programmteil bekannt, in dem sie erklärt wurden. Alle anderen externen Objekte sind im ganzen Programm mit Namen bekannt. Es dürfen daher keine zwei unterschiedlichen, globalen Objekte mit gleichem Namen erklärt werden. (auch nicht in verschiedenen Programmteilen) Zu einem Programmteil gehören alle externen Definitionen, die gemeinsam in einem Compilerlauf übersetzt werden.

Beispiel
In den Dateien, in denen die Variable gebraucht aber nicht deklariert wird, muß eine Definition: extern Typ Name; erfolgen.

```
Programmteil 1 :
            int   PARA[JMAX];
            char  *PTRP2;
                  ....
Programmteil 2 :
            extern int  PARA[1];
            extern char *PTRP2;
```

D. BESCHREIBUNG DER BIBLIOTHEKSFUNKTIONEN

Für die Benutzung der folgenden Ein- Ausgabe Funktionen sind die Definitionen aus der Datei STDIO.H erforderlich. (u.a. EOF, NULL, FILE). Sie können durch #include STDIO.H ins Programm eingefügt werden.

I. UNFORMATIERTE EIN- AUSGABE

Folgende Funktionen stehen für die unformatierte Ein-Ausgabe zur Verfügung:

I1 EIN- AUSGABE FÜR DAS TERMINAL

GETCHAR()
: Das nächste Zeichen vom Terminal wird als Funktionswert zurückgegeben.

UNGETCHAR(c)
: Das Zeichen c wird der Funktion getchar für den nächsten Aufruf zur Verfügung gestellt.

PUTCHAR(c)
char c;
: Das Zeichen in c wird auf dem Terminal ausgegeben und als Wert zurückgeliefert.

CHRRDY()
: Der Funktionswert ist 1, falls ein Zeichen vom Terminal ansteht, 0 sonst.

PUTS(ptr)
char *ptr;
: Gibt den String ab ptr auf dem Terminal aus (d.h. bis eine binäre 0 gefunden wird) und dann 'cr''lf'

GETS(buff)
char *buff;
: Eine Eingabezeile wird nach buff gebracht. ('cr' wird nicht übernommen.) Das letzte Zeichen in buff ist eine binäre 0. Der Funktionswert ist die Anzahl der eingegebenen Zeichen.

I2 GEPUFFFERTE DATEIVERARBEITUNG

Bei CP/M Dateien gibt es einige Besonderheiten. Die Länge einer Datei kann nur Vielfache der Recordlänge (128 Byte) annehmen. Eine Ausnahme bilden ASCII-Dateien bei denen ein vorzeitiges Ende durch ein Byte 0X1A angezeigt werden kann. Außerdem wird das Zeichen '\n' durch die Folge '\r''\n' repräsentiert. Solche ASCII-Dateien können unabhängig von dieser speziellen Darstellungsweise bearbeitet werden, wenn beim Eröffnen mit FOPEN angegeben wird, daß es sich um eine ASCII-Datei handelt. Die Zeichen verarbeitenden Funktionen ermitteln dann das tatsächliche Dateiende und '\n' wird in '\r''\n' gewandelt und umgekehrt. Wird bei diesen Funktionen statt der Dateibeschreibung (FILE *) STDIN bzw. STDOUT, STDERR als Parameter angegeben, so wird die Ein- Ausgabe auf das Terminal gelenkt, wobei genauso wie bei einer Datei, die im ASCII-Modus eröffnet wurde, verfahren wird.

```
FILE *FOPEN(dateiname, typ)
char *dateiname, *typ;
```

typ kann sein: "R" zum Lesen
"W" zum Schreiben
"A" zum Anfügen

Zusätzlich zu diesen Grundtypen gibt es bei MI - C folgende zusätzliche Typen:

"RA" zum Lesen im ASCII-Modus
"WA" zum Schreiben im ASCII-Modus
"AA" zum Anfügen an eine ASCII-Datei

"AR" zum Anfügen an eine Datei mit zusätzlicher Leseerlaubnis
"AAR" zum Anfügen an eine ASCII-Datei mit zusätzlicher Leseerlaubnis.

Die Datei dateiname wird eröffnet.
Soll die Datei zum Schreiben eröffnet werden, wird, falls die Datei bereits existiert, die alte Information überschrieben. Existiert noch keine Datei mit dem Namen dateiname, so wird eine neue angelegt.
Soll die Datei zum Lesen eröffnet werden, muß die Datei dateiname bereits existieren. Anders als beim typ "W" oder "A" liefert das Eröffnen einer noch nicht existierenden Datei einen Fehler. Im Fehlerfall wird NULL (Null-Pointer) zurückgegeben.

Falls kein Fehler aufgetreten ist, wird ein Pointer auf die Dateibeschreibung zurückgegeben, der bei getc, putc, fread, fwrite, ungetc, fclose benutzt werden kann.
Die Anzahl der gleichzeitig eröffneten Dateien ist nur durch den Speicherplatz begrenzt.
Die Typen "AR" und "AAR" können nicht bei einer CP/M Version kleiner als 2.0 benutzt werden.
Beispiel: FILE *fp; fp = fopen("B:ZB.C", "R");

FCLOSE(fp)
FILE *fp;
Die zu fp gehörige Datei wird geschlossen und der Puffer freigegeben. Im Fehlerfall wird EOF zurückgegeben, sonst 0.

FPUTS(ptr,fp)
FILE *fp;
Wie PUTS(ptr) nur wird in die zu fp gehörige Datei geschrieben. Bei Fehler oder Dateiende wird EOF als Wert zurückgegeben.

FGETS(buff,max,fp)
char *buff; int max; FILE *fp;
Die nächste Zeile aus der zu fp gehörigen Datei wird nach buff gebracht. Das Zeichen '\n' wird mit übernommen. Eine binäre Null wird ans Ende angefügt. Es werden höchstens max-1 Zeichen gelesen. Im Fehlerfall oder bei Dateiende wird NULL zurückgegeben, sonst buff.

GETC(fp)
FILE *fp;
GETC(fp) gibt das nächste Zeichen aus der zu fp gehörigen Datei zurück oder, falls das Dateiende erreicht ist oder ein Fehler aufgetreten ist, EOF. Falls zu fp eine im ASCII-Modus eröffnete Datei gehört, wird auch beim Auftreten von 0X1A EOF zurückgeliefert.

UNGETC(c,fp)
int c; FILE *fp;
Mit UNGETC(c,fp) wird das Zeichen c in die zu fp gehörige Datei zurückgegeben. Beim nächsten Aufruf von GETC etc. wird dieses Zeichen c verarbeitet. Falls es nicht möglich ist, das Zeichen c in die Datei zurückzuschreiben (z.B. c gleich EOF ist

oder im Fehlerfall) wird EOF zurückgegeben, sonst wird das Zeichen c zurückgegeben. Wenn die Datei eine zusätzliche Schreiberlaubnis hat, kann mittels UNGETC die Datei verändert werden.

```
PUTC(c,fp)
int c;  FILE *fp;
```

Mit PUTC(c,fp) wird das Zeichen c in die zu fp gehörige Datei geschrieben und der Wert c zurückgegeben. Falls das Zeichen nicht in die Datei geschrieben werden konnte (z.B. Platzmangel oder Fehler), wird EOF zurückgegeben.

```
FWRITE(buf,laenge,zahl,fp)
char *buf;  unsigned laenge, zahl;  FILE *fp;
```

zahl oft werden laenge viele Zeichen aus dem Puffer buf in die durch fp gekennzeichnete Datei geschrieben. Falls die gewünschte Zeichenzahl in die Datei geschrieben wurde, wird zahl zurückgegeben. Falls (z.B. durch Platzmangel verursacht) nicht die gewünschte Zeichenzahl in die Datei geschrieben werden konnte, wird die Anzahl der tatsächlich geschriebenen Zeichen in Vielfachen von laenge zurückgegeben. d.h. Ein Fehler ist aufgetreten, falls die Zahl der zu schreibenden Zeichen nicht mit der zurückgegebenen Anzahl übereinstimmt.

```
FREAD(buf,laenge,zahl,fp)
char *buf;  unsigned laenge, zahl;  FILE *fp;
```

zahl oft werden laenge viele Zeichen aus der durch fp gekennzeichneten Datei in den Puffer buf gelesen. Zurückgegeben wird wie oft laenge viele Zeichen aus der Datei gelesen wurden. Im Fehlerfall (z.B. Datei nicht zum Lesen eröffnet) wird 0 zurückgegeben.

```
FSEEK(fp,offset,origin)
FILE *fp;  long offset;  int origin;
```

Durch FSEEK(fp,offset,origin) wird die aktuelle Position in der zu fp gehörigen Datei verändert.
origin kann sein: 0 für Position vom Dateianfang aus um offset verändern
1 für Position von aktueller Position aus um offset verändern

2 für Position vom Dateiende aus um offset verändern

Sollte die ermittelte Position "vor" der Datei liegen, so wird auf Dateianfang positioniert. Beim nächsten Zugriff auf die Datei wird ab der neu ermittelten Position gearbeitet. Im Fehlerfall (z.B. ermittelte Position zu hoch, origin unzulässig) wird EOF zurückgegeben.
FSEEK kann nicht bei einer CP/M Version kleiner als 2.0 benutzt werden.

Beispiel:
Das folgende Programm kopiert eine Datei um, wobei der Quelldateiname und der Zieldateiname am Terminal angefragt werden.

```
#include STDIO.H
main() {
FILE *eunit, *aunit;
char line[80], c;
   PUTS ("\r\neingabedatei :");
   GETS( line);
   if((eunit=FOPEN(line,"R"))==NULL) return error();
   PUTS ("\r\nausgabedatei :");
   GETS (line);
   if((aunit=FOPEN(line,"W"))==NULL) return error();
   while ((c= GETC(eunit)) != EOF)
         if (PUTC (c,aunit) == EOF) return error();
   if (FCLOSE (aunit) == EOF) error();
}
error() { PUTS("\r\nFEHLER BEIM KOPIEREN"); }
```

I3 UNGEPUFFERTE DATEIVERARBEITUNG

Da das CP/M Betriebssystem nur Vielfache von Records (128 Byte) aus Dateien liest oder hinein schreibt, muß bei der ungepufferten Ein- Ausgabe, wenn die Recordgrenzen nicht eingehalten werden, trotzdem zwischengepuffert werden. (z.B. wenn bei READ oder WRITE die Länge einmal kein Vielfaches von 128 ist oder mit LSEEK nicht auf Recordgrenze positioniert wird.) Dadurch wird dann der Zugriff langsamer. Wenn mit READ oder WRITE jedesmal nur wenige Zeichen bei einem Aufruf verarbeitet werden sollen, bietet i. a. die gepufferte Ein- Ausgabe wesentlich kürzere Verarbeitungszeiten.
Die ungepufferte Ein- Ausgabe kann bei einer CP/M Version

kleiner als 2.0 nicht benutzt werden.

```
OPEN(dateiname,typ)
char *dateiname;  int typ;
```

typ kann sein: "0" zum Lesen
"1" zum Schreiben
"2" zum Lesen und Anfügen

Mit OPEN(dateiname,typ) wird die Datei dateiname, die bereits existieren muß, eröffnet. Im Fehlerfall (z.B. typ unzulässig, Datei existiert noch nicht) wird -1 zurückgegeben. Sonst wird die Dateibeschreibung, die eine int ist, zurückgegeben.
z.B. int fd; fd = OPEN ("BSP","2")
Im Fall typ = 1 (schreiben) wird der alte Dateiinhalt überschrieben, falls nicht mittels fseek ans Dateiende positioniert wird.
Die Anzahl der gleichzeitig eröffneten Dateien ist nur durch den Speicherplatz begrenzt.

```
CREAT(dateiname,typ)
char *dateiname;  int typ;
```

typ kann sein: "0" zum Lesen
"1" zum Schreiben
"2" zum Lesen und Schreiben

Mit CREAT(dateiname,typ) wird die Datei Dateiname neu angelegt, falls sie noch nicht existiert. Falls die Datei bereits existiert, wird die alte Information gelöscht.
Im Fehlerfall (kein Platz mehr etc.) wird -1 zurückgegeben, sonst die Dateibeschreibung.

```
CLOSE(fd)
int fd;
```

Die mittels OPEN oder CREAT eröffnete Datei, die durch fd bestimmt wird, wird geschlossen. Der von der Dateibeschreibung benötigte Platz wird wieder freigegeben. Im Fehlerfall wird -1 sonst 0 zurückgegeben.

```
READ(fd,buff,n)
int fd;  char *buff;  int n;
```

Mit READ(fd,buff,n) werden n Zeichen aus der Datei, die durch fd gekennzeichnet ist, in den Puffer buff gelesen. Zurückgegeben wird die Anzahl Zeichen, die gelesen wurde. Falls das Dateiende erreicht wurde, bevor n Zeichen gelesen wurden, wird 0 und im Fehlerfall -1 zurückgegeben.

```
WRITE(fd,buff,n)
int fd; char *buff; int n;
```

Mit WRITE(fd,buff,n) werden n Zeichen aus dem Puffer buff in die Datei, die durch fd gekennzeichnet ist, geschrieben. Zurückgegeben wird die Anzahl der tatsächlich geschrieben Zeichen. Es ist ein Fehler (Datei nicht zum Schreiben geöffnet oder Platzmangel etc.) aufgetreten, falls die vorgegebene Zahl n nicht mit der zurückgegebenen Anzahl übereinstimmt.

```
LSEEK(fd,offset,origin)
int fd; long offset; int origin;
```

Durch LSEEK(fd,offset,origin) wird die aktuelle Position in der zu fd gehörigen Datei verändert.
origin kann sein:
- 0 für Position vom Dateianfang aus um offset verändern
- 1 für Position von aktueller Position aus um offset verändern
- 2 für Position vom Dateiende aus um offset verändern

Sollte die ermittelte Position "vor" der Datei liegen, so wird auf Dateianfang positioniert. Beim nächsten Zugriff auf die Datei wird ab der neu ermittelten Position gearbeitet. Im Fehlerfall (z.B. ermittelte Position zu hoch, origin unzulässig) wird -1 zurückgegeben.

```
ISEEK(fd,offset,origin)
int fd, offset, origin;
```

Beschreibung siehe LSEEK.

```
REWIND(fd)
int fd;
```

In der Datei, die durch fd angegeben ist, wird die aktuelle Position auf den Dateianfang gesetzt.

II. FORMATIERTE EIN- AUSGABE

Die folgenden Funktionen können für die formatierte Ein-Ausgabe verwandt werden. Ihnen ist gemeinsam, daß der erste Parameter ein String ist, aus dem die Anzahl der ein- oder auszugebenden Werte hervorgeht. Außerdem kann die Form der Ein- Ausgabe an diesem String abgelesen werden.

SCANF(CONTROL, ARG1, ARG2, ...)

CONTROL ist ein String, und ARG1, ... sind Pointer auf Objekte, die in CONTROL beschrieben sind. SCANF liest Zeichen vom Terminal, interpretiert sie gemäß CONTROL, und speichert das Ergebnis anschließend an die Stelle, auf die der zugehörige Pointer ARGi zeigt.
Wichtig!! Die Anzahl und der Typ der Parameter muß den Angaben im Controlstring entsprechen. Anderenfalls kann das Programm ohne Vorwarnung zerstört werden.
Die Argumente müssen Pointer sein. !!!
SCANF liefert die Anzahl der erfolgreich weggespeicherten Werte zurück. Bei Dateiende wird EOF statt 0 zurückgeliefert.

Zum CONTROL STRING:

Blanks, Tabs, Newlines werden ignoriert.
Umwandlungsspezifikationen haben folgende Form:
% *(optional) Zahl(optional) Umwandlungszeichen

* bedeutet:	Die Eingabe wird übersprungen, d.h. nicht weggespeichert.
Zahl:	Maximale Länge des Eingabefeldes, das gemäß dem folgenden Format bearbeitet werden soll. Das Eingabefeld besteht aus Zeichen, die verschieden sind von Tab, Newline, Blank. Das Ende des Eingabefeldes wird bestimmt durch Zahl oder durch Blank, Tab oder Newline.

Folgende Umwandlungszeichen gibt es:
D,d Die Eingabe wird als dezimale Zahl vom Typ int inter-
H,h pretiert.
O,o Die Eingabe wird als oktale Zahl ohne führende Null interpretiert. Bei Angabe von Zahl wird diese 0 mitgezählt.

Beispiel
Durch SCANF("%2o %2o",&OKZAHL1, &OKZAHL2); wird bei der Eingabe 34 034 nach OKZAHL1 die Zahl 34 gespeichert und nach OKZAHL2 eine 3.

X,x Die Eingabe wird als hexadezimale Zahl ohne führendes 0X interpretiert. Bei Angabe von Zahl werden 0X mitgezählt.

Das zugehörige Argument sollte für die Fälle: D,d,H,h,O,o,X,x ein int Pointer sein.

C,c Das nächste Eingabezeichen wird als ASCII - Zeichen interpretiert. Das zugehörige Argument sollte ein char Pointer sein. In diesem Fall werden Blanks, Tabs, Newlines nicht überlesen, sondern an die durch das entsprechende Argument angegebene Adresse gespeichert.

S,s Die Eingabe wird als Folge von char (String) aufgefaßt. Das zugehörige Argument sollte auf ein Feld vom Typ char zeigen, das groß genug ist, die char Folge und eine binäre Null (als Endezeichen) aufzunehmen.

Beispiel
char FEL[5];
SCANF("%4S", &FEL); ist bei jeder Eingabe möglich.
Aber bei SCANF("%S", &FEL); mit der Eingabe COMPUTER passiert folgendes: COMPUTER0 wird ab FEL weggespeichert. COMPU wird innerhalb der Feldgrenzen abgelegt, aber TER0 wird über die Feldgrenze hinweg gespeichert und zerstört dort den Speicherinhalt.

F,f Die Eingabe wird als Gleitkommazahl aufgefaßt. Die
E,e Eingabe kann so aussehen: [-]nnnnn.nnnE[-]nnn

Beispiel
float RA;
...
SCANF("%F", &RA);
speichert die folgenden Zahlen nach RA:
100 oder 1.E2 oder 100.0E0 oder 0.0001E6 ...
Folgende Eingabewerte sind unzulässig: .E oder 35.E oder 0.3.4 oder E ...
Zulässig sind Eingabewerte, die höchstens einen Punkt (ein E (e)) enthalten, und falls E (e) vorkommt, danach auch eine Zahl.

Bei SCANF("%5F", &RA); würde die folgende Eingabe eine Fehlermeldung verursachen: 234.E4 (die Zahl nach E wird nicht verarbeitet!)

Falls in den Fällen D,d,H,h,O,o,X,x, dem Umwandlungszeichen ein L oder l vorausgeht, wird angenommen, daß das zugehörige Argument ein Pointer auf long ist. Entsprechend gibt ein L oder l vor F,f,E,e an, daß das Argument ein Pointer auf double statt float ist.

Beispiel
double DOU;
bei SCANF("%E", &DOU); werden nach DOU nur 4 Byte gebracht, die anderen 4 enthalten noch ihren alten Wert, so daß der in DOU enthaltene Wert verschieden ist vom eingegebenen Wert.
Es muß hier also SCANF("%LE", &DOU) heißen.

Falls im CONTROL STRING ein Zeichen auftritt, das mit keinem zulässigen Umwandlungszeichen übereinstimmt, wird der Inhalt des Eingabestroms solange ignoriert, bis er mit diesem Zeichen übereinstimmt. Hinter dem Zeichen wird die Bearbeitung fortgesetzt.

Beispiel

```
long k;
int i,j;
float m;
char feld[10];
SCANF ("%d %2d %*3d %ld Y %4f %s", &i, &j, &k, &m, feld)
Eingabestrom: 10231 1289834567 8910Y8125MI-C
liefert:   10231 -> i,
              12 -> j,
         898 wird  übergangen,
          34567  -> k,
        8910 wird ignoriert,
    Y stimmt mit dem Zeichen im CONTROL STRING überein,
          8125  -> m,
          MI-C0 -> feld.
```

PRINTF(CONTROL, ARG1, ARG2, ...)

CONTROL ist ein String, und ARG1, ... sind die Werte, die auf dem Terminal ausgegeben werden sollen. PRINTF formatiert die Argumente ARG1, ARG2, ... entsprechend den Angaben im CONTROL STRING, bevor sie ausgegeben werden. Alle Zeichen aus CONTROL, die keine gültigen Formatsteuerungen mit vorausgegangenem % sind, werden unverändert mit ausgegeben.

```
Beispiel: int h; h=60;
          PRINTF (" Auf der Wiese sitzen %O Hasen.",h)
```

liefert die Ausgabe: Auf der Wiese sitzen 74 Hasen.

Wichtig!! PRINTF erwartet soviele Argumente, wie aus den Angaben im Controlstring ermittelt werden. Wenn zu wenige Argumente vorkommen, oder wenn der Typ der Argumente mit dem jeweiligen Umwandlungszeichen nicht vereinbar ist, gibt es unsinnige Ergebnisse.

Beispiel

```
long L1;
int I1, I2, I3, I4;
PRINTF ("%e %d %d %d %d", I1, I2, I3, I4, L1);
```

Mit %e werden die acht Byte von I1, I2, I3 und I4 als "double" ausgegeben. Mit %d werden die zwei höherwertigen Byte von L1 als "int" ausgegeben. Mit dem nächsten %d werden die niederwertigen Byte von L1 als "int" ausgegeben. Die nächsten zwei %d holen die nächsten vier Byte aus dem Stack und sie werden als zwei "int" ausgegeben. (Unsinnige Ausgabe, Stack nicht mehr richtig)

Zeichen, die Umwandlung und Ausgabeformat beeinflussen:

%

Jede Umwandlungsbeschreibung beginnt mit %. Falls man das Zeichen % ausgeben möchte, muß man %% schreiben!

-

Das umgewandelte Argument wird linksbündig im Ausgabefeld abgelegt, sonst rechtsbündig. - hat nur Auswirkungen, wenn das Ausgabefeld größer ist als die Ausgabe.

1.Ziffernfolge

Die Zahl gibt die minimale Ausgabefeldlänge an. Ist die Ausgabe länger, wird auch das Ausgabefeld verlängert. Ist die minimale Ausgabefeldlänge größer als die Aus-

gabe, so wird das Ausgabefeld, je nachdem ob '-' gesetzt ist oder nicht, rechts oder links mit ' ' aufgefüllt. Falls die Ziffernfolge mit 0 beginnt, wird das Auffüllzeichen ' ' durch '0' ersetzt.
Angabe % 1.Ziffernfolge .0 und % 1.Ziffernfolge . ohne zweite Ziffernfolge liefern das gleiche Ergebnis)

2.Ziffernfolge
Die Zahl gibt die maximale Anzahl der Zeichen an, die von einer char Folge gedruckt werden sollen, bzw. bei double und float Zahlen die Stellenzahl nach dem Dezimalpunkt.

L, l
L oder l gibt an, daß das zugehörige Argument vom Typ long ist.

Umwandlungszeichen:

D,d Das Argument wird als Dezimalzahl ausgegeben.

O,o Das Argument wird als Oktalzahl ohne Vorzeichen und ohne führende 0 ausgegeben.

X,x Das Argument wird als Hexadezimalzahl ohne Vorzeichen und ohne führendes 0X ausgegeben.

U,u Das Argument wird als Dezimalzahl ohne Vorzeichen ausgegeben.

C,c Das Argument wird als Zeichen vom Typ char aufgefaßt.

S,s Das Argument ist ein String. Es werden solange Zeichen ausgegeben, bis der String zu Ende ist (binäre 0) oder die Anzahl der Zeichen, die durch die 2.Ziffernfolge angegeben wird, erreicht ist.

In den beiden folgenden Fällen ist das Argument eine Zahl vom Typ float oder double:

E,e liefert: [-]m.ddd...dE[+-]nnn Die Anzahl der d ist gleich 6, falls keine 2.Ziffernfolge angegeben ist, sonst gleich der dort angegeben Zahl (maximal 13).

F,f liefert: [-]mmmm.ddd...d Anzahl d wie bei E,e.

G,g je nachdem ob E,e oder F,f die kürzere Ausgabe darstellt, wird nach E,e oder F,f umgewandelt.

Jedes Zeichen nach %, das mit keinem Umwandlungzeichen übereinstimmt, wird nach der Angabe 1.Ziffernfolge . 2.Ziffernfolge genauso wie die umgewandelten Argumente ausgegeben. Falls die Angabe 1.Ziffernfolge . 2.Ziffernfolge fehlt, werden diese Zeichen genauso ausgegeben, wie sie nach % stehen, also genauso wie Zeichen, denen kein % vorausgeht.

Beispiel
PRINTF ("%MI-C-COMPILER"); liefert die gleiche Ausgabe wie PRINTF ("MI-C-COMPILER"); nämlich: MI-C-COMPILER
aber PRINTF ("%010.7MI-C-COMPILER"); liefert die Ausgabe 000MI-C-CO
und PRINTF ("010.7MI-C-COMPILER"); liefert
010.7MI-C-COMPILER

double z; z enthält 432.56789345
PRINTF("%-015.3E",z) liefert +4.326E+0020000
PRINTF ("%015.3F",z) liefert 00000000432.568

SSCANF(STRING, CONTROL, ARG1, ARG2,...)
char *STRING;
Beschreibung siehe SCANF.
Statt von der Standarteingabe werden Zeichen aus STRING nach den Angaben in CONTROL an die Stellen gespeichert, auf die das jeweilige Argument zeigt.

SPRINTF(STRING, CONTROL, ARG1, ARG2,...)
char *STRING;
Beschreibung siehe PRINTF.
Wie bei PRINTF werden auch hier ARG1, ARG2, ... so umgewandelt, wie es CONTROL angibt. Die Ergebnisse der Umwandlungen werden in STRING abgelegt.

FSCANF(F, CONTROL, ARG1, ARG2, ...)
FILE *F;
Beschreibung siehe SCANF.
Die Zeichen werden aus der zu F gehörigen Datei (mit FOPEN eröffnet) nach den Angaben in CONTROL an die Stelle gespeichert, auf die das jeweilige Argument ARGi zeigt.

FPRINTF(F, CONTROL, ARG1, ARG2, ...)
FILE *F;
Beschreibung siehe PRINTF.
Die umgewandelten Argumente ARG1, ARG2, ... werden in die zu F gehörige Datei geschrieben.

III. ALLGEMEINE SYSTEMFUNKTIONEN

BDOS(DE,C)
char *DE; int C;

Eine Betriebssystemleistung wird angefordert. Die BDOS Funktion, deren Nummer im Parameter C steht, wird mit dem Eingabeparameter aus Parameter DE ausgeführt. Das Resultat wird als Funktioswert zurückgeliefert.
z.B. BDOS('A',2); gibt ein A auf dem Terminal aus.

_EXIT()

_EXIT bewirkt sofortigen Programmabbruch.

EXIT(n)
int n;

EXIT bewirkt einen Programmabbruch durch Aufruf von _EXIT, nachdem vorher alle offenen Dateien geschlossen wurden. Der Parameter n hat keine Bedeutung.

CHAIN(name,parameter)
char *name, *parameter;

CHAIN schließt alle Dateien, beendet das laufende Programm und startet das Programm aus der Datei name, wobei im String parameter Information weitergegeben werden kann. Der String muß genauso aufgebaut sein wie die Kommandozeile für den CCP.
Z.B. CHAIN("CC.COM","/T BEISPIEL");

CLOSAL()

Mit CLOSAL() werden alle offenen Dateien geschlossen. Hierbei ist es gleichgültig ob sie mit fopen, open oder creat eröffnet wurden.

UNLINK(name)
char *name;

Mit UNLINK(name) wird die Datei name aus dem "Dateienverzeichnis" gelöscht. Hierbei ist es egal ob die Datei eröffnet ist oder nicht.

IV. STRINGFUNKTIONEN

```
STRCPY(strz,stra)
char *strz, *stra;
```
Der Ausgangsstring stra wird in den Zielstring strz kopiert.

```
STRNCPY(strz,stra,max)
char *strz, *stra;
int max;
```
Vom Ausgangsstring stra werden maximal max viele Zeichen in den Zielstring strz kopiert. Eventuell steht dann in strz ein String, der nicht durch \0 beendet wird!!

```
STRCMP(str1,str2)
char *str1, *str2;
```
str1 und str2 werden miteinander verglichen. Falls alle Zeichen aus str1 mit denen aus str2 übereinstimmen, wird 0 zurückgegeben. Falls im String str1 ein Zeichen gefunden wird, das nicht mit dem entsprechenden Zeichen aus str2 übereinstimmt, wird die Differenz dieser beiden Zeichen zurückgegeben. Falls das Zeichen in str1 einen kleineren Wert hat als das in str2, eine negative Zahl sonst eine positive.

```
STRNCMP(str1,str2,max)
char *str1, *str2;
int max;
```
Wie STRCMP, aber nur maximal max viele Zeichen werden verglichen.

```
STRLEN(str)
char *str;
```
STRLEN(str) gibt die Länge des Strings str zurück. Die binäre Null am Ende des Strings wird nicht mitgezählt.

```
STRCAT(str1,str2)
char *str1, *str2;
```
Der String str2 wird an das Ende des Strings str1 angefügt. str1 muß groß genug gewählt werden!

```
STRNCAT(str1,str2,max)
char *str1, *str2;
int max;
```

Vom String str2 werden höchstens max viele Zeichen an das Ende des Strings str1 angefügt. Eine binäre Null wird ans Ende gesetzt.

```
char *STRSAVE(str)
char *str;
```

STRSAVE sichert den String str in einen mit CALLOC bereitgestellten Platz und gibt einen Pointer darauf zurück. Falls mit CALLOC kein Platz mehr zur Verfügung gestellt werden konnte, wird NULL zurückgegeben.

```
char *INDEX(str1,str2)
char *str1, *str2;
```

Es wird überprüft, ob str2 in str1 enthalten ist. Falls nein, wird eine -1 zurückgegeben. Falls ja, wird die Position zurückgegeben, an der str2 in str1 anfängt.

```
char *RINDEX(str1,str2)
char *str1, *str2;
```

RINDEX arbeitet wie INDEX. Allerdings wird, falls str2 mehr als einmal im String str1 enthalten ist, ein Pointer auf das letzte Auftreten zurückgegeben.

V. TEST- UND UMWANDLUNGSFUNKTIONEN

In der Bibliothek stehen noch folgende Test und Umwandlungsfunktionen zur Verfügung:

ISALPHA(C) int C;
liefert das Zeichen aus C zurück, falls es ein (klein oder groß geschriebener) Buchstabe ist, sonst 0.

ISUPPER(C) int C;
liefert das Zeichen aus C zurück, falls es ein groß geschriebener Buchstabe ist, sonst 0.

ISLOWER(C) int C;
liefert das Zeichen aus C zurück, falls es ein klein geschriebener Buchstabe ist, sonst 0.

ISDIGIT(C) int C;
liefert das Zeichen aus C zurück, falls es eine Ziffer ist, sonst 0.

ISALNUM(C) int C;
liefert das Zeichen aus c zurück, falls es ein Buchstabe oder eine Ziffer ist, sonst 0

ISASCII(C) int C;
liefert das Zeichen aus c zurück, falls es ein ASCII-Zeichen ist, 0 sonst.

ISSPACE(C) int C;
liefert das Zeichen aus C zurück, falls es = ' ' oder '\T' oder '\N' oder '\R' ist, sonst 0.

TOLOWER(C) int C;
liefert den klein geschriebenen Buchstaben aus C zurück, falls das Zeichen in C ein Großbuchstabe war, sonst das Zeichen selbst.

TOUPPER(C) int C;
liefert den groß geschriebenen Buchstaben aus C zurück, falls das Zeichen in C ein klein geschriebener Buchstabe war, sonst das Zeichen selbst.

```
ATOI(str1)
char *str1;
```
Der String str1 wird in eine Zahl vom Typ int umgewandelt.

```
double ATOF(str1)
char *str1;
```
Der String str1 wird in eine Zahl vom Typ double gewandelt.

```
long ATOL(str1)
char *str1;
```
Der String str1 wird in eine long Zahl umgewandelt.

```
ITOA(zahl,str)
int zahl; char *str;
```
Die Zahl zahl wird in eine Folge von ASCII - Zeichen gewandelt

```
ABS(n)
int n;
```
ABS(n) liefert den Absolutbetrag der int n zurück.

```
long ABSL(n)
long n;
```
liefert den Absolutbetrag der long Zahl n zurück.

```
double ABSD(n)
double n;
```
liefert den Absolutbetrag der double Zahl n zurück.

VI. SPEICHERPLATZVERWALTUNG

```
char *CALLOC(n, laenge)
unsigned n, laenge;
```

Mit CALLOC(n, laenge) wird freier Speicherplatz für n*laenge viele Zeichen angefordert. Es wird NULL zurückgegeben, falls nicht genügend freier Platz mehr vorhanden ist. Anderenfalls wird ein Pointer auf einen freien Speicherbereich, der mindestens n*laenge viele Zeichen aufnehmen kann, zurückgegeben.

```
CFREE(ptr)
*ptr;
```

Mit CFREE(ptr) wird der mit CALLOC angeforderte Speicherplatz wieder in die Liste des von CALLOC verwalteten freien Speicherplatzes zurückgegeben. ptr muß ein von CALLOC gelieferter Pointer sein!

```
char *SBRK(n)
unsigned n;
```

Mit SBRK(n) wird weiterer Speicher von n Byte Länge angefordert. SBRK liefert einen Pointer auf den freien Speicherplatz zurück oder, falls kein Platz mehr zur Verfügung gestellt werden kann, NULL.

Man beachte, daß bei gleichzeitiger Benutzung von SBRK und CALLOC der Speicher von CALLOC möglicherweise nicht mehr effektiv verwaltet werden kann.

E. LISTE DER EINSCHRÄNKUNGEN, ERWEITERUNGEN UND BESONDERHEITEN:

- #include darf nicht geschachtelt sein (d.h. eine mittels #include eingefügte Datei darf kein weiteres #include enthalten)
- Variablen, Felder und Strukturen werden nicht automatisch mit 0 vorbesetzt (aber: Felder und Strukturen, bei denen mindestens ein Element initialisiert ist, werden mit 0 aufgefüllt) Diese Einschränkung gilt nicht bei der Verwendung von M80/L80.
- Bitfelder (bei Strukturen) gibt es nicht
- Variablen vom Typ float oder double können nur mit Konstanten vom Typ double initialisiert werden. (also: float a = 1.0; aber nicht: float a = 1;)
- Namen und Typen von Elementen von Strukturen unterliegen nicht der Beschränkung, daß nur solche Elemente aus verschieden Strukturen gleichnamig sein dürfen, die gleichen Typ und gleichen Abstand vom Anfang der jeweiligen Struktur haben.
- Bei switch (AUSDRUCK) darf AUSDRUCK auch den Typ long oder double haben. Dann muß hinter case eine Konstante vom gleichen Typ stehen.
- Ein Pointer, auf den andere arithmetische Operatoren als Addition oder Subtraktion angewandt werden, wird behandelt, als wäre er eine Variable vom Typ unsigned.
- die Anweisung #asm ... #endasm ist hinzugekommen. Alle Zeilen, die dazwischen stehen, werden unverändert in die Ausgabedatei gegeben. So können Assemblerprogramme an geeigneten Stellen des C - Programms eingefügt werden. Z.B. :

a)
```
input0()
{
#ASM
    IN 0
    MOV L,A
    MVI H,0
#ENDASM
}
```

b)
```
output0(byte)
int byte; {
byte;
#ASM
     MOV A,L
     OUT 0
#ENDASM
}
```

c)
```
output0(byte)
int byte; {
#asm
     POP B
     POP H
     PUSH H
     PUSH B
     MOV A,L
     OUT 0
#endasm
}
```

input0 liefert ein Byte von Port 0 als Funktionswert zurück und output gibt eines aus. b) funktioniert so nur unter der Option X. c) kann in jedem Fall verwendet werden.

F. ZAHLENDARSTELLUNG

Ganzzahlige Typen sind Binärzahlen , wobei negative Zahlen im Zweierkomplement dargestellt werden.

Typ:			char		
Zahl:	-128	-1	0	1	127
HEX:	80H	FFH	0	01H	7FH

Typ:		char (Option S)	
Zahl:	0	1	255
HEX:	0	01H	FFH

Typ:			int		
Zahl:	-32768	-1	0	1	32767
HEX:	8000H	FFFFH	0	0001H	7FFFH

Typ:		unsigned	
Zahl:	0	1	65535
HEX:	0	0001H	FFFFH

Typ:			long		
Zahl:	-2147483648	-1	0	1	2147483647
HEX:	80000000H	FFFFFFFFH	0	00000001H	7FFFFFFFH

Gleitkommazahlen haben eine gepackte BCD Zahl als Mantisse und einen 1 Byte langen binären Exponenten (zur BASIS 10) mit einem Offset von 128. Der Exponent 0H bedeutet, daß die ganze Zahl unabhängig von der Mantisse 0 ist. Zahlen ungleich 0 sind immer normalisiert, d.h. die höchstwertige Ziffer der Mantisse einer positiven Zahl ist ungleich 0 und hat den Wert 'Ziffer * 10 ** -1'. Negative Zahlen werden im neuner Komplement dargestellt. Beim Typ float ist die Mantisse 3 Byte lang (5 Stellen) und beim Typ double 7 Byte (13 Stellen). Gerechnet wird immer mit 13 Stellen, wobei die 14-te Stelle zum Runden berücksichtigt wird.
Extremwerte:

Zahl		(high) Hex (low)
0.9999999999999	* 10 ** 127	FF 09 99 99 99 99 99 99
- 0.9999999999999	* 10 ** 127	FF 90 00 00 00 00 00 01
0		00 .. ..
0.1	*10 **(-127)	01 01 00 00 00 00 00 00
- 0.1	*10 **(-127)	01 99 00 00 00 00 00 00

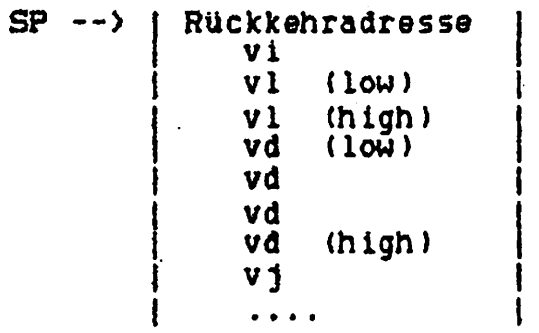
G. GESCHWINDIGKEITSOPTIMIERUNG

'i.A.' bedeutet hier, daß in komplizierteren Ausdrücken durch Optimierung leichte Verschiebungen stattfinden können.

- Statische und externe Variable erlauben einen schnelleren Zugriff als temporäre (Speicherklasse auto).
- Die zuletzt deklarierte, temporäre Variable der Länge 2 Byte (keine temporäre Variable im gleichen oder einem innereren Block danach erklärt), erlaubt i.A. einen schnelleren Zugriff als andere temporäre Variable. In VA++ oder ++VA ist eine wie vor beschriebene Variable i.a. 'schneller' als eine statische oder externe Variable.
- Variablen vom Typ float sind 'langsamer' als solche vom Typ double und diese 'langsamer' als die vom Typ long und diese 'langsamer' als die übrigen.
- Variable vom Typ char können mit Option S schneller 'geholt' werden als ohne. Das Abspeichern ist zeitgleich.
- der Typ char ist unter der Option S schneller.
- Nach float va; ist va = 1.1; schneller als va = 1; da keine Typumwandlung stattfinden muß.
- Ein for hat einen Assembler JMP in die Schleife hinein mehr als die äquivalent while Anweisung.
- Sei definiert static int ch;
  a) if (ch == 'A') . else if (ch == 'D') .. else ..
  b) switch(ch) {case 'A': ... case 'D': ... ...}
  b) ist schneller als a). Ab 3 Alternativen ist b) auch kürzer als a).
- *(ptr + x) ist das gleiche wie ptr[x].

H. ANSCHLUSS VON ASSEMBLERPROGRAMMEN AN C - PROGRAMME UND ANSCHLUSS VON C - FUNKTIONEN AN ANDERE PROGRAMME

Beim Anschluß in beiden Richtungen muß die richtige Parameterübergabe beachtet werden. Funktionen ohne Parameter werden mit einem Assembler CALL aufgerufen und kehren mit RET zurück. Der Speicher hinter dem Stackpointer wird als frei angesehen. Aktuelle Parameter werden im Stack übergeben und dürfen vom aufgerufenen Programm verändert werden. Diese Änderung hat keine Auswirkung auf das aufrufende Programm. (Um dort Werte zu verändern können Pointer verwendet werden.)
Sei definiert: int vi, vj; long vl; double vd;
Die Funktion fu findet den Stack nach dem Aufruf
fu(vi,vl,vd,vj); wie folgt vor:



Auf den Parameter vj kann wie folgt zugegriffen werden, wenn seit Eintritt in die Funktion der Stackpointer nicht verändert wurde:

```
LXI H,16
DAD SP
MOV E,M
INX H
MOV D,M
```

Die Funktion fu kann mit einem Assembler RET zurückkehren. Die Parameter werden vom aufrufenden Programm vom Stack entfernt. Die Register sowie die Parameter dürfen verändert sein.
Parameter vom Type char werden nach int gewandelt und solche vom Typ float nach double, bevor sie auf den Stack gebracht werden.

Beispiel für die Ausgabe des Zeichens ? von einem Assemblerprogramm aus mittels der Funktion PUTCHAR:

```
LXI  H,'?'
PUSH H
CALL PUTCHAR
POP  H
```

Ein Funktionswert kann abhängig vom Typ der Funktion zurückgeliefert werden.

Typ	Ort
double	8 Byte ab der Adresse CFPRIM
long	die beiden höherwertigen Byte ab der Adresse CLPRIM und die beiden niederwertigen Byte im HL = Register
sonst	im HL - Register

Auf externe nicht static deklarierte C - Objekte kann mittels des Namens zugegriffen werden. (z.B. wie oben auf die Funktion PUTCHAR)

I. SYSTEMGROSSEN

Es gelten folgende Begrenzungen:

- die Länge der C - Quelle ist unbegrenzt
- die Länge der Ausgabe wird nur durch die maximal zulässige Dateigröße eingeschränkt
- maximal 300 verschiedene externe Symbole (z.B. Variablen,Felder, Funktionen) in einem Programmteil gleichzeitig (im ganzen Programm unbegrenzt)
- maximal 60 aktive lokale Symbole (d.h. in einer Verschachtelung von Blöcken) pro Funktion
- maximal 50 verschiedene externe Strukturdefinitionen in einem Programmteil
- maximal 25 verschiedene aktive, lokale Strukturdefinitionen pro Funktion
- maximal 259 Macrodefinitionen (#define) in einem Programmteil
- maximal 80 Zeichen für alle Parameter in einem Macroaufruf mit Parametern
- Anzahl der Parameter bei einem Funktionsaufruf: maximal 40
- maximale Zeilenlänge 159 Zeichen
- Schachtelungstiefe von Blöcken: 40
- Schachtelungstiefe von while, do .. while, for, switch (auch gemischt) 33
- maximale Anzahl von Marken bzw. goto zu noch nicht definierter Marke 50 pro Funktion
- maximal 14 Modifikatoren pro definiertem Objekt (z.B. (******* (*fu)())[][][][][][])

Wenn nicht ausreichend Speicherplatz vorhanden ist, können eventuell nicht alle Werte gleichzeitig ausgeschöpft werden.
Bei besonderen Erfordernissen kann eine Änderung dieser Größen vorgenommen werden.

1. HARDWARE UND SOFTWARE VORAUSSETZUNGEN

Es wird ein 8080,8085 oder Z80 Rechner mit einem CP/M System (Version 1.4 oder höher) benötigt. Der Arbeitsspeicher muß von Adresse TPA = 100H bis BDOS frei benutzbar sein. Der Compiler benötigt in der Version 3.18 mindestens 50k Speicher (d.h. ein 56k CP/M System). Für große Programme wird mehr Speicher benötigt. Ein 60k CP/M System ist empfehlenswert.
Ist weniger Speicher vorhanden, kann auf Anfrage auch eine Spezialversion bezogen werden.
Außerdem wird ein Assembler benötigt, wobei der zum CP/M System mitgelieferte ASM.COM ausreicht. Ein Assembler mit Linker (speziell MAC80 / L80 von Microsoft) erleichtert die Handhabung, besonders, wenn Programme in mehrere Teile aufgeteilt werden.

K. FEHLERMELDUNGEN

I. FEHLERMELDUNGEN DES COMPILERS

Wenn vom Compiler ein Fehler oder ein Mangel entdeckt wird, gibt er eine Fehlermeldung aus. Die Stelle, an der der Fehler bemerkt wird, ist durch ein ^ unter der Quellzeile, die auch ausgegeben wird, markiert. Es können natürlich nur syntaktische Fehler entdeckt werden, und keine Fehler, die in der Logik des Programms ihre Ursache haben. Es gilt die Regel:
Das Fehlen von (,), [,], ; und : bei bedingten Ausdrücken wird in 'eindeutigen' Fällen korrigiert.
Alle anderen Fehler werden in keinem Fall vom Compiler korrigiert.
Eine typische Meldung ist folgende:

```
;for (i =0 i < 5 ; ++i) funk(i);
:
;*****Zeile: 5   SEMIKOLON FEHLT  ******
```

Dieser Fehler wird vom Compiler korrigiert, wie auch das Fehlen der beiden zum for gehörenden Klammern korrigiert würde, aber es wird ausdrücklich geraten, falls Fehler aufgetaucht sind, diese in der Quelle zu korrigieren und neu zu übersetzen. Eine Fehlermeldung kann nämlich zu falschen Schlüssen verleiten, und das erzeugte Programm ist dann falsch. Sei z.B. folgendes ein vollständiger Programmteil:

```
outstr(ptr) char *ptr; { while (*ptr) PUTCHAR(*ptr++);
nz()  { PUTCHAR('\r'); PUTCHAR('\N'); }
```

Folgende Fehlermeldung erscheint.

```
;;nz()  { PUTCHAR('\r'); PUTCHAR('\N'); }
:
;*****Zeile: 2  SEMIKOLON FEHLT  ******
```

Der eigentliche Fehler ist hier nicht das Fehlen eines ; sondern die vergessene } ,die die Funktion PUTCHAR am Ende der ersten Zeile abschließt. Ein solcher Fehler läßt sich nur korrigieren, wenn man weiß, was das Programm an dieser Stelle leisten soll.
An diesem Beispiel kann man auch sehen, daß ein Fehler Folgefehlermeldungen nach sich zieht. Am Ende des Programms wird bemerkt, daß zu einer { die zugehörige } fehlt, und der Compiler meldet das Fehlen einer } .

Im folgenden sind die Fehlermeldungen des Compilers aufgelistet und mögliche Ursachen angegeben.

"ANZAHL PARAM. FALSCH"
"WRONG NUMBER ARGS"
Bei der Definition einer Funktion stimmen Parameterliste und zugehörige Parameterdefinitionen nicht überein.

"AUSGABEDATEI EROEFFNEN NICHT MOEGLICH"
"OPEN FAILURE"
Die Diskette ist voll, oder ein Schreibschutz ist gesetzt.

"AUSGABEDATEIFEHLER"
"OUTPUTFILE ERROR"
Mögliche Fehlerursache: Die Diskette ist voll oder defekt.

"DEKL. FEHLT"
"DECL. MISSING"
Fehlerhafte Strukturdefinition. Vielleicht wurde der Name einer Variablen vergessen.

"DEKLARATION ZU KOMPLEX"
"DECLARATION TOO COMPLEX"
Siehe Abschnitt Systemgrößen

"EROEFFNEN .LST NICHT MOEGLICH"
"CAN'T OPEN .LST"
Die Diskette ist voll, oder ein Schreibschutz ist gesetzt.

"FALSCHE MAKRO-ARGUMENTE"
"WRONG MACRO-ARGUMENTS"
Die Parameterliste eines Makroaufrufes ist fehlerhaft.

"FALSCHER AUSDRUCK"
"INVALID EXPRESSION"
In einem Ausdruck wird an dieser Stelle ein Name oder eine Konstante erwatet.

"FALSCHER PARAMETERTYP"
"WRONG PARAMETER TYPE"
Eine Funktion, Struktur oder Union ist als Parameter einer Funktion in C nicht zulässig. (Wohl Pointer darauf)

"FALSCHER SYMBOLNAME"
"ILLEGAL SYMBOL NAME"
An dieser Stelle wird zwingend ein Name erwartet. z.B. int +i;

"FLOAT FALSCH"
"WRONG FLOAT"
Es liegt eine fehlerhafte Gleitkommakonstante vor.

"FUNKTION GEFORDERT"
"MUST BE FUNCTION"
Nur auf einen Ausdruck vom Typ Funktion darf eine Parameterliste folgen. Möglicherweise wurde in einem Ausdruck ein Operator vergesssen: z.B. in
A * (b + c); der Operator *

"FUNKTION NICHT ERLAUBT"
"FUNCTION NOT ALLOWED"
Hier ist die Definition einer Funktion nicht erlaubt. z.B. innerhalb einer Struktur kann keine Funktion definiert werden. Vielleicht fehlen Klammern, um einen Pointer auf eine Funktion zu definieren, was erlaubt ist: int (* fu) ();

"FUNKTIONS-DEF VERBOTEN"
"FUNCTION-DEF NOT ALLOWED"
In einer externen Definition kann, wenn nicht vorher das Schlüsselwort extern steht, nach , keine Funktionsdefinition vorkommen. z.B.
static int i,funk(); ist als externe Definition nicht erlaubt.

"#IF FEHLT"
"MISSING #IF"
Es taucht #else oder #endif auf ohne zugehöriges #if.

"INCLUDE-DATEI NICHT DA"
"OPEN FAILURE INCLUDE-FILE"
Selbsterklärend

"INDIREKTION"
"INDIRECTION"
Der monadische Operator * wird auf einen Ausdruck angewandt, der nicht vom Typ Pointer ist.

"INDIZIERUNG VERBOTEN"
"CAN'T SUBSCRIPT"
Nur ein Ausdruck vom Typ Pointer oder Feld kann mit [..] indiziert sein.

"INIT. NICHT ERLAUBT"
"INIT NOT ALLOWED"
Objekte wie z.B. Funktionen können nicht mit einem Wert vorbesetzt (initialisiert) werden.

"INIT. UNION VERBOTEN"
"CAN'T INIT. UNION"
Eine union kann nicht initialisiert werden.

"KEIN FUNKTIONENFELD"
"NO ARRAY OF FUNCTION"
In C gibt es kein Feld dessen Elemente Funktionen sind. (Pointer auf Funktionen sind zulässig.)

"KEIN PARAMETERNAME"
"EXPECTED ARGUMENT"
Bei einer Funktionsdefinition taucht bei der Parameterdefinition ein Name auf, der nicht in der Parameterliste erschienen ist.

"KEIN PLATZ FUER STRINGS"
"NO STRING SPACE"
Der Platz für Stringkonstante ("....") innerhalb der gerade bearbeiteten Funktion ist verbraucht. Abhilfe: Hilfsfunktionen oder Initialisieren von Feldern oder Pointern mit den Strings. Siehe auch Abschnitt Systemgrößen.

"KEIN PLATZ: DEKL."
"NO SPACE: DECL."
Es ist zu wenig Speicherplatz vorhanden. Abhilfe: Das Programm wird in mehrere Teile aufgeteilt und diese getrennt compiliert. Siehe auch Abschnitt Systemgrößen

"KEIN PLATZ: LOKALE DEKL."
"NO SPACE: LOCAL DECL."
Es ist zu wenig Speicherplatz vorhanden. Eventuelle Abhilfe durch die Verwendung externer Variablen oder Hilfsfunktionen möglich. Siehe auch Abschnitt Systemgrößen

"KEIN PLATZ: LOKALE S/U DEKL."
"NO SPACE: LOCAL S/U DECL."
Es ist zu wenig Speicherplatz vorhanden. Eventuelle Abhilfe durch die Verwendung externer Variablen oder Hilfsfunktionen möglich. Siehe auch Abschnitt Systemgrößen

"KEIN PLATZ: S/U DEKL."
"NO SPACE: S/U DECL."
Es ist zu wenig Speicherplatz vorhanden. Abhilfe: Das Programm wird in mehrere Teile aufgeteilt und diese getrennt compiliert. Siehe auch Abschnitt Systemgrößen

"KEIN STRUCT/UNION VAR-NAME"
"NO STRUCT/UNION VAR-NAME"
Nach einem . oder -> wird ein Name aus der zugehörigen Strukturdefinition erwartet.

"KEINE AKTIVE SCHLEIFE"
"NO OPEN LOOP"
continue darf nur innerhalb einer Schleife benutzt werden.

"KEINE AKTIVEN SCHLEIFEN/SWITCH"
"NO ACTIVE LOOPS/SWITCH"
break darf nur innerhalb einer Schleife/switch benutzt werden.

"KEINE FELDFUNKTION"
"NO ARRAY FUNCTION"
In C gibt es keine Funktion die als Funktionswert ein Feld zurückliefert. (Pointer sind zulässig.)

"KEINE FUNKTIONS-FUNKTION"
"NO FUNCTION FUNCTION"
In C gibt es keine Funktion die als Funktionswert eine Funktion zurückliefert. (Pointer auf Funktionen sind zulässig.)

"KEINE STRUCT/UNION FUNKTION"
"NO STRUCT/UNION FUNCTION"
In C gibt es keine Funktion die als Funktionswert eine Struktur oder Union zurückliefert. (Pointer darauf sind zulässig.)

"KLAMMER FEHLT"
"MISSING BRACKET"
Die Art der Klammer wird mit ausgegeben. Die eigentliche Fehlerursache kann insbesondere bei geschweiften Klammern an einer anderen Stelle als der gemeldeten liegen.

"KOMMA ERWARTET"
"EXPECTED COMMA"
Die Parameterliste oder Parameterdefinitionen bei einer Funktionsdefinition sind fehlerhaft.

"LABEL FEHLT"
"LABEL NOT DEFINED"
Diese Fehlermeldung erscheint am Ende einer Funktion, wenn eine Sprungmarke, die bei einem goto aufgetreten ist, nicht innerhalb der Funktion als Marke vorgekommen ist. Der betreffende Name wird in der Meldung mitangegeben.

"LAENGE FEHLT"
"LENGTH MISSING"
In der Definition eines mehrdimensionalen Feldes fehlt die Längenangabe in einer anderen als der ersten Dimension. (Fehlt sie in der ersten Dimension, so wird automatisch ein Pointer definiert.)

"MAKRO TAB. VOLL"
"MACRO TABLE FULL"
Es ist zu wenig Speicherplatz vorhanden. Abhilfe: Das Programm wird in mehrere Teile aufgeteilt und diese getrennt compiliert. Siehe auch Abschnitt Systemgrößen.

"MARKE FEHLT"
"MUST BE LABEL"
Nach goto fehlt die Sprungmarke.

"MUSS FUNKTION SEIN"
"MUST BE FUNCTION"
Befindet man sich außerhalb von Funktionen, so muß eine externe Definition ohne Speicherklasse und Typ zu einer Funktionsdefinition gehören und auf den definierten Namen eine (folgen. Diese Meldung erscheint z.B. auch wenn ein Typ falsch geschrieben wurde wie innt statt int.

"MUSS GANZZAHLIG SEIN"
"MUST BE INTEGRAL"
Manche Operatoren wie bitweises und, oder, exklusives oder erfordern als Operanden Ausdrücke von ganzzahligem Typ.

"MUSS KONSTANT SEIN"
"MUST BE CONSTANT"
An einigen Stellen dürfen nur konstante Ausdrücke auftreten. (#if, Länge in Felddefinitionen, case, Initialisierung). Die Adresse einer temporären Variable ist keine Konstante. Siehe auch C Sprachbeschreibung.

"MUSS LVALUE SEIN"
"MUST BE LVALUE"
An verschiedenen Stellen (z.B. links von einem Gleichheitszeichen) kann nur ein Ausdruck, der ein lvalue ist, auftreten. Ein Feldname z.B. ist kein lvalue.

"MUSS TYP SEIN"
"MUST BE TYPE"
In einer Definition wird ein Typ oder ein typedef - Name erwartet. Eventuell liegt ein Rechtschreibefehler vor.
"NEGATIVE LAENGE"
"NEGATIVE SIZE"
Die Länge eines Feldes darf nicht negativ sein.

"OPTION A"
Unter der Option A ist die Initialisierung von internen statischen Variablen nicht erlaubt. Mögliche Abhilfe: Verwendung von externen statischen Variablen.
Außerdem müssen unter der Option A statische Funktionen vor dem ersten Auftreten deklariert werden.

"REFLEXIV S/U -MUSS POINTER SEIN"
"REFLEXIV S/U -MUST BE POINTER"
Innerhalb einer Struktur- oder Union Definition darf die gerade definierte Struktur oder Union nicht noch einmal auftauchen sondern nur ein Pointer darauf.

"SCHON DEFINIERT"
"ALREADY DEFINED"
In der Meldung folgt der Name, der bereits an anderer Stelle definiert wurde.

"SEMIKOLON FEHLT"
"MISSING SEMICOLON"
Es wird ein Semikolon erwartet als Ende einer Anweisung oder in for - Anweisungen.

"STACK UEBERLAUF ERWARTET"
"STACK OVERFLOW EXPECTED"
Es ist zu wenig Speicherplatz vorhanden. Der Stack läuft möglicherweise während der weiteren Bearbeitung der Anweisung in die Tabelle der lokalen Symbole über. Das erzeugte Programm kann fehlerhaft werden. Treten keine weiteren Fehlermeldungen auf, so kann, nach Prüfung der Assemblerdatei an dieser Stelle, das Assemblerprogrammm weiterverwandt werden.

"STRUCT/UNION FALSCH"
"WRONG STRUCT/UNION"
Fehlerhafte Definition einer Struktur oder Union.

"STRUCT/UNION LEER"
"STRUCT/UNION EMPTY"
Eine Strukturdefinition darf nicht leer sein.

"STRUCT/UNION NOETIG"
"MUST BE STRUCT/UNION"
. oder -> kann nur auf einen Ausdruck vom Typ Struktur angewandt werden.

"TYP FALSCH"
"WRONG TYPE"
Es liegt kein zulässiger Typ vor. z.B. Bei einer Definition, einer Typumwandlung oder bei sizeof

"TYP UNVEREINBAR"
"TYPE MISMATCH"
Zwei Typen sind nicht miteinander verträglich. z.B. wird eine Variable einmal als extern int var; ein anderes Mal als extern long var; erklärt. Oder: in einem bedingten Ausdruck sind die beiden möglichen Ergebnisse Pointer von verschiedenem Typ.

"UEBERLAUF GLOBALE SYMBOLTAB."
"GLOBAL SYMBOL TABLE OVERFLOW"
Es wurden zu viele externe Namen definiert. Abhilfe: Das Programm wird in mehrere Teile aufgeteilt und diese getrennt compiliert. Siehe auch Abschnitt Systemgrößen

"UEBERLAUF LOKALE SYMBOLTAB."
"LOCAL SYMBOL TABLE OVERFLOW"
Es wurden zu viele lokale Variablen etc. in einem Nest von Blöcken innerhalb einer Funktion deklariert. Eventuelle Abhilfe durch die Verwendung externer Variablen oder Hilfsfunktionen möglich.
Siehe auch Abschnitt Systemgrößen

"UNDEFINIERTER NAME"
"UNDEFINED IDENTIFIER"
Ein Name taucht auf, ohne vorher definiert worden zu sein.

"WHILE FEHLT"
"MISSING WHILE"
Nach do fehlt das zugehörige while; . Eventuell wurde vergessen { .. } um die von do und while eingeschlossenen Anweisungen zu setzen.

"ZEILE ZU LANG"
"LINE TOO LONG"
Siehe Abschnitt Systemgrößen

"ZUVIELE AKTIVE SCHLEIFEN/SWITCH"
"TOO MANY ACTIVE LOOPS/SWITCH"
Siehe Abschnitt Systemgrößen

"ZUVIELE BLOECKE"
"TOO MANY LEVELS"
Die Schachtelungstiefe bei Blöcken ist zu groß.
Siehe auch Abschnitt Systemgrößen

"ZUVIELE GOTO/LABEL"
"TOO MANY GOTO/LABEL"
Siehe Abschnitt Systemgrößen

"ZUVIELE MAKROS"
"TOO MANY MACROS"
Es wurden zu viele Makros definiert (#define). Abhilfe: Das Programm wird in mehrere Teile aufgeteilt und diese getrennt compiliert. Siehe auch Abschnitt Systemgrößen.

"ZUVIELE PARAMETER"
"TOO MANY PARAMETERS"
Ein Funktionsaufruf hat zu viele Parameter. Siehe auch Abschnitt Systemgrößen.

"ZUVIELE REFLEXIVE S/U POINTER"
"TOO MANY REFLEXIV S/U POINTERS"
in einer Strukturdefinition tauchen zu viele Pointer auf eine Struktur vom gerade definierten Typ auf.

"" FEHLT"
"NO QUOTE"
Ein String ist nicht richtig abgeschlossen. Möglicherweise sind dadurch nachfolgende Anweisungen mit in den String hineingenommen worden.

"' FEHLT"
"NO APOSTROPHE"
Eine Zeichenkonstante ist nicht richtig abgeschlossen.

"& OHNE LVALUE"
"ILLEGAL ADRESS"
Der monadische Operator & wird auf einen Ausdruck, der kein lvalue ist, angewandt. Siehe C Sprachbeschreibung.

II. LAUFZEITFEHLERMELDUNGEN

Während des Laufes eines Programmes können die folgenden Fehlermeldungen am Terminal auftreten:

"0 - DIVISION"
Wenn eine Division durch 0 festgestellt wird, bricht das Programm ab.

"OVERFLOW"
Wenn eine Gleitkommazahl bei einer arithmetischen Operation oder beim Runden dem Betrage nach größer als die größte zulässige Gleitkommazahl wird, erscheint diese Meldungen auf dem Terminal. Das Programm wird fortgeführt, und es wird die betragsmäßig größte positive oder negative Zahl eingesetzt.

"UNDERFLOW"
Wenn eine Gleitkommazahl bei einer arithmetischen Operation oder beim Runden dem Betrage nach kleiner als die kleinste zulässige Gleitkommazahl wird, erscheint diese Meldungen auf dem Terminal. Das Programm wird fortgeführt, und es wird die Zahl 0.0 eingesetzt.

Die Reaktion auf einen Laufzeitfehler kann vom Benutzer geändert werden. Dazu müssen die entsprechenden Bibliotheksprogramme CCODIV, CLODIV, CFODIV, CFOVERFLOW oder CFUNDERFLOW geändert werden.

STICHWORTVERZEICHNIS ZU KAPITEL C UND D

Liste der Dateien auf der Diskette

ABBRUCH.COM	Hilfsprogramm für Kommandofolgen
BEISPIEL.C	Beispielprogramm
CC.COM	MI - C Compiler 8080 - Version
CCZ.COM	MI - C Compiler Z80 - Version
CC.SUB	Beispiel einer Kommandofolge
CCA.SUB	
CFLIB.ASM	Assemblerquelle der Gleitkommabibliothek
CFUMWAND.ASM	Assemblerquelle der Gleitkommazahlenumwandlungen
CHAIN.C	Aufruf eines Nachfolgeprogramms
CHAIN.REL	
CIO.C	C-Quelle der unformatierten Ein- Ausgabe
CIO-KURZ.C	C-Quelle der unformatierten Ein- Ausgabe (Kurzversion)
CLIB.ASM	Assemblerquelle der Integer-Bibliothek
CLIBZ.ASM	Z80 - Variante für die Integer-Bibliothek
CLIBZ.REL	
CLLIB.ASM	Assemblerquelle der Long-Bibibliothek
CSTART.ASM	
CSTART.CC	Initialisierungsdatei für den Anschluß von LIB.HEX
CTEST.C	C-Quelle Test-, String-, System- und Speicherverwaltungsfunktionen
LIB.HEX	Laufzeitbibliothek (für ASM.COM Kurzversion)
LIB.REL	Laufzeitbibliothek (für MAC80 / L80)
LIBK.REL	Laufzeitbibliothek (Kurzversion)
LIBP.REL	Laufeitbibliothek (für Option .)
STDIO.H	Definitionen für die Ein- Ausgabe
STDIO-KU.RZ	Definitionen für die verkürzte I/O
STDIOP.H	Definitionen zur LIBP.REL
TRACFF.ASM	Assemblerquelle für den Trace
XXXMAIN.MAC	Initialisierung
XXXMAIN.REL	
XXXMAINZ.MAC	
XXPMAIN.REL	Initialisierung zur LIBP.REL
ZAHLEN.C	Beispielprogramme für die Zahlenausgabe
ZBG.C	Beispielprogramm Erklärung in ZBG.DOC
ZBG.DOC	
ZBG2.C	
ZBGDEF.C	
ZBGDEXT.C	