PROGRAMMIEREN IN C

Eine Einführung

Vorlesung an der Universität Gießen
SS 1995

von:
Dr. Kurt Ackermann

E-Mail: **kurt.ackermann@hrz.uni-giessen.de**

INHALTSVERZEICHNIS DES KURSES

1. Einleitung

1.1 Entwicklungsgeschichte der Sprache C

1969 Ken Thomson (Bell Laboratories) erstellt erste Version von UNIX in Assembler => ist nicht portabel !!!

1970 Ken Thomson entwickelt auf einer PDP/7 die Sprache B als Weiterentwicklung der Sprache BCPL, B ist eine typlose Sprache, sie kennt nur Maschinenworte

1974 Dennis M. Ritchie Weiterentwicklung von B zu C, erste Implementation auf einer PDP 11

C ist in den USA weit verbreitet und findet auch bei uns immer mehr Anwendung. Dies hängt zum einen mit den Eigenschaften der Sprache C und zum anderen mit der wachsenden Verbreitung des Betriebssystems UNIX und seiner Derivate zusammen.

Als Quasi-Standard für die Programmiersprache C galt lange Zeit allgemein das folgende Buch:

Brian W. Kernighan und Dennis M. Ritchie
The C programming language"
Prentice Hall, Englewood Cliffs N.Y. (1978).

Die in diesem Buch gemachten Festlegungen bezüglich Syntax und Semantik der Programmiersprache C stellten quasi den kleinsten gemeinsamen Nenner für alle C-Programme dar.

Inzwischen hat sich C zu einer eigenständigen, betriebssystemunabhängigen Programmiersprache entwickelt. Sie ist auf praktisch allen Rechnerplattformen vom PC bis hin zum Supercomputer und unter allen wichtigen Betriebssystemen verfügbar. Im Laufe der Zeit entstanden "C-Dialekte", die zusätzlich die dringende Notwendigkeit einer "richtigen" Standardisierung zeigten.

1988 hat das ANSI-Kommitee X3J11 diesen Sprachstandard für die Programmiersprache C veröffentlicht, der kurz ANSI C genannt wird. Neuere C-Compiler sollten dem ANSI-Standard entsprechen.

Das oben erwähnte Buch von Kernighan und Ritchie gibt es mittlerweile in einer neueren Auflage, die auch den ANSI-Standard als Grundlage beschreibt.

1.2 Kurzdarstellung der Sprache C

C nimmt eine Zwischenstellung zwischen Assembler und Hochsprache ein. Sie vereint zwei an sich widersprüchliche Eigenschaften:

- gute Anpassung an die Rechnerarchitektur (Hardware)
- hohe Portabilität

C-Compiler erzeugen sehr effizienten Code (sowohl bzgl. Laufzeit als auch bzgl. Programmgröße), daher kann für die meisten Anwendungsfälle auf den Einsatz eines Assemblers verzichtet werden. Die einfachere Programmierung bedingt kürze Entwicklungszeiten für die Software und die hohe Portabilität einfachere und vor allem schnellere Anpassungen an eine andere Hardware.

Bei Mikrorechnern ist ein Trend hin zu "maschinennahen" Hochsprachen festzustellen (TURBO-PASCAL, APL). C ist quasi schon von Hause aus maschinennah.

C ist eine vergleichsweise einfache Sprache (Daher nicht ganz einfach anzuwenden!!!). Sie kennt nur 4 Datentypen: **char**, **int**, **float** und **double**. Diese Grunddatentypen können noch qualifiziert werden. Auch zusammengefaßte Datentypen wie Vektoren(Felder) und Strukturen sind möglich, allerdings mit weniger eingebauten Kontrollen als z.B. in PASCAL.

Es gibt einfache Kontrollstrukturen: Entscheidungen, Schleifen, Zusammenfasssungen von Anweisungen (Blöcke) und Unterprogramme.

Ein-/Ausgabe ist nicht Teil der Sprache C sondern wird über Bibliotheksfunktionen erledigt. Weiterhin gibt es keine diffizielen Dateimanipulationen und keine Operationen auf zusammengesetzten Objekten als Ganzes (z.B. Zeichenketten).

Parameter werden als Werte an Funktionen übergeben, daher kann eine Funktion diese nicht ändern. Allerdings gibt es die Möglichkeit von Zeigern (Adressen) als Parametern, deren Zielobjekt dann natürlich geändert werden kann. Zeiger stellen sehr mächtige und vielseitige Anwendungsmöglichkeiten bereit. Sie werden durch eine automatische Adreßarithmetik unterstützt.

Für Datenwandlungen zwischen den einzelnen Typen gibt es kaum Beschränkungen, so daß man hier sehr flexibel ist. Weiterhin gibt es z.B. mittels der Speicherklasse "register" Bezüge zur Hardware.

1.3 Einsatzgebiete

Die Programmiersprache C findet heutzutage eine immer weitere Verbreitung, da sie einerseits fast so flexibel wie ein Assembler ist und andererseits über viele Möglichkeiten heutiger moderner Hochsprachen verfügt. Das hat dazu geführt, daß sich C besonders als Sprache für Betriebssystementwicklungen eignet (für die sie ja auch entwickelt wurde). Weiterhin lassen sich numerische Verfahren, Textverarbeitung und Datenbanken effizient in C realisieren. (Das Datenbankprogramm DBASE ist z.B. in C geschrieben.)

C gehört zu UNIX in dem Sinne, daß UNIX ohne C kaum vorstellbar ist (wohl aber C ohne UNIX). So ist z.B. die Syntax des UNIX Dienstprogrammes **awk** in vielen Dingen identisch mit der von C. Selbst das Lesen und Verstehen der UNIX-Dokumentation ist an vielen Stellen ohne C-Kenntnisse nicht möglich.

Eine Programmierung in C ist immer dann angebracht, wenn sehr portabler Code erzeugt werden soll, da keine andere Sprache (trotz Normung) dazu in dem Maße in der Lage ist wie C.

Mittlerweile gibt es viele verschieden C-Implementationen:

Auf UNIX Systemen heißt der C-Compiler **CC** und es gibt ein Dienstprogramm **LINT**, das ein C-Quellenprogramm unter strengen Maßstäben auf Portabilität prüft.

Für MS-DOS Rechner sind z.B. folgende C Compiler und Interpreter verfügbar: **RUN/C** (Interpreter), **LATTICE** C Compiler, **AZTEK** C Compiler, **ZORLITE** C Compiler, **MICROSOFT C (Quick C)** und **TURBO-C** sowie **C++** von **BORLAND**. Die letzten 3 stellen dabei eine vollständige Programmierumgebung mit Editor und teilweise mit Debugger zur Verfügung (wie z.B. TURBO-PASCAL).

2. COMPILER-INFORMATIONEN

Dieses Kapitel ist das einzige Kapitel, das sich notwendigerweise auf einen bestimmten Rechner und sogar auf einen bestimmten C-Übersetzer beziehen muß, da die Bedienung der einzelnen C-Übersetzer sehr unterschiedlich sein kann. Wir werden hier auch nur C-Compiler betrachten und uns nicht mit C-Interpretern beschäftigen, bei denen die Programmerzeugung etwas anders aussieht.

2.1 WIE ERZEUGE ICH EIN PROGRAMM?

Der grundsätzliche Weg von der Idee zum lauffähigen Programm kann so gekennzeichnet werden:

Editor ---> C-Compiler ---> Linker ---> Programm

Mit dem Editor wird eine **Quellencodedatei** erzeugt, die vom C-Compiler in eine **Objektdatei** übersetzt wird. Aus dieser erzeugt der Linker unter Verwendung der C-Bibliothek dann das **lauffähige Programm**. Oftmals sind die Funktionen von Compiler und Linker in einem Programm zusammengefaßt.

2.1.1 MS-DOS SYSTEM

Auf einem **MS-DOS System** gibt es z.B. die Editoren bzw. Textverarbeitungsprogramme **EDLIN**, **EDIT** und **WORD**. Der Compiler hat je nach Hersteller einen anderen Namen. Zum Linken wird meist der MS-DOS Linker verwendet. Die Dateinamen haben üblicherweise folgende Dateinamenergänzungen:

.c	für den Quellencode
.h	für Include-Dateien
.obj	für den Objektcode
.exe	für das lauffähige Programm

Mit dem Aufkommen von Programmierumgebungen für PCs (Zorlite C, Microsoft C und Turbo C) hat der Benutzer eine komfortable Umgebung zur Erzeugung und zum Austesten von Programmen. Sie sind (fast) selbsterklärend und arbeiten mit Pull-Down-Menüs. Weiterhin verfügen diese Umgebungen über einen integrierten Editor und eine On-Line-Hilfe-Funktion. Sie zeigen Syntaxfehler im Quellencode an. Dies ist insbesondere für Anfänger in der Programmiersprache C sehr wertvoll, da Fehler somit leicht gefunden werden. Meist verfügen Sie auch über einen Debugger, mit dem man das ablaufende Programme an beliebig wählbaren Stellen anhalten, Variablen abfragen und dann weiterlaufen lassen kann. Damit lassen sich auch Laufzeitfehler (insbesondere bei der Zeigerverarbeitung) aufspüren.

2.1.2 UNIX SYSTEM

Auf einem **UNIX System** kann man als Editor z.B. **vi**, **emacs** oder **pine** verwenden. Der C-Compiler schließt hier die Linkfunktion mit ein und heißt **CC**. Der Compiler entspricht meist der ANSI-Norm. Die Dateinamen haben üblicherweise folgende Dateinamenergänzungen:

.c	für den Quellencode
.h	für Include-Dateien
.o	für den Objektcode
a .out	für das lauffähige Programm

Die Bibliotheksdateien befinden sich meist im Verzeichnis */usr/lib* und die Include-Dateien stehen in */usr/lib/include*.

Auf einem UNIX System gibt es weiterhin einen C-Syntaxchecker. Dieses Programm heißt **LINT** und erzeugt keinen Code, unterwirft das Quellenprogramm aber einer ausführlichen Syntaxanalyse, die der normale C-Compiler CC nicht bietet. Ein Programm, das von **LINT** ohne Fehlermeldung bearbeitet wurde, kann als höchst portabel betrachtet werden. Schließlich gibt es, je nach System, auch eine symbolischen Debugger **SDB**. Nähere Informationen zu *cc*, *lint* und *sdb* kann man auf einem UNIX System mit Hilfe des *man*-Kommandos erhalten.

2.2 AUFRUF UND UMGANG MIT DEM COMPILER IN DEN ÜBUNGEN

Der grundlegende Aufruf für den Compiler lautet:

cc *Dateiname*.c

Damit wird das Quellenprogramm in der Datei "Dateiname.c" übersetzt und anschließend automatisch der Linker aufgerufen. Es entsteht im allgemeinen keine Objektdatei (mit der Dateinamenerweiterung .o) sondern nur das fertige Programm in der Datei a.out. Wandelt man den obigen Aufruf wie folgt ab:

cc -o *Dateiname1* *Dateiname2*.c

so wird das fertige Programm in die Datei "Dateiname1" geschrieben. Die Dateinamen "Dateiname1" und "Dateiname2" können auch gleich sien. Die Rechte zur Programmausführung für den Benutzer werden in beiden Fällen automatisch gesetzt.

Es empfiehlt sich, fertige Programme und Quellencodedateien in einem Unterverzeichnis des eigenen Home-Verzeichnisses aufzubewahren, damit man einen besseren Überblick behält.

2.3 INCLUDE-DATEIEN UND BIBLIOTHEKSFUNKTIONEN VON C

Wichtig zur Programmerzeugung sind weiterhin die C-Bibliothek und die Header-(Include-)Dateien. Deren Namen sind, da sie fast in jedem C-Programm benötigt werden, standardisiert. Ebenso natürlich die Namen der Bibliotheksfunktionen. Die folgenden Listen der Include-Dateien und der Bibliotheksfunktionen beziehen sich auf TURBO C 2.0. Sie werden hier angegeben, um Ihnen einen Gesamtüberblick zu geben. Viele Funktionen und Include-Dateien sind aber nicht bei allen C Compilern vorhanden (insbesondere natürlich nicht die MS-DOS spezifischen).
Hier sind zunächst die Include-Dateien aufgelistet. Das Hauptanwendungsgebiet der jeweiligen Include-Datei ist ebenfalls angegeben.

ALLOC.H	Speicherverwaltung
ASSERT.H	Assert-Debugging-Makro
BIOS.H	BIOS-Funktionen
CONIO.H	Ein-/Ausgabe für die DOS-Konsole
CTYPE.H	Zeichenklassifizierung und -umwandlung
DIR.H	Directory-Verwaltung
DOS.H	DOS-Funktionen
ERRNO.H	Mnemonics für Fehlercodes
FCNTL.H	Konstantendefinitionen für OPEN (von Dateien)
FLOAT.H	Parameter für Gleitpunktroutinen
IO.H	Low-Level Ein-/Ausgaberoutinen
LIMITS.H	Informationen zu Grenzwerten versch. Parameter
MATH.H	Mathematik-Funktionen
MEM.H	Speicher-Manipulations-Funktionen
PROCESS.H	Prozeßverwaltung
SETJMP.H	Routinen für FAR-Sprünge
SHARE.H	Parameter für File-Sharing
SIGNAL.H	Abfrage von Betriebssystem-Flaggs
STDARG.H	Makros für variable Anzahl von Funktionsparametern
STDDEF.H	Definition gebräuchlicher Datentypen und Makros
STDIO.H	Standard Ein-/Ausgabe
STDLIB.H	oft beötigte gebräuchliche Funktionen zur Umwandlung, Suche und zum Sortieren von Daten
STRING.H	Zeichenketten-Funktionen
SYS\STAT.H	Symbolische Konstanten für Dateioperationen
TIME.H	Zeit-Funktionen
VALUES.H	wichtige Konstanten einschließlich Maschinenabhängigkeiten

Es folgt nun eine Liste der Bibliotheksfunktionen. Sie sind zu Kategorien zusammengefaßt, um ein schnelleres Suchen nach einer Funktion für eine bestimmte Aufgabe zu ermöglichen. Manche Funktionen kommen mehrfach vor, da sie verschiedenen Kategorien zugeordnet werden können. Die Kategorien sind alphabetisch geordnet; ebenso die Funktionsnamen innerhalb der einzelnen Kategorien.
Die Funktionsnamen geben oft schon einen Hinweis auf ihre Funktion. Weitergehende Informationen erhält man durch das Schreiben des Namens der Funktion, Positionieren des Cursors auf diesen Namen und die Anfrage nach Syntaxhilfe (Funktionstaste LOOKUP). Eine ausführliche Beschreibung der Funktionen findet man im Referenzhandbuch für den jeweiligen Compiler.

Diagnose-Routinen (Prototypen in *assert.h*, *math.h* und *errno.h*)

assert matherr perror

Directory-Routinen (Prototypen in *dir.h*)

```
chdir     findfirst findnext  fnmerge   fnsplit   getcurdir
getcwd    getdisk   mkdir     mktemp    rmdir     searchpath
setdisk
```

Ein-/Ausgabe-Routinen (Prototypen überwiegend in *stdio.h* und *io.h*, aber auch in *conio.h* und *signal.h*)

```
access    cgets     chmod     _chmod    clearerr  _close
close     cprintf   cputs     creat     _creat    creatnew
creattemp dup       dup2      eof       fclose    fcloseall
fdopen    feof      ferror    fflush    fgetc     fgetchar
fgets     filelength          fileno    flushall  fopen    fprintf
fputc     fputchar  fputs     fread     freopen   fscanf
fseek     ftell     fwrite    getc      getch     getchar
getche    getftime  getpass   gets      getw      gsignal
ioctl     isatty    kbhit     lock      lseek     _open
open      perror    printf    putc      putch     putchar
puts      putw      read      _read     remove    rename
rewind    scanf     setbuf    setftime  setmode   setvbuf
sopen     sprintf   sscanf    ssignal   strerror  tell
ungetc    ungetch   unlock    vfprintf  vfscanf   vsprintf
vsscanf   _write    write
```

Interface-Routinen für DOS, BIOS und 8086 (Prototypen überwiegend in *dos.h*, aber auch in *bios.h*)

```
absread   abswrite  bdos      bdosptr   bioscom   biosdisk
biosequip bioskey   biosmemory          biosprint biostime  country
ctrlbrk   disable   dosexterr enable     FP_OFF    FP_SEG
freemem   geninterrupt        getcbrk   getdfree  getdta    getfat
getfatd   getpsp    getvect   getverify harderr   hardresume
hardretn  inport    inportb   int86     int86x    intdos
intdosx   intr      keep      MK_FP     outport   outportb
parsfnm   peek      peekb     poke      pokeb     randbrd
randbwr   segread   setdta    setvect   setverify sleep
unlink
```

Klassifizierungs-Routinen (Prototypen in *ctype.h*)

```
isalnum   isalpha   isascii   iscntrl   isdigit   isgraph
islower   isprint   ispunct   isspace   isupper   isxdigit
```

Konvertierungs-Routinen (Prototypen überwiegend in *stdlib.h*, einige auch in *ctype.h*)

```
atof      atoi      atol      ecvt      fcvt      gcvt
itoa      ltoa      strtod    strtol    toascii   _tolower
tolower   _toupper  toupper   uitoa
```

Mathematik-Routinen (Prototypen überwiegend in *math.h*, aber auch in *float.h* und *stdlib.h*)

```
abs       acos      asin      atan      atan2    atof
atoi      atol      cabs      ceil_     clear87  _control87
cos       cosh      ecvt      exp       fabs     fcvt
floor     fmod      _fpreset87          frexp    gcvt  hypot
itoa      labs      ldexp     log       log10    ltoa
_matherr  matherr   modf      poly      pow      pow10
rand      sin       sinh      sqrt      srand    _status87
strtod    strtol    tan       tanh      uitoa
```

Prozeß-Routinen (Prototypen in *process.h*)

```
abort     execl     execle    execlp    execlpe   execv
execve    execvp    execvpe   _exit     exit      spawnl
spawnle   spawnlp   spawnlpe  spawnv    spawnve   spawnvp
spawnvpe  system
```

Sonstige-Routinen (Prototypen in *setjmp.h*)

```
setjmp    longjmp
```

Speicher-Verwaltungs-Routinen (Prototypen überwiegend in *alloc.h*, aber auch in *stdlib.h* und *dos.h*)

```
allocmem  brk       calloc    coreleft  farcalloc farcoreleft
farmalloc farrealloc          free      malloc    realloc   sbrk
setblock
```

Standard-Routinen (Prototypen in *stdlib.h*)

```
abort     abs       atexit    atof      atoi      atol
bsearch   calloc    ecvt      _exit     exit      fcvt
free      gcvt      getenv    itoa      labs      lfind
lsearch   ltoa      malloc    putenv    qsort     rand
realloc   srand     strtod    strtol    swab      system
uitoa
```

String- und Speicher-Manipulations-Routinen (Prototypen überwiegend in *string.h*, aber auch in *mem.h*)

```
memcpy    memchr    memcmp    memicmp   memmove   memset
movebytes movedata  movmem    setmem    stpcpy    strcat
strchr    strcmp    strcpy    strcspn   strdup    strerror
stricmp   strlen    strlwr    strncat   strncmp   strncpy
strnicmp  strnset   strpbrk   strrchr   strrev    strset
strspn    strstr    strtok    strtol    strtoul   strupr
```

Zeit- und Datums-Routinen (Prototypen in *time.h* und *dos.h*)

```
memcpy    memchr    memcmp    memicmp   memmove   memset
asctime   ctime     difftime  dostounix getdate   gettime
gmtime    localtime setdate   settime   stime     tzset
unixtodos
```

3. BEISPIELPROGRAMME ZUR EINFÜHRUNG

Beispielprogramm P3-1.C

```
main()
{
printf ("Viel Spass mit C !\n");
}
```

Erläuterungen zu diesem Programm:

- C-Programme bestehen aus Funktionen, diese sind an den dem Funktionsnamen folgenden runden Klammern zu erkennen
- Funktion **main()** muß immer vorhanden sein
- sie wird meist am Anfang des Quellencodes plaziert
- die geschweiften Klammern umschließen den Funktionskörper wie z.B. BEGIN und END in PASCAL
- das Ende des Quellencodes muß nicht explizit angegeben werden
- in C gibt es nur Funktionen, keine CALLs
- **printf** ist eine Funktion zur formatierten Ausgabe auf den Bildschirm
- sie hat hier die Stringkonstante **"Viel Spass mit C !\n"** als Parameter
- die Zeichenkombination **\n** steht für das Zeilenendezeichen
- C kennt keine Zeilenstruktur (Ausnahme: innerhalb von Zeichenketten oder #define-Anweisungen)
- Anweisungen sind mit **;** zu beenden

Beispielprogramm P3-2.C

```
main()
{
printf ("Viel");
printf (" Spass");
printf (" mit");
printf (" C !");
printf ("\n");
}
```

Erläuterungen zu diesem Programm:

gleiche Wirkung wie Programm **P3-1.C** aber anderes Programm
kein explizites Zeilenendezeichen, daher Verkettung der Ausgabe
das Zeichen \ wirkt als Fluchtsymbol, es dient zur Darstellung von Sonderzeichen:

```
\t    =  Tabulatorzeichen
\b    =  Backspacezeichen
\"    =  das Zeichen "
\\    =  das Zeichen \
```

Beispielprogramm P3-3.C

```
/* Beispiel 3 */
/* Umwandlung von Fahrenheit in Celsius
   fuer f = 0, 20, ..., 300 */

main()
{
int lower,upper,step;
float fahr,celsius;

lower = 0;  /* untere Grenze der Temperaturtabelle */
upper = 300; /*obere Grenze */
step = 20;   /*Schrittweite */

fahr = lower;
while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf ("%4.0f %6.1f\n",fahr,celsius);
    fahr = fahr + step;
    }
}
```

Erläuterungen zu diesem Programm:

- alle Angaben zwischen **/*** und ***/** werden als Kommentar behandelt
- Kommentar kann überall im Quellencode stehen wo auch ein Leerzeichen, Tabulatorzeichen oder Zeilentrennzeichen stehen darf, sinnvollerweise schreibt man ihn aber an das Ende einer Zeile und/oder verwendet Kommentarzeilen
- in C müssen alle Variablen vor ihrer Verwendung vereinbart werden
- Vereinbarungen müssen am Anfang eines Blocks vor der ersten Anweisung stehen
- man unterscheidet Deklarationen und Definitionen
- C kennt folgende elementaren Datentypen

  char := einzelnes Zeichen (Character)
  int := Integerzahlen
  float := Gleitkommazahlen
  double := Gleitkommazahl mit doppelter Genauigkeit

- zur Qualifizierung dienen dann noch die Zusätze:

  short := kleine Integerzahl
  long := große Integerzahl
  unsigned := ohne Vorzeichen
  signed := mit Vorzeichen (Weglasswert)

- die Zeilen :

  lower = 0;
  upper = 300;
  fahr = lower;

  stellen Zuweisungen dar, dabei wird im 3.Fall eine Typwandlung von **int** nach **float** durchgeführt

```
-       while (fahr <= lower) {
            ...
         }
```

ist eine von mehreren in C realisierten Kontrollstrukturen, die Anweisungen in den geschweiften Klammern werden nur ausgeführt, wenn der Ausdruck in der Klammer nach **while** wahr ist

- hängt nur eine einzige Anweisung von der **while** Bedingung ab, so kann man die geschweiften Klammern auch weglassen:

```
while (i < j)
        i = 2 * j;
```

- in C gilt generell die Konvention:
  gleich 0 := FALSE (FALSCH)
  ungleich 0 := TRUE (WAHR)

- die Formulierung 5.0/9.0 gegenüber 5/9 ist notwendig, da wegen der sonst verwendeten Ganzzahlarithmetik 5/9=0 gilt
- dagegen kann man die Zahl 32 als 32 oder 32.0 schreiben, weil C die Verwendung gemischter Typen in Ausdrücken erlaubt; zum Zwecke der besseren internen Dokumentation sollte man in unserem Falle aber 32.0 verwenden (Gleitkommazahlen)
- **printf** kann mehr als einen Parameter haben und gleichzeitig die Ausgabe formatieren:

```
printf("%4.0f  %6.1f\n",fahr,celsius);
```

bedeutet:
gibt die Variable **fahr** im Gleitkommaformat mit insgesamt 4 Stellen ohne Nachkommastellen aus, füge ein Leerzeichen (Blank) ein, gib dann die Variable **celsius** im Gleitkommaformat mit insgesamt 6 Stellen und einer Nachkommastelle aus und schließe die Ausgabe mit einem Zeilenendezeichen ab

- **printf** kennt noch andere Umwandlungsformate:
  %c := ein einzelnes Zeichen
  %d = Integerzahl
  %f := Gleitkommazahl
  %o := Oktaldarstellung
  %s := Zeichenkette
  %x := Hexadezimaldarstellung
  %% := das %-Zeichen selbst ausgeben

Beispielprogramm P3-4.C

```
/* Beispiel 4 */
/* wie Beispiel 3 aber mit for-Schleife */

main()
{
int fahr;

for (fahr=0; fahr <= 300; fahr=fahr+20)
    printf ("%4.0d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));
}
```

Erläuterungen zu diesem Programm:

- dieses Programm liefert das gleiche Ergebnis, wie das vorige Beispiel, aber es sieht ganz anders aus
- **fahr** ist jetzt eine Integerzahl, dies dient nur der Demonstration der %d Umwandlung
- **printf** akzeptiert auch Ausdrücke für die auszugebenden Parameter
- die **for** Schleife ist eine weitere in C mögliche Kontrollstruktur, sie hat die Form:

```
for (Initialisierung; Bedingung; Reinitialisierung)
```

Beispielprogramm P3-5.C

```
/* Beispiel 5 */
/* wie Beispiel 4 aber mit #define-Konstanten */

#define LOWER 0    /* untere Grenze der Tabelle */
#define UPPER 300  /* obere Grenze der Tabelle */
#define STEP 20    /* Schrittweite */

main()
{
int fahr;

for (fahr=LOWER; fahr <=UPPER; fahr=fahr+STEP)
    printf ("%4.0d %6.1f\n",fahr,(5.0/9.0)*(fahr-32));
}
```

Erläuterungen zu diesem Programm:

- die Verwendung von symbolischen Konstanten garantiert eine einfache Änderungsmöglichkeit
- symbolische Konstanten werden üblicherweise zur einfacheren Identifizierung in **Großbuchstaben** geschrieben
- C **unterscheidet** zwischen Groß- und Kleinschreibung
- nach **#define** Zeilen steht **kein** Semikolon, da **#define** einen Textersatz (also auch mit evtl. Semikolon) bewirkt
- das # muß in der 1. Spalte stehen
- der C-Präprozessor verarbeitet diejenigen Anweisungen, bei denen ein # in der ersten Spalte steht (in unserem Falle hier ist diese Verarbeitung ein Textersatz)

Beispielprogramm P3-6.C

```
/* Beispiel 6 */
/* ein Kopierprogramm */

#include <stdio.h>       /* enthaelt wichtige
                            Konstantendefinitionen */

main()
```

```
{
int c;

c = getchar();
while (c != EOF) {
    putchar(c);
    c=getchar();
    }
}
```

Erläuterungen zu diesem Programm:

- **#include** bewirkt das Einfügen von Quellencode aus einer anderen Datei
- die Hauptanwendung liegt im Einfügen von sogenannten Header-Dateien, die üblicherweise zu jedem C-Bibliothekssystem dazugehören
- eine solche Headerdatei hat die Dateinamenergänzung **.h** und enthält Symbole und Vereinbarungen, die bei bestimmten Anwendungen (z.B. Ein-/Ausgabe, Dateiverarbeitung) immer wieder gebraucht werden
- die spitzen Klammern um den Dateinamen bewirken, daß zunächst das "INCLUDE"-Verzeichnis (bei UNIX Systemen meist */usr/lib/include*) und erst dann das Arbeitsverzeichnis nach der angegebenen Datei durchsucht wird
- auf verschiedenen Rechnern kann **EOF** 0 oder -1 sein, daher die Definition von EOF in einer Headerdatei
- **c** ist **als** int deklariert, um den Wert **EOF** abfragen zu können
- **getchar** ist eine Funktion aus der Funktionsbibliothek, die 1 Zeichen von der Konsole (stdin) einliest
- **putchar** gibt 1 Zeichen auf die Konsole (stdout) aus
- **!=** bedeutet ungleich
- nach der **while** Bedingung sind 2 Anweisungen mittels geschweifter Klammern zu einem Block zusammengefaßt
- man hätte das Programm auch noch kompakter so formulieren können

```
while ((c=getchar()) != EOF)
    putchar(c);
```

  die Klammerung ist sehr wichtig sonst wird **c** der Wert des Vergleiches von **getchar()** mit **EOF** und nicht das gelesene Zeichen zugewiesen

Beispielprogramm P3-7.C

```
/* Beispiel 7 */
/* Programm zum Zaehlen von Eingabezeichen */

#include <stdio.h>

main()
{
long nc;

nc = 0;
while (getchar() != EOF)
    ++nc;
printf ("%ld\n",nc);
}
```

Erläuterungen zu diesem Programm:

- **++nc** ist eine Inkrementoperation für die Variable **nc**, analog ist **--nc** eine Dekrementoperation
- **++nc** und **nc++** sind beide möglich, unterscheiden sich aber
- **++nc** heißt erst inkrementieren, dann Wert von **nc** verwenden
- **nc++** heißt erst Wert von **nc** verwenden, dann inkrementieren
- die Formatangabe **%ld** ist notwendig, da **nc** ein Long-Integerwert ist

Beispielprogramm P3-8.C

```
/* Beispiel 8 */
/* wie Beispiel 7, aber mit for-Schleife
   und anderer Datentyp fuer den Zaehler */

#include <stdio.h>

main()
{
double nc;

for (nc=0;getchar() != EOF; ++nc)
    ;
printf ("%.0f\n",nc);
}
```

Erläuterungen zu diesem Programm:

- die **for**-Schleife in diesem Fall hat keine abhängige Anweisung, daher das einzelne Semikolon in einer neuen Zeile (wichtig zur Dokumentation)
- die Formatangabe **%.0f** bewirkt die Ausgabe eine doppelt genauen Gleitpunktzahl ohne Nachkommastellen

Beispielprogramm P3-9.C

```
/* Beispiel 9 */
/* Programm zum Zaehlen der Eingabezeilen */

#include <stdio.h>

main()
{
int c,nl;

nl = 0;
while ((c=getchar()) != EOF)  /* Klammern sind wichtig */
    if (c == '\n')
        ++nl;
printf("%d\n",nl);
}
```

Erläuterungen zu diesem Programm:

- hier sehen wir eine **if** Kontrollstruktur, wobei die Bedingung wie bei **while** in Klammern gesetzt wird
- die **if** Kontrollstruktur sieht so aus:
  if (Bedingung)
  Anweisung(en)
  else
  Anweisung(en)
  der else Zweig ist optional

- **==** bedeutet identisch
- **'\n'** ist eine Zeichenkonstante, deswegen in Hochkommas eingeschlossen

Beispielprogramm P3-10.C

```
/* Beispiel 10 */
/* Eingabezeilen, -worte und -zeichen zaehlen */

#include <stdio.h>

#define YES 1
#define NO 0

main()
{
int c, nl, nw, nc, inword;

inword = NO;
nl = nw = nc = 0;
while ((c=getchar()) != EOF) {
    ++nc;
    if (c == '\n')
        ++nl;
    if (c==' ' || c=='\n' || c=='\t')
        inword = NO;
    else if (inword == NO) {
        inword = YES;
        ++nw;
    }
}
printf("%d %d %d\n",nl,nw,nc);
}
```

Erläuterungen zu diesem Programm:

- verbundene Anweisungen wie **nl=nw=nc=0;** werden von rechts nach links abgearbeitet, sie sollten nach Möglichkeit vermieden werden
- **||** ist der (logische) ODER-Operator
- **&&** ist der (logische) UND-Operator
- Ausdrücke mit logischen Operatoren werden von links nach rechts bewertet und dann abgebrochen, wenn das Ergebnis feststeht (also der erste Teilausdruck ungleich 0 (bei ODER) oder gleich 0 (bei UND) ist

Beispielprogramm P3-11.C

```
/* Beispiel 11 */
/* Ziffern, Zwischenraeume und andere Zeichen zaehlen */

#include <stdio.h>

main()
{
int c,i,nwhite,nother;
int ndigit[10];

nwhite=nother=0;
for (i=0; i<10; ++i)
    ndigit[i]=0;

while ((c=getchar()) != EOF)
    if (c>='0' && c<='9')
        ++ndigit[c-'0'];
    else if (c==' ' || c=='\n' || c=='\t')
        ++nwhite;
    else
        ++nother;

printf("Zahlen =");
for (i=0; i<10; ++i)
    printf(" %d",ndigit[i]);
printf("\n\"White Space\" = %d, andere = %d\n",
    nwhite,nother);
}
```

Erläuterungen zu diesem Programm:

- Vektoren (Felder) werden durch Anhängen von eckigen Klammern an den Variablennamen definiert
- die **Feldindizes** laufen von **0** bis **N-1**, sie werden nicht abgeprüft
- das Beispielprogramm gilt nur für Zeichencodes mit lückenlosem Aufeinanderfolgen von Ziffern (sowohl in EBCDIC als auch in ASCII erfüllt)
- **char** wird intern in **int** gewandelt, deshalb ist der Ausdruck

  ```
  c-'0'
  ```

  so einfach möglich
- Verkettungen von **if** und **else** sollte man so schreiben:

```
if              oder          if
else                          else if
    if
    else                      else
```

Beispielprogramm P3-12.C

```
/* Beispiel 12 */
/* die Funktion X hoch N */

main()
{
int i;

for (i=0; i<10; ++i)
    printf("%d %d %d\n",i,power(2,i),power(-3,i));
}

power(x,n)    /* x hoch n, n>0 */
int x,n;
{
int i,p;

p=1;
for (i=1; i<=n; ++i)
    p = p * x;
return(p);
}
```

Erläuterungen zu diesem Programm:

- hier haben wir zum ersten Mal eine "richtige" Funktion
- eine Funktion hat immer die Form:
  Name(Parameterliste,*optional***)**
  Parameterdeklarationen,*optional*
  {
  Vereinbarungen
  Anweisungen
  return Funktionswert,*optional*
  }

- die Parameternamen sind nur lokal bekannt
- die Parameterwerte sind durch die Funktion nur lokal nicht aber mit Wirkung für die aufrufende Funktion veränderbar
- die **return** Anweisung bewirkt einen Rücksprung mit der Übergabe des Funktionswertes, die Klammern um den Funktionswert sind syntaktisch nicht notwendig, aber üblich
- fehlt **return**, dann erfolgt der Rücksprung nach der letzten schließenden geschweiften Klammer, der Funktionswert ist dann unbestimmt

Beispielprogramm P3-13.C

```
/* Beispiel 13 */
/* Wie Beispiel 3, aber mit Einlesen der
   Grenzen und der Schrittweite */

#include <stdio.h>

main()
{
int i;
int lower,upper,step;
float fahr,celsius;

i=0;
while (i != 2) {
    printf("Bitte die Grenzen fuer die Berechnung in \
    der Form \"untere - obere\" angeben\n");
    i=scanf("%d - %d",&lower,&upper);
    if (i != 2 || upper<=lower) {
        printf("\n Falsche Eingabe \07\n");
        i = 0;
    }
}

i=0;
while (i != 1) {
    printf("Bitte die Schrittweite eingeben: ");
    i=scanf("%d",&step);
    if (i != 1 || step <= 0) {
        printf("\n Falsche Eingabe \07\n");
        i = 0;
    }
}

fahr = lower;
while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf ("%4.0f %6.1f\n",fahr,celsius);
    fahr = fahr + step;
    }
}
```

Erläuterungen zu diesem Programm:

- wir haben hier das gleiche Programm wie in Beispiel **P3-3.C**, allerdings werden hier einige Daten interaktiv eingelesen
- das Einlesen wird von der Funktion **scanf** bewerkstelligt
- sie hat die gleiche Formatsteuerung wie **printf**
- die Parameter müssen aber Adressen sein (Warum?)
- **&** ist der Adressoperator, Ergebnis ist die Adresse des Objekts
- Funktionswert von **scanf** ist die Anzahl erfolgreich eingelesener Parameter
- alle Angaben in den Anführungsstrichen von **scanf**, die nicht zu Formatangaben gehören, müssen genauso eingegeben werden
- Strings dürfen über das Zeilenende hinausgehen, wenn unmittelbar vor dem Zeilenende ein \ steht
- \" gibt " aus
- \07 gibt das ASCII-Zeichen mit dem Oktalcode 07 (=BELL) aus

4. GRUNDLAGEN

4.1 ELEMENTE DES QUELLENCODES

C Programme bestehen aus "Worten": Schlüsselwörtern, (Objekt)Namen, Konstanten, Strings, Operatoren und anderen Trenn- und Sonderzeichen. Trennzeichen sind BLANK, TAB und Zeilenende. Kommentare wirken auch als Trennzeichen. Der Quellentext ist (fast) formatfrei.

Der Zeichenvorrat beinhaltet folgende Zeichen:

die Klein-und Großbuchstaben:	**a-z**, **A-Z**
die Ziffern:	**0-9**
die Sonderzeichen:	**+ - * / \ = , . ; ? " # % & _ '**
	< > ( ) [ ] { } | ^ ~ @ :

- die Umlaute (ä, ö, ü, Ä, Ö und Ü) sowie das "ß" gehören **nicht** zu den erlaubten Buchstaben
- Klein- und Großschreibung wird strikt unterschieden
- **Schlüsselwörter** müssen **kleingeschrieben** werden
- weiterhin ist folgende Konvention üblich:
  Variablennamen: **Kleinbuchstaben**
  Symbolische Konstanten: **Großbuchstaben**

- Anweisungen werden mit **;** (Semikolon) abgeschlossen (Achtung: Das Vergessen des ; ist einer der häufigsten Anfängerfehler!!!)
- Schlüsselwörter sind:

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

- oft werden auch noch:

asm	**fortran**	**near**	**far**	**huge**	**pascal**

- verwendet
- Bezeichner sind Namen von Objekten (z.B. Variablen- oder Funktionsnamen). Sie dürfen aus Buchstaben, Ziffern und dem Zeichen Underline "_" bestehen, sie müssen mit einem Buchstaben oder Underline anfangen (Hinweis: Namen, die mit Underline anfangen, werden oft auf Systemebene verwendet, daher eigene Namen besser ohne führendes Underline.)
- Variablennamen dürfen je nach C-Compiler unterschiedlich lang sein, man sollte sich aber bei Namen, die als Querbezug zu Programmen in anderen Quellendateien dienen, auf maximal 6 Zeichen beschränken. (Der Linker ist hier die beschränkende Komponente.) Namen, die nur programmintern auftreten haben meist 31 signifikante Stellen.
- zum Zwecke der internen Dokumentation sollten Variablennamen ihren Zweck und ihre Bedeutung erkennen lassen
- Beispiele für gültige Namen (links) und ungültige bzw. nicht zu empfehlende Namen (rechts):

```
prog1                1_label        /* falsch */
was_nun              heutegehenwir  /* ungünstig bei
                                       externen Bezügen */
punkte               wer.ist        /* falsch */
vorsicht            _vorsicht      /* ungünstig */
                     if             /* falsch */
```

- Kommentar kann überall dort stehen, wo auch sonst ein Trennzeichen stehen darf, er muß mit der Zeichenkombination **/*** eingeleitet und mit ***/** abgeschlossen werden (Natürlich darf Kommentar über mehrere Zeilen gehen.)

- sinnvollerweise schreibt man Kommentar an das Ende einer Zeile nach einer Anweisung oder auf eine eigene Zeile
- ein # in der 1.Spalte dient als Kennung für eine nachfolgende Präprozessoranweisung, diese werden vor der eigentlichen Kompilation ausgewertet, besonders wichtig sind hier:

  a) das Einfügen von (Header)Dateien mittels:

```
#include "stdio.h"  /* Suche beginnt im Homeverzeichnis */
#include <stdio.h>  /* Suche beginnt im "include"
                            Verzeichnis*/
```

  b) und die Definition von symbolischen Konstanten, die dann vor der eigentlichen Kompilation mittels Textersatz eingesetzt werden:

```
#define MAXIMUM 100
```

  (das Wort "MAXIMUM" wird durch den Text "100" ersetzt
- Präprozessoranweisungen werden **nicht** mit Semikolon abgeschlossen, da dieses sonst beim Textersatz mit eingefügt wird

4.2 DATENTYPEN

In C gibt es 4 Grunddatentypen; diese sind:

char	ein Zeichen	(meist 1 Byte, z.B. ASCII-Code)
int	ganze Zahl	(2 oder 4 Byte je nach Rechner)
float	Gleitpunktzahl	(meist 4 Byte je nach Rechner)
double	Gleitpunktzahl	(meist 8 Byte je nach Rechner)

Der Wertebereich für diese Grunddatentypen ist **rechnerabhängig**.

Diese Grunddatentypen können durch Voranstellen von "Eigenschaften" noch qualifiziert werden. Wird nur eine Eigenschaft angegeben, so wird als Typ **int** angenommen. Die möglichen Eigenschaften sind:

unsigned	für **char**
unsigned, **long** und **short**	für **int**
long	für **float**

Beispiele:

```
unsigned int
long int
short int
unsigned char
unsigned short int
long
unsigned
long float
```

4.3 DEKLARATIONEN

Alle Variablen müssen vor ihrer Verwendung deklariert werden. Die Deklaration legt den Namen, den Typ und die Speicherklasse der Variablen fest. Deklarationen müssen am Anfang eines Blockes, am Anfang einer Funktion (Parameter) oder außerhalb von Funktionen (globale Variablen) stehen.

Variablentypen haben wir gerade in Punkt 4.2 kennengelernt. Speicherklassen legen fest, wie die Variable behandelt werden soll, insbesondere in welchem Speicherbereich sie angelegt werden soll. Es gibt 4 Speicherklassen:

auto
static
extern
register

auto ist sozusagen die Normalbehandlung und folglich auch der Weglasswert für Variablen, die in Blöcken deklariert werden. Der Weglasswert außerhalb von Funktionen ist **extern** zur Erzeugung globaler Variablen. **static** Variablen in Funktionen behalten ihren Wert zwischen 2 Funktionsaufrufen bei, **auto** Variablen werden jeweils neu angelegt.

Eine Besonderheit in C ist die Speicherklasse **register**. Damit kann man erreichen, daß Variablen (sofern es die Hardware des Rechners erlaubt) in CPU-Registern gehalten werden, was zu einer schnelleren Programmausführung führt.

Beispiele für Deklarationen:

```
auto int i;
static int i;
auto punkte;
auto char c;
char c;
register j;
register int i,j;
```

Gleichzeitig mit der Deklaration kann man die Variablen auch initialisieren:

```
auto int i=1;
static int i=0;
register int i=1,j=100;
```

Hinweis: Eine eingehendere Besprechung der 4 Speicherklassen wird im Zusammenhang mit Funktionen im Kapitel 6.4 vorgenommen.

4.4 KONSTANTEN

4.4.1 GANZZAHLKONSTANTEN

Ganzzahlkonstanten bestehen aus einer Ziffernkette. Sie werden mormalerweise dezimal interpretiert. Eine führende **0** führt zu einer oktalen Interpretation und die führende Zeichenkombination **0x** oder **0X** gilt als Kennung für Hexadezimalzahlen. Dezimalzahlen werden als normale **int**-Typen behandelt, Oktal- und Hexadezimalzahlen als **unsigned**.

Die folgenden Zahlenangaben sind zeilenweise jeweils gleich:

```
   1      01     0x1
  10     012     0xa
4368  010420  0x1110
```

Konstanten vom Typ **long** werden implizit gebildet, wenn die Zahl für **int** zu groß ist. Durch Anhängen von **l** oder **L** an die Konstante kann man eine **long**-Konstante erzwingen.

```
   35000 0104270  0x8868
  35000l0104270l 0x8868l
     10l    012l    0xal
```

4.4.2 GLEITKOMMAKONSTANTEN

Sie haben immer den Typ **double**. Der Dezimalpunkt und/oder die Exponentenangabe **e** oder **E** muß vorhanden sein, so wie man das gewohnt ist:

```
          .1234
         0.1234
         1.345e-10
         3E+3
```

4.4.3 ZEICHENKONSTANTEN

Sie haben den Typ **char** und werden durch Angabe eines Zeichens eingeschlossen in (einfache) Anführungszeichen erzeugt. Ihr Wert ist der numerische Wert des Zeichens im gültigen Zeichensatz (z.B. ASCII).

```
     'a'     /* numerischer Wert 97  (0x61) */
     'F'     /* numerischer Wert 70  (0x46) */
     '1'     /* numerischer Wert 49  (0x31) */
```

Nicht druckbare Zeichen können mit Hilfe des Fluchtsymbols \ (Backslash) nach folgender Tabelle angebeben werden:

Backspace	BS	\b
Tabulator Zeichen	HT	\t
Zeilentrenner	LF	\n
Formfeed	FF	\f
Carriage Return	CR	\r
Fluchtsymbol	\	\\
Anführungszeichen	'	\'
Null Zeichen	NUL	\0
Bitmuster	ddd	\ddd

ddd steht dabei für maximal 3 **Oktalziffern**, damit ist jedes beliebige Zeichen des Zeichensatzes darstellbar. Auch die Fluchtsymbolfolgen müssen natürlich in Anführungszeichen eingeschlossen werden.

4.4.4 ZEICHENKETTENKONSTANTEN

Zeichenkettenkonstanten bestehen aus einer linearen Folge von Einzelzeichen mit einem wichtigen Unterschied: Am Ende einer Zeichenkette wird immer ein NUL-Zeichen ('**\0**') als Endekennung automatisch angehängt. Zeichenkettenkonstanten werden in doppelte Anführungszeichen eingeschlossen. Es gelten die gleichen Fluchtsymbolfolgen wie bei Zeichenkonstanten; zusätzlich gibt es noch die Fluchtsymbolfolge **\"**. Zeichenketten können über Zeilengrenzen gehen, wenn unmittelbar vor dem Zeilentrenner (im Quellencode) das Fluchtsymbol \ steht.

```
'A'                  /* das Zeichen A */
"A"                  /* die Zeichenkette A\0 */
"Dies ist eine Zeichenkette"
"\"Hahaha\" lachte er"
"Diese Zeichenkette piepst \07"
```

4.5 OPERATOREN

4.5.1 ARITHMETIKOPERATOREN

Es gibt 6 arithmetische Operatoren. Sie stehen für die 4 Grundrechenarten (**+ - * /**), die Modulofunktion (Rest nach Division; **%**) und die arithmetische Negation (das Vorzeichen; -).

Vorzeichen	**-**
Multiplikation	*****
Division	**/**
Modulofunktion	**%**
Addition	**+**
Subtraktion	**-**

Die Modulofunktion ist nicht auf **float** oder **double** Operanden anwendbar. Die Rangfolge ist wie gewohnt: *, / und % gleich und höher als + und -. Das Minuszeichen als unärer Operator (Vorzeichen) hat einen höheren Vorrang als *, / und %. Es gibt kein positives Vorzeichen.

Beispiele für die Anwendung arithmetischer Operatoren:

```
int i=3, j=5, k=10;
int m;
float r=3., s=5.;
float u;

m = i+j+k;              /* 18 */
m = k-j-i;              /*  2 */
m = i-k;                /* -7 */
m = k/i;                /*  3 */
m = k/i+j;              /*  8 */
m = k/(i+j);            /*  1 */
m = k*i+j;              /* 35 */
m = k*(i+j);            /* 80 */
m = k%i;                /*  1 */
u = s+r;                /* 8.       */
u = s/r;                /* 1.666... */
u = s*r;                /*15.       */
u = k/r;                /* 3.333... */
u = k/r +s/r;           /* 5.       */
u = k/i + s/i;          /* 4.666... */
u = k%r;                /* nicht erlaubt */
```

Höhere Rechenoperationen wie Wurzelziehen, Potenzieren, Exponentiation, usw. fehlen in C als Operator ganz; sie sind über Bibliotheksfunktionen realisiert. Bei ihrer Verwendung muß in jedem Fall die Inlcude-Datei *math.h* mit eingebunden werden.

Beispielprogramm P4-1.C

```
#define LAUFZEIT 10
#define ANF_KAP 1000.
#define ZI_SATZ 2.5

main()
{
int i;
float anf_kap,end_kap;
printf("\nGuthabenzinsenberechnung\n");
printf("Jahr Anfangskapital Endkapital\n");
anf_kap=ANF_KAP;
for (i=1; i <= LAUFZEIT; i++) {
   end_kap = anf_kap + ZI_SATZ/100 * anf_kap;
   printf("%2d    %8.2f      %8.2f\n",i,anf_kap,end_kap);
   anf_kap = end_kap;
}
}
```

4.5.2 VERGLEICHE UND LOGISCHE VERKNÜPFUNGEN

Für **Vergleiche** stehen folgende Operatoren zur Verfügung:

kleiner	**<**
kleiner gleich	**<=**
größer	**>**
größer gleich	**>=**
gleich (identisch)	**==**
ungleich	**!=**

Die ersten 4 haben untereinander gleichen Rang, stehen aber eine Rangstufe höher als die beiden letzten. Es gibt in C grundsätzlich keinen Datentyp BOOLEAN; WAHR oder FALSCH werden einfach über den numerischen Wert entschieden. Dabei gilt:

```
ungleich  0     WAHR (erhält den Wert eins)
  gleich  0     FALSCH
```

Beispiele für die Verwendung dieser Operatoren:

```
int a=5;
int b=12;
int c=7;

   a < b              /* WAHR   */
   b < c              /* FALSCH */
   a+c <= b           /* WAHR   */
   b-a >= c           /* WAHR   */
   a == b             /* FALSCH */
   a+c == b           /* WAHR   */
   a != b             /* WAHR   */
   a = b<c;           /* möglich: a=0 */
   a = c<b;           /*   -"-    a=1 */
```

Unter die **logischen Verknüpfungen** fallen die Operatoren:

logische Negation	**!**

```
logisches UND        &&
logisches ODER       ||
```

Sie sind hier in ihrer Rangfolge geordnet aufgeführt, logisches UND hat also einen höheren Vorrang als das logische ODER. In Ausdrücken erfolgt die Abarbeitung von links nach rechts aber nur solange, bis das Ergebnis eindeutig feststeht. Das führt zu einer kürzeren Ausführungszeit, wenn man die häufigst verwendete Bedingung an den Anfang einer Bedingungsabfrage stellt. Bei Operationen, die Nebeneffekte haben können (z.B. Inkrementation), muß man allerdings vorsichtig sein.

```
if (a<b            &&  (d=(a+b)) != c)   /* diese 2 Zeilen */
if ((d=(a+b)) != c &&  a<b)              /* wirken nicht gleich */
```

4.5.3 INKREMENT- UND DEKREMENTOPERATOREN

C kennt spezielle Operatoren zum In- und Dekrementieren. Es sind dies:

```
Inkrement          ++
Dekrement          --
```

Sie sind (wie der Vorzeichenoperator -) rechts assoziativ, werden also von rechts her zusammengefaßt. Sie können **vor** oder **nach** ihrem Operanden stehen. Im **ersten Fall** wird der Operand **zunächst in-** oder **dekrementiert** und dann weiterverwendet, im **zweiten Fall** wird der Operand **erst verwendet** und dann in- oder dekrementiert.

```
int i;
int k;

i=1;

if (++i > 1)
    k = 5;            /* wird durchlaufen */
if (i++ > 2)
    k = 10;           /* wird nicht durchlaufen */
printf ("\nk hat den Wert %d\n",k);    /* Ausgabe: 5 */



int i,k;
int j;

i=2;
k=3;

j = i+++k;            /* i+ (++k); i=2,k=4,j=6  */
j = (i++)+k;          /*           i=3,k=4,j=6  */
j = i++ +k;           /*           i=4,k=4,j=7  */
j = (i+k)++;          /* ist verboten           */
```

Beispielprogramm P4-2.C:

```
strcat(s,t)      /* String t wird an String s angehängt */
char s[],t[];
{
   int i,j;
   i=j=0;

   while (s[i] != '\0')
      i++;
   while ((s[i++]=t[j++]) != '\0')
      ;
}
```

Beispielprogramm P4-3.C:

```
squeeze(s1,s2)   /* Entfernt aus String s1 alle Zeichen, */
                 /* die in String s2 vorkommen */
char s1[],s2[];
{
   int i,j,k;

   for (i=j=0; s1[i] != '\0'; i++) {
      s1[j++] = s1[i];
      for (k=0; s2[k] != '\0'; k++)
         if (s1[i] == s2[k]) {
            j--;
            break;
         }
   }
   s1[j] = '\0';
   return (j);     /* Anzahl Zeichen im String */
}
```

Beispielprogramm P4-4.C:

```
any(s1,s2)    /* Gibt die Position des ersten Zeichens in
                 s1 an, das in s2 enthalten ist */
char s1[],s2[];
{
   int i,j,n;

   for (i=0,n=-1; s1[i]!='\0' && n==-1; i++)
      for (j=0; s2[j] != '\0'; j++)
         if (s1[i] == s2[j]) {
            n = i;
            break;
         }
   return (n);
}
```

4.5.4 BITMANIPULATIONEN

C stellt auch Operatoren für Bitmanipulationen zur Verfügung. Sie können **nicht** auf Variablen oder Konstanten vom Typ **float** oder **double** angewendet werden. Die Operatoren sind wie folgt:

Komplement	**~**
Linksshift	**<<**
Rechtsshift	**>>**
bitweises UND	**&**
bitweises EXCLUSIVES ODER	**^**
bitweises ODER	**|**

Beispiele:

```
int i=7;
int j=9;
int k;

k = i & j;        /* k=1   */
k = i | j;        /* k=15  */
k = i ^ j;        /* k=14  */
k = ~0            /* k=max(unsigned int) */
k = ~i;           /* k=max(unsigned int) -7 */
k = i << 3;       /* k=56  */
k = i >> 3;       /* k=0   */
k = i & 0x03;     /* k=3; alle Bits bis auf die beiden */
                  /* unteren ausmaskiert */
```

Beispielprogramm P4-5.C:

```
getbits(x,p,n)  /* n Bits ab Position p rechtsbündig */
                /* ... 7 6 5 4 3 2 1 0               */
unsigned x;
int p,n;
{
   return ((x >> (p+1-n)) & ~(~0<<n));
}
```

Beispielprogramm P4-6.C:

```
wordlength()    /* Bestimmt die Bitlänge von int */
{
   int i,j;     /* oder: unsigned i; int j; */

   i = ~0;
   j = 0;
   while (i) {
      j++;
      i = i << 1;
   }
   return (j);
}
```

Beispielprogramm P4-7.C:

```
bitcount(n)  /* zählt die Anzahl der 1 Bits in n */
unsigned int n;
{
   int b;
   for (b=0; n!=0; n >>= 1)
      if (n & 01)
         b++;
   return (b);
}

bitcount(n)  /* 2. schnellere Version */
unsigned int n;
{
   int b;
   b=0;
   while (n != 0) {
      b++;
      n = n & (n-1);
   }
   return (b);
}
```

4.5.5 BEDINGTE BEWERTUNG

C kennt einen Operator, der eigentlich aus einem Operatorenpaar (**? :**), besteht. Es handelt sich dabei um die bedingte Bewertung. Sie hat die Form:

```
Ausdruck1 ? Ausdruck2 : Ausdruck3
```

Ist **Ausdruck1** WAHR, dann wird **Ausdruck2** ausgeführt, ansonsten **Ausdruck3**. Die bedingte Bewertung entspricht daher folgender **if-else** Konstruktion:

```
if (Ausdruck1)
   Ausdruck2;
else
   Ausdruck3;
```

Beispiele:

```
/* Maximum von a und b */
z = (a > b) ? a : b;


/* Ausgabe eines Feldes, wobei immer nach 10 Zahlen
   eine neue Zeile angefangen werden soll */
for (i=0; i < N; i++)
   printf ("%6d%c",a[i],(i%10==9||i==N-1)?'\n':' ');
```

4.6 AUSDRÜCKE UND ZUWEISUNGEN

Ein Ausdruck besteht aus Operanden und Operatoren oder einem Funktionsaufruf. Ein Ausdruck gefolgt von einem Semikolon ist eine Anweisung. Zur Konstruktion gültiger Ausdrücke bzw. Anweisungen sind noch einige Dinge zu klären, die bisher noch nicht besprochen wurden.

4.6.1 OPERATORRANGFOLGE

Bis zu diesem Zeitpunkt haben wir die meisten Operatoren besprochen. Zu ihrer richtigen Anwendung gehört nicht nur das Wissen um ihre Funktion, sondern auch das um ihre Rangfolge und ihre Assoziativität. Diese Angaben sind in der folgenden Tabelle für alle Operatoren zusammengestellt, auch für die bisher noch nicht besprochenen. Die Operatoren in der Tabelle sind ihrer Rangfolge nach von oben (höchster Rang) nach unten (niedrigster Rang) angeordnet. Alle Operatoren, die auf einer Zeile stehen haben gleiche Rangfolge und können bei einem gemeinsamen Auftreten in einem Ausdruck vom Compiler in beliebiger Reihenfolge bewertet werden.

Operator	Zusammenfassung (Assoziativität)
() [] -> .	von links her
! ~ ++ -- - (*typ*) * & sizeof	von rechts her
* / %	von links her
+ -	von links her
<< >>	von links her
< <= > >=	von links her
== !=	von links her
&	von links her
^	von links her
|	von links her
&&	von links her
||	von links her
?:	von rechts her
= += -= += *(etc.)*	von rechts her
,	von links her

Auf folgende Dinge wird hier noch einmal besonders hingewiesen:

- es werden teilweise die gleichen Operatorzeichen für verschiedene Operatoren verwendet
  - **()** 1. Klammerung 2. type cast
  - ***** 1. Pointerdekl. 2. Multiplikation
  - **-** 1. Vorzeichen 2. Subtraktion
  - **&** 1. Adresse von 2. bitweises UNDD
- In- und Dekrementoperatoren haben einen sehr hohen Vorrang
- die Arithmetikoperatoren haben einen Vorrang wie erwartet und gewohnt
- Shifts haben einen höheren Vorrang als Vergleiche, im Gegensatz zu den anderen Bitmanipulationen, die nach den Vergleichen stehen
- bitweise logische Operationen stehen vor den logischen Operationen, die sich auf ganze Zahlen beziehen
- die bedingte Bewertung hat einen sehr niedrigen Vorrang, trotzdem sollte man sich bei ihrer Anwendung eine Klammerung angewöhnen, damit die Anweisung leichter lesbar ist.
- außer durch den Vorrang ist die Reihenfolge der Bewertung undefiniert: **z = (a*2) + ++a** kann als **(a*2)+ ++a** aber auch als **++a + (a*2)** bewertet werden. Man nennt dies einen Nebeneffekt. Diese müssen aus naheliegenden Gründen vermieden werden.

- ebenso ist die Reihenfolge der Bewertung von Funktionsparametern nicht gewährleistet:
```
i=1;
printf("%d %d",++i,i*3)   /* kann 2,6 oder 4,3 ausgeben */
```

- ebenfalls nicht eindeutig definiert ist folgende Konstruktion:
```
a[i]=i++;
```

4.6.2 ZUWEISUNGEN

- die linke Seite einer Zuweisung muß ein einfacher Ausdruck und ein sogenannter L-Wert sein
- einfache Ausdrücke bezeichnen **ein einzelnes** Objekt, dieses Objekt kann eine Variable, eine Konstante, ein Funktionsaufruf, oder ein Feld-, Struktur-, Aufzählungs- oder Varianten**element** sein.
- L-Werte sind Objekte, die veränderbar sind z.B. Variablen
- **keine** L-Werte sind: Konstanten, Feldnamen ohne folgende Indexangabe, Funktionsnamen, Namen aus einer Aufzählung, Objekte, die explizit mit *const* als konstant vereinbart wurden
- Zuweisungen können in Ausdrücken an der Stelle eines Operanden stehen (z.B.: while ((c=getchar()) != EOF)
- Zuweisungen in einer Zuweisungskette werden von rechts nach links ausgeführt:
```
a = b = c = 1;     /* alle 1 */
```

- mit Hilfe des Kommaoperators können Listen von Anweisungen gebildet werden, sie werden von links nach rechts abgearbeitet
```
a=b, b=c, c=1;    /* nur c=1, andere unbestimmt */
c=1, b=c, a=b;    /* alle 1 */
```

- Ausdruckslisten werden oft im Zusammenhang mit einer **for**-Schleife verwendet, um z.B. in der Reinitialisierung 2 verschiedene Operationen durchzuführen:
```
for (i=0,j=10;i < N; i++,j++)
```

- Vorsicht ist allerdings dann geboten, wenn das Komma wie z.B. in einer Funktionsparameterliste noch eine andere Funktion hat, durch eine Klammerung kann man aber jederzeit das Gewünschte erreichen:
```
printf("Was wird hier %d ausgeben ?",(i=1,i=i+10));
```

4.6.3 DATENTYPWANDLUNGEN

- Datentypwandlungen sind immer dann notwenig, wenn 2 Operanden in einem Ausdruck verschiedene Typen haben
- Datentypwandlungen werden soweit notwendig implizit durchgeführt
- **char** wird bei Bewertungen immer in **int** gewandelt, dadurch sind **int** und **char** beliebig mischbar
- allerdings kann die **char-int** Wandlung rechnerabhängig sein, **char** kann auf die Zahlenwerte -128 bis +127 oder 0 bis +255 abgebildet sein, die "normalen" Zeichen liegen aber immer im positiven Bereich
- die **char-int** Wandlung ist vor allem bei der Abfrage eines eingelesenen Zeichens auf **EOF** wichtig (Eingelesen werden immer **int** Werte, die dann aber einfach **char** Werten zugewiesen werden können
- die Wandlung von **int** nach **long** erfolgt durch Vorzeichenerweiterung (bei signed) oder Voranstellen von Nullen (unsigned), bei der umgekehrten Wandlung werden die höchstwertigen Bits einfach abgeschnitten
- **float** wird bei Bewertungen immer in **double** gewandelt, alle Rechnungen erfolgen daher mit derselben großen Genauigkeit
- die Wandlung von **int** nach **float** ist wohldefiniert, die umgekehrte Richtung kann rechnerabhängig sein, insbesondere bei negativen Gleitkommazahlen

- den genauen Ablauf der Wandlungen bei der Auswertung eines Ausdrucks oder einer Zuweisung beschreiben folgende Regeln (übliche arithmetische Umwandlungen):

  zuerst werden **char** oder **short** Operanden in **int** Werte und **float** Operanden in **double** Werte umgewandelt

  hat jetzt ein Operand den Typ **double**, so wird der andere ebenfalls in **double** gewandelt und das Resultat ist auch **double**

  ist stattdesssen einer der Operanden **long**, so wird der andere auch in **long** gewandelt und das Resultat ist auch **long**

  ist andernfalls einer der Operanden **unsigned**, so wird auch der andere in **unsigned** gewandelt und das Resultat ist ebenfalls **unsigned**

  liegt keiner dieser Fälle vor, so handelt es sich um **int** Werte, diesen Typ hat dann natürlich auch das Resultat

- durch eine sogenannte **type cast** Operation kann man in einem Ausdruck den Typ eines Operanden auf den geünschten Typ abändern, dies Operation ändert den Typ nur für diese eine Operation:

```
int n;
float r;
r=sqrt((double)n);    /* sqrt benötigt double Parameter */
```

- einige Beispiele für implizite Typwandlungen haben wir schon in Kapitel 4.5.1 bei den Arithmetikoperatoren kennengelernt, hier sind noch einige mehr, insbesondere zur **char-int** Wandlung

Beispielprogramm P4-8.C:

```
atoi(s)     /* wandelt die nächsten x Ziffern aus dem
               String s in eine Integerzahl */
char s[];
{
   int i,n;
   n=0;

   for (i=0; s[i]>='0' && s[i]<='9'; i++)
      n = 10*n + s[i]-'0';
   return (n);
}
```

Beispielprogramm P4-9.C:

```
htoi(s)     /* wie atoi, aber für Hexadezimalzahlen */
char s[];
{
   int i,n;
   n=0;

   for (i=0; ; i++)
      if (s[i] >= '0' && s[i] <= '9')
         n = 16*n + s[i]-'0';
      else if (s[i] >= 'A' && s[i] <= 'F')
         n = 16*n + s[i]-'A'+10;
      else if (s[i] >= 'a' && s[i] <= 'f')
         n = 16*n + s[i]-'a'+10;
      else
         break;
   return (n);
}
```

4.6.4 ZUSAMMENGESETZTE OPERATOREN

-- Eine Spezialität von C ist die abgekürzte Schreibweise für bestimmte Zuweisungen
-- diese abgekürzte Schreibweise ist vorteilhaft, wenn die linke Seite eine komplizierte Struktur hat (weniger Tippfehler, bessere Lesbarkeit)
-- bei der abgekürzten Schreibweise gilt folgendes:

```
Ausdruck1  op= Ausdruck2
```

ist äquivalent zu (beachten Sie die Klammern!):

```
Ausdruck1  =  (Ausdruck1) op (Ausdruck2)
```

- **op** kann einer der Operatoren: **+ - * / % << >> & ^** oder **|** sein
- die Klammern sind sehr wichtig, damit keine unerwünschten Nebeneffekte auftreten:

```
i *= k+1;      /* wird wie */
i = i * (k+1); /* behandelt und nicht wie */
i = i * k +1;
```

- der Vorteil der abgekürzten Schreibweise wird an folgendem Beispiel deutlich:

```
feld[i*3+j-7] = feld[i*3+j-7] + 10;     /* normal */
feld[i*3+j-7] += 10;                    /* abgekürzt */
```

5. KONTROLLSTRUKTUREN

Kontrollstrukturen dienen dazu, ein Problem übersichtlich und strukturiert zu formulieren. Die wichtigsten Kontrollstrukturen sind die Auswahl (Entscheidungen) und die Schleifen. Dazu kommen noch einige andere Kontrollanweisungen.

5.1 AUSWAHL

5.1.1 "IF" ANWEISUNG

- die "if"-Anweisung dient zur Auswahl unter genau zwei Möglichkeiten
- sie hat folgenden Aufbau

```
if (Ausdruck)
   Anweisung(en)1
else
   Anweisung(en)2
```

- der **else** Teil ist optional, kann also auch fehlen
- hinter (Ausdruck) steht kein Semikolon
- mehrere abhängige Anweisungen müssen in geschweifte Klammern eingeschlossen werden (Blockbildung)
- Ausdruck wird, wenn es sich nicht um eine Bedingung handelt, numerisch bewertet (also gleich 0 = FALSCH und ungleich 0 = WAHR), daher sind folgende 2 **if** Konstruktionen äquivalent, bei der Entscheidung, welche man verwenden will, sollte man auf gute Dokumentation und leichte Lesbarkeit achten

```
if (i%k)
   printf("Division mit Rest\n");

if (i%k != 0)
   printf("Division mit Rest\n");
```

- noch 2 Beispiele zur Verwendung von **if** und **if-else**:

```
if (punkte > 50)
   printf ("Bestanden\n");


if (punkte > 50)
   printf ("Bestanden\n");
else
   printf ("Nicht Bestanden\n");
```

- es können auch **else-if** Ketten gebildet werden, **else** gehört dabei immer zum jeweils vorangehenden **if**, ggf. müssen daher Anweisungen mittels geschweifter Klammern zusammengefaßt werden

```
if (punkte > 50)
   printf("Note 1\n");
else if (punkte > 40)
   printf("Note 2\n");
else if (punkte > 30)
   printf("Note 3\n");
else if (punkte > 20)
   printf("Note 4\n");
else
   printf("Nicht Bestanden\n");
```

Im folgenden Programmstück steckt ein Fehler:

```
if (n > 0)
   for (i=0; i > n; i++)
      if (s[i] > 0) {
         printf("...");
         return (i);
      }
else     /* hier ist ein Fehler */
   printf("Fehler - n ist kleiner gleich 0\n")
```

Hier ist es nun richtig:

```
if (n > 0) {
   for (i=0; i > n; i++)
      if (s[i] > 0) {
         printf("...");
         return (i);
      }
}
else     /* jetzt ist es richtig */
   printf("Fehler - n ist kleiner gleich 0\n")
```

Beispielprogramm P5-1.C

```
binary(x,v,n)     /* binäres Suchen von x im sortierten
                     Feld v[0] ... v[n-1] */
int x,v[],n;
{
   int low, high, mid;

   low = 0;
   high = n-1;
   while (low <= high) {
      mid = (low + high) / 2;
      if (x < v[mid])
         high = mid - 1;
      else if (x > v[mid])
         low  = mid + 1;
      else      /* gefunden */
         return (mid);
   }
   return (-1);    /* Fehlerkennung */
}
```

5.1.2 "SWITCH" ANWEISUNG

Zur Mehrfachauswahl ist es anstelle von **else-if** Ketten oft günstiger die Kontrollstruktur, die für eine solche Mehrfachauswahl vorgesehen ist, zu verwenden. In C heißt diese Kontrollstruktur **switch-case**. Sie hat folgenden Aufbau:

```
switch (Ausdruck) {
   case Konstante1 : Anweisung(en)1
   case Konstante2 : Anweisung(en)2
   case Konstante3 : Anweisung(en)3
                .
                .
                .
   default:          Anweisung(en)
}
```

- die Konstanten müssen ganzzahlig bzw. Zeichenkonstanten sein
- **default** ist optional und wird durchlaufen, wenn keine der anderen Bedingungen zutrifft.
- es können beliebig viele **case**-Teile vorhanden sein
- die Abarbeitung und Überprüfung geht vom ersten **case**-Teil bis zum letzten
- nach der 1. Übereinstimmung wird der gesamte Rest der **switch**-Anweisung durchlaufen, insbesondere auch alle Anweisungen der nachfolgenden **case**-Teile !!!
- daher wird man normalerweise immer eine **break**-Anweisung am Ende eines **case**-Teils verwenden

- hier ist noch einmal das Beispiel mit der **else-if** Kette von oben, jetzt aber mittels **switch** gelöst:

```
punkte = ((punkte-1)/10)*10;  /* erzeugt: 0, 10, 20, 30 .. */
switch (punkte) {
   case 50:
      printf("Note 1\n");
      break;
   case 40:
      printf("Note 2\n");
      break;
   case 30:
      printf("Note 3\n");
      break;
   case 20:
      printf("Note 4\n");
      break;
   default:
      printf("Nicht bestanden\n");
}
```

Beispielprogramm P5-2.C

```
main()       /* ein Menuegeruest */
{
int c;
printf("     HAUPTMENUE\n\n");
printf("A    Assembler\n");
printf("C    C-Compiler\n");
printf("D    DBASE III\n");
printf("T    Turbo-PASCAL\n");
printf("\nBitte geben Sie Ihre Wahl ein: ");

switch (c=getchar()) {
   case 'a':
   case 'A':
      printf("\nSie haben Assembler gewählt\n");
      break;
   case 'c':
   case 'C':
      printf("\nSie haben den C-Compiler gewählt\n");
      break;
   case 'd':
   case 'D':
      printf("\nSie haben DBASE III gewählt\n");
      break;
   case 't':
   case 'T':
      printf("\nSie haben Turbo-PASCAL gewählt\n");
      break;
   default:
      printf("\nFalsche Wahl !!! \07\n");
}
}
```

5.2 SCHLEIFEN

5.2.1 "WHILE" ANWEISUNG

- die **while**-Anweisung realisiert eine Schleife in der meist benötigten Art mit der Prüfung der Schleifenbedingung **vor** jedem (auch dem ersten) Schleifendurchlauf
- sie sieht so aus:

```
   while (Ausdruck)
      Anweisung(en)
```

- die Schleife wird solange wiederholt, wie der Ausdruck WAHR d.h. ungleich 0 ist
- eine (so natürlich nicht sinnvolle) unendliche Schleife wäre:

```
   while(1)
      Anweisung(en)
```

Beispiele für die Verwendung von while-Schleifen:

```
int c;
while ((c=getchar()) != EOF)
   printf("Zeichen war nicht EOF %c\n",c);
printf("Zeichen war EOF\n");
```

```
int c;
char s[80];
i=0;
while ((c=getchar()) != EOF && i < 80-1) {
   s[i]=c;           /* oder: s[i++]=c;   das erspart
   i++;                 die geschweiften Klammern */
```

```
    }
    s[i]='\0';
```

5.2.2 "DO-WHILE" ANWEISUNG

- die **do-while** Schleife überprüft die Schleifenbedingung **nach** jedem einzelnen Schleifendurchlauf
- eine solche Schleife wird also immer mindestens einmal durchlaufen
- dieser Anwendungsfall ist selten im Vergleich mit der **while**-Schleife (nur etwa 5% der Schleifen sind **do-while**-Schleifen)
- die Struktur der "**do-while**" Anweisung ist wie folgt:

```
do
   Anweisung(en)
while (Ausdruck) ;
```

- beachten Sie das Semikolon

Beispiel:

```
main()     /* "Lobmaschine" */
{
int c;
do {
   printf("\nDas hast Du prima gemacht !");
   printf("\nNochmal wiederholen (J/N) ?");
   c=getchar(); }
while (c=='j' || c=='J') ;
}
```

5.2.3 "FOR" ANWEISUNG

- eine in C sehr häufig verwendete Schleife ist die **for**-Schleife
- sie entspricht einer **while**-Schleife mit Initialisierung und Abschlußanweisung(en)
- die Struktur der Anweisung sieht so aus:

```
for (Ausdruck1; Ausdruck2; Ausdruck3)
   Anweisung(en)
```

- **Ausdruck1** stellt die Initialisierung dar, er wird nur einmal vor dem ersten Schleifendurchlauf abgearbeitet
- **Ausdruck2** ist die Schleifenbedingung, die vor jedem Schleifendurchlauf bewertet wird
- **Ausdruck3** ist die Reinitialisierung der Schleife, er wird am Ende der Schleife unmittelbar vor der Bewertung der Bedingung ausgeführt
- jeder der 3 Ausdrücke kann fehlen, der Weglasswert für den Ausdruck2 ist WAHR!
- Ausdrücke können auch Ausdruckslisten (Kommaoperator) sein

- äquivalent zur **for**-Schleife ist folgende Konstruktion mit **while**:

```
Ausdruck1;
while(Ausdruck2) {
   Anweisung(en)
   Ausdruck3;
}
```

- die folgende **for**-Anweisung realisiert eine unendliche Schleife:

```
for (;;)
   Anweisung(en)
```

- die übliche Schleife für die Abarbeitung eines eindimensionalen Feldes (Vektor) sieht so aus:

```
for (i=0; i<N; i++)
```

- rückwärts:

```
for (i=N-1; i>=0; i--)
```

Beispielprogramm P5-3.C (siehe auch P4-8.C)

```
atoi(s)     /* Ziffern der Zeichenkette s in int wandeln */
char s[];
{
   int i,n,sign;

   for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)
      ;     /* führende Zwischenräume überlesen */
   sign = 1;
   if (s[i]=='+' || s[i]=='-')     /* Vorzeichen */
      sign = (s[i++]=='+') ? 1 : -1;
   for (n=0; s[i] >='0' && s[i] <='9'; i++)
      n = 10*n + s[i] - '0';
   return (sign*n);
}
```

Beispielprogramm P5-4.C

```
shell(v,n)   /* Shell Sort (aufsteigend) für den Vektor v */
int v[],n;
{
   int gap, i, j, temp;

   for (gap=n/2; gap > 0; gap /=2)
      for (i=gap; i<n; i++)
         for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
            temp = v[j];
            v[j] = v[j+gap];
            v[j+gap] = temp;
         }
}
```

Beispielprogramm P5-5.C

```
reverse(s)     /* Vektor s umkehren */
char s[];
{
   int c, i, j;

   for (i=0,j=strlen(s)-1; i<j; i++,j--) {
      c    = s[i];
      s[i] = s[j];
      s[j] = c;
   }
}
```

5.3 VORZEITIGER ABBRUCH UND SPRÜNGE

5.3.1 "CONTINUE" ANWEISUNG

- dient zum vorzeitigen **Abbruch eines Schleifendurchlaufes** der innersten **while**-, **do-while**- oder **for**-Schleife
- bei **while** und **do-while** wird dann sofort die Schleifenbedingung geprüft, bei **for** wird die Reinitialisierung vorgenommen
- die "**continue**" Anweisung ist für C nicht unbedingt notwendig, ihre Wirkung kann immer auch ohne continue durch eine andere Schachtelung erreicht werden

Beispiel:

```
/* mit continue */
for (i=0; i<N; i++) {
   if (a[i] < 0)     /* negative Elemente überspringen */
      continue;
   ...               /* nicht negative Elemente bearbeiten */
}


/* ohne continue */
for (i=0; i<N; i++) {
   if (a[i] > 0) {   /* negative Elemente überspringen */
   ...               /* nicht negative Elemente bearbeiten */
   }
}
```

5.3.2 "BREAK" ANWEISUNG

- diese Anweisung haben wir schon im Zusammenhang mit der **switch**-Anweisung kennengelernt
- sie dient zum **Verlassen** der innersten **while**-, **do-while**- oder **for**-Schleife, bzw. der **switch**-Anweisung

Beispielprogramm P5-6.C

```
#define MAXLINE 1000

main()     /* entfernt Zwischenraeume am Zeilenende */
{
int n;
char line[MAXLINE];

while ((n=getline(line,MAXLINE)) > 0) {
   while (--n >= 0)
      if (line[n] != ' ' && line[n] != '\t'
          && line[n] != '\n')
         break;
   line[n+1] = '\0';
   printf("%s\n",line);
}
}
```

5.3.3 EXIT() FUNKTION

- die **exit()** Funktion dient zum Verlassen (Beenden) des gesamten Programmes
- sie hat die Form:
  exit (Parameter**)** ;

- **Parameter** muß vom Typ **int** sein, der Zahlenwert wird vom Betriebssystem bzw. allgemeiner vom Vaterprozeß ausgewertet (bei MS-DOS mit ERRORLEVEL)
- es gilt die Konvention: Parameter = 0 ==> Programmlauf okay, sonst Parameter gleich Fehlernummer

Beispielprogramm P5-7.C

```
main()     /* Ja-Nein Abfrage */
{
int c;
printf("\nWeitermachen (J/N) ? ");
c = getchar();
if (c=='j' || c=='J')
   exit(0);
else
   exit(1);
}
```

5.3.4 SPRÜNGE

- in seltenen Fällen kann es angebracht sein, Sprünge zu verwenden
- C stellt dazu eine Sprunganweisung zur Verfügung:

```
    goto  marke;          oder            marke:
       .                                       .
       .                                       .
       .                                       .
    marke:                                goto  marke;
```

- **marke** ist ein frei gewählter Name
- das Sprungziel muß sich in derselben Funktion wie die Sprunganweisung befinden
- vor jeder Anweisung dürfen beliebig viele Sprungmarken stehen
- **Sprünge sollten vermieden werden** !!!

6. FUNKTIONEN

- Funktionen sind Hilfsmittel, um Problemstellungen in kleine Teilprobleme zerlegen zu können
- sie dienen damit einer strukturierten Programmierung
- typische Anwendungen für Funktionen ist weiterhin die Erledigung immer wiederkehrender Aufgaben
- für die wichtigsten Aufgaben gibt es bereits Funktionen, sie sind in der C-Programmbibliothek enthalten
- für die eigenen Zwecke kann man natürlich auch eigene Funktionen schreiben
- C Programme bestehen typischerweise aus vielen kleinen und nicht aus wenigen großen Funktionen

6.1 AUFBAU EINER FUNKTION

- eine Funktion hat einen festgelegten Aufbau, der wie folgt aussieht:

```
name ( Parameterliste, optional )
{
    Vereinbarungen
    Anweisungen
    return Funktionswert,optional
}
```

- die runden Klammern müssen stehen, damit **name** zur Funktion wird (Zwischenraum zwischen **name** und **(** ist erlaubt
- die Parameterliste in den runden Klammern ist optional, d.h. sie muß nur vorhanden sein, wenn der Funktion wirklich Parameter übergeben werden
- die Parameterliste enthält sowohl die Namen als auch die Typdeklarationen der Parameter
- die Gesamtheit der Vereinbarungen und Anweisungen der Funktion selbst nennt man den Funktionskörper, er muß durch geschweifte Klammern eingeschlossen sein
- die **return**-Anweisung darf an beliebiger Stelle im Funktionskörper stehen, mit ihr erfolgt der Rücksprung in die aufrufende Funktion
- die **return**-Anweisung kann auch ganz fehlen, dann erfolgt der Rücksprung in die aufrufende Funktion beim Erreichen des Funktionsendes (der schließenden geschweiften Klammer um den Funktionkörper)
- der Funktionswert hinter der **return**-Anweisung wird meist in runde Klammern eingeschlossen, dies ist aber nicht notwendig, fehlt der Funktionswert, so wird an die aufrufende Funktion auch kein Wert zurückgegeben (siehe **void**-Datentyp für Funktionen)
- ebenfalls erlaubt und bei älteren C-Programmen (nicht ANSI-C) häufig zu finden ist folgender Funktionsaufbau:

```
name ( Parameternamen, optional )
Parameterdeklarationen, optional
{
    Vereinbarungen
    Anweisungen
    return Funktionswert,optional
}
```

- die Parameternamen müssen alle in zugehörigen Parameterdeklarationen auftauchen

- die einfachste Funktion ist die leere Funktion:

```
dummy () {}
```

Beispielprogramm P6-1.C:

```
# include <stdio.h>

# define MAXLINE 1000

main()     /* alle Zeilen mit Suchmuster finden */
{
char line[MAXLINE];

while (getline(line,MAXLINE) > 0)
    if (index(line,"the") > 0)
        printf("%s",line);
}

getline(char s[], int lim)    /* Zeile in s ablegen, Länge
                                 liefern */
{
    int c,i;

    i=0;
    while (--lim>0 && (c=getchar()) != EOF && c!='\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return(i);
}

index (char s[],char t[])    /* Position von t in s liefern,
                                -1 falls nicht da */
{
    int i, j, k;

    for (i=0; s[i] != '\0'; i++) {
        for (j=i,k=0; t[k]!='\0' && s[j]==t[k]; j++,k++)
            ;
        if (t[k] == '\0')
            return(i);
    }
    return(-1);
}
```

6.2 DATENTYP UND DEKLARATION EINER FUNKTION

- bisher haben wir nur **int** Funktionen besprochen
- Funktionsaufrufe und Funktionen ohne Typangabe werden wie **int** Funktionen behandelt
- da **char** immer in **int** gewandelt wird, sind damit auch schon die **char** Funktionen abgedeckt
- die meisten Funktionen sind **int** Funktionen

- soll eine Funktion einen anderen Datentyp als **int** liefern, so müssen 2 Dinge getan werden:
  1. die Funktion muß vor ihrem ersten Aufruf als eine solche mit dem gewünschten Typ deklariert werden (die Typen der Parameter müssen hier nicht festgelegt werden, man kann dies aber tun (Prototyping, siehe am Ende dieses Kapitels))
  2. der Funktionsdefinition selbst muß der Typ vorangestellt werden

Beispielprogramm P6-2.C

```
# define PI 3.1415

main()     /* Umfang und Fläche eines Kreises */
{
int i;
float umf(), radius;
double flae();

for (i=1,radius=1.; i <= 10; i++, radius += .5)
    printf("Kreis %2d Radius=%6.2f Umfang=%8.4f Fläche=\
        %8.4f\n",i,radius,umf(radius),flae(radius));
}

float umf(float rad) ;
{
    return(2.*PI*rad);
}

double flae(float r) ;
{
    return(PI*r*r);
}
```

Beispielprogramm P6-3.C

```
double atof(char s[])    /* Zeichenkette s nach double
                            wandeln */
{
    double val, power;
    int i, sign;

    for (i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t`; i++)
        ;     /* Zwischenraum uebergehen */
    sign = 1;
    if (s[i]=='+' || s[i]=='-')   /* Vorzeichen */
        sign = (s[i++]=='+') ? 1 : -1;
    for (val=0; s[i]>='0' && s[i] <='9'; i++)
        val = 10 * val + s[i]-'0';
    if s[i] == '.')
        i++;
    for (power=1, s[i]>='0' && s[i]<='9'; i++) {
        val = 10 * val + s[i]-'0';
        power *= 10;
    }
    return (sign * val / power);
}
```

- Funktionen haben oft Wirkungen, die über den Funktionswert hinausgehen (z.B. **getline**: Füllen eines Zeichenvektors)
- der Funktionswert dient in diesen Fällen dann häufig nur als Statusanzeige

- der in der Funktion ermittelte Funktionswert muß in der aufrufenden Funktion nicht ausgewertet, also auch nicht zugewiesen werden
- soll eine Funktion grundsätzlich keinen Funktionwert liefern, dann sollte man sie mit dem speziellen Datentyp **void** deklarieren; die Funktion entspricht dann einer SUBROUTINE in FORTRAN bzw. einer PROCEDURE in PASCAL

Beispielprogramm P6-4.C

```
void help(int nr)      /* Ausgabe eines Hilfstextes */
{
    switch (nr) {
        case 1:
            printf("Hilfe 1\n");
            break;
        case 2:
            printf("Hilfe 2\n");
            break;
        case 3:
            printf("Hilfe 3\n");
            break;
        default:
            printf("Fehlerhafter Aufruf von HELP\n");
    }
}
```

- will man in einer Funktionsdeklaration auch die Parametertypen deklarieren, dann kann man das so tun, daß man anstatt der Parameter nur deren Typen in der Funktionsdeklaration angibt:

```
float umf (float)
double flae (float)
int funcbei (int,float,double)
```

- nach einer solchen Deklaration der Parametertypen ist z.B. *lint* in der Lage, bei jedem Funktionsaufruf die richtigen Typen der verwendeten aktuellen Parameter sowie deren Anzahl abzuprüfen

6.3 PARAMETERÜBERGABE

- alle Parameter werden "by value" übergeben, d.h. eine Kopie des Parameterwertes wird übergeben
- das bedeutet, daß Funktionen ihre Parameter nur innerhalb der Funktion, aber nicht im aufrufenden Programm ändern können

Beispielprogramm P6-5.C

```
main()     /* Beispiel fuer Parameteruebergabe und die
              Aenderbarkeit von Parametern in der Funktion */
{
float radius, neurad(), r;
int i;

i = 10;
radius = 5.23;

printf("Vor Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);
r = neurad(i,radius);
printf("Nach dem Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);
printf("Aber Funktionswert: R=%6.2f\n",r);
}

float neurad(int i, float rad)
{
    i = 0;
    rad = rad * 2;
    printf("In der Funktion: I=%2d RADIUS=%6.2f\n",i,radius);
    return (rad);
}
```

- sollen Parameter in der Funktion dauerhaft geändert werden können, so müssen **Adressen** der zu ändernden Objekte übergeben werden
- wird eine Vektor-(Feld-)name als Parameter übergeben, so wird eigentlich die (Anfangs-)Adresse des Feldes übergeben (deshalb funktioniert z.B. **strcat** (siehe P4-2.C))

Beispielprogramm P6-6.C

```
main()     /* Beispiel fuer Parameteruebergabe und die
              Aenderbarkeit von Parametern in der Funktion
                   Anwendung von Pointern */
{
float radius, neurad(), r;
int i;

i = 10;
radius = 5.23;

printf("Vor Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);
r = neurad(&i,&radius);
printf("Nach dem Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);
printf("Aber Funktionswert: R=%6.2f\n",r);
}

float neurad(int *i,float *rad)
{
    *i = 0;
    *rad = *rad * 2;
    printf("In der Funktion: I=%2d RADIUS=%6.2f\n",i,radius);
    return (*rad);
}
```

6.4 SPEICHERKLASSEN UND GELTUNGSBEREICHE VON NAMEN

- wir haben bisher an verschiedenen Stellen schon einmal die 4 Speicherklassen **extern**, **auto**, **static** und **register** flüchtig kennengelernt
- im Zusammenhang mit Funktionen ist jetzt ein guter Zeitpunkt, diese Kenntnisse zu vertiefen

1. Speicherklasse **auto**

- alle Variablen, denen nicht explizit eine Speicherklasse zugewiesen wird und die nicht außerhalb von Funktionen vereinbart werden, fallen in die Speicherklasse **auto**
- man kann einer Variablen explizit die Speicherklasse **auto** zuordnen, indem man vor die Typangabe bei der Variablenvereinbarung das Schlüsselwort **auto** setzt
- automatische Variable werden bei jedem Funktionsaufruf neu erzeugt und beim Verlassen der Funktion wieder zerstört
- daher ist ihr Geltungsbereich auf die lokale Funktion, in der sie vereinbart wurden, beschränkt
- dies gilt auch für Blöcke (Variablen, die innerhalb von Blöcken vereinbart werden und die in die Speicherklasse **auto** fallen, sind nur lokal in diesem Block bekannt)
- **auto** Variablen können beliebig (nicht nur mit Konstanten) initialisiert werden
- nicht initialisierte **auto** Variablen haben einen undefinierten Wert (es existiert kein Weglasswert!)
- **auto Vektoren** können **nicht** initialisiert werden

2. Speicherklasse **extern**

- alle Objekte, die außerhalb von Funktionen vereinbart werden, sind in der Speicherklasse extern
- die Funktionsnamen selbst sind ebenfalls in der Speicherklasse extern
- externe Objekte sind ab ihrer Vereinbarung bis zum Ende des Quellencodes und sogar in anderen Quellencodedateien bekannt
- bei der Vereinbarung eines Objektes als extern kann es sich um eine Definition oder um eine Deklaration handeln
- die folgenden Vereinbarungen (immer außerhalb von Funktionen) unterscheiden sich wesentlich:

```
int sp = 0;                 /* Definition */
double vector[N];

extern int sp;              /* Deklaration */
extern double vector[];
```

- bei einer **Definition** wird eine Variable erzeugt
- nur hier ist eine Initialisierung möglich
- fehlt eine Initialisierung, so wird der Weglasswert 0 eingesetzt

- bei einer **Deklaration** werden nur die Variableneigenschaften festgelegt, die Variable selbst muß an anderer Stelle im Quellencode definiert sein
- Deklarationen dürfen für eine bestimmte Variable mehrmals im (gesamten) Quellencode vorkommen, eine Definition aber nur einmal

Beispielprogramm P6-7.C

```
# define BUFSIZE 100

char buff[BUFSIZE];     /* Puffer fuer ungetch() */
int bufp = 0;           /* naechste freie Position */

getch()     /* (evtl. zurueckgest.) Zeichen holen */
{
    return ((bufp > 0) ? buff[--bufp] : getchar());
}

ungetch(int c)     /* Zeichen zurueckstellen */
{
    if (bufp > BUFSIZE)
        printf("ungetch: Zuviele Zeichen\n");
    else
        buff[bufp++] = c;
}
```

3. Speicherklasse **static**

- **static** Objekte können sowohl intern als auch extern sein
- **static** Objekte innerhalb von Funktionen sind nur lokal bekannt, behalten im Gegensatz zu **auto** Objekten aber ihre Werte zwischen den Funktionsaufrufen bei
- bzgl. der Initialisierung gilt dasselbe wie für **externe** Objekte, static Vektoren sind daher initialisierbar
- Zeichenketten innerhalb von Funktionen sind immer in der Speicherklasse **static** (z.B. printf() Parameterstring)
- **static** Objekte außerhalb von Funktionen sind **externe** Objekte, deren Namen aber nur in dieser Quellencodedatei bekannt ist
- Beispiele zur Initialisierung eines Vektors:

```
static int ndigit[10] = { 0,1,2,3,4,5,6,7,8,9 };
char string[] = "Dies ist ein String";
```

4. Speicherklasse **register**

- in dieser Speicherklasse können sich einfache Variable befinden
- sie entspricht in ihren sonstigen Eigenschaften der Speicherklasse **auto**
- der Compiler versucht **register** Variablen so zu verwenden, daß sie in einem wirklichen Hardwareregister der CPU gehalten werden
- ist dies nicht möglich, so wird **register** ignoriert und die Variable wie **auto** behandelt

6.5 REKURSIVE FUNKTIONEN

- Funktionen dürfen zwar nicht geschachtelt sein, aber sie können sich selbst direkt oder indirekt aufrufen
- bei jedem Funktionsaufruf werden neue eigene lokale Variablen erzeugt (außer bei der Speicherklasse **static**)
- viele Probleme lassen sich kompakt und elegant mittels Rekursion lösen

Beispielprogramm P6-8.C

```
printd(int n)     /* n dezimal ausgeben */
                  /* 1. Version ohne Rekursion*/
{
    char s[10];
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    i = 0;
    do {
        s[i++] = n%10 + '0';   /* naechstes Zeichen holen */
    } while ((n /= 10) > 0);   /* und durch Division
                                        entfernen */
    while (--i >= 0)           /* in umgekehrter Reihen- */
        putchar(s[i]);         /* folge ausgeben */
}
```

Beispielprogramm P6-9.C

```
printd(int n)     /* n dezimal ausgeben */
                  /* 2. Version  Rekursiv */
{
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if ((i = n/10) != 0)
        printd(i);     /* printd rekursiv aufrufen */
    putchar(n%10 + '0');
}
```

Beispielprogramm P6-10.C

```
long fak(int n)     /* Fakultaet berechnen */
{
    if (n <= 1)
        return(1);
    else
        return(n*fak(n-1));
}
```

7. ZEIGER UND VEKTOREN
7.1 VEKTOREN
7.1.1 EINDIMENSIONALE VEKTOREN

Zur Zusammenfassung mehrerer Objekte gleichen Typs zu einer Einheit verwendet man Vektoren, oft auch als Felder (arrays) bezeichnet. Sie bilden eine lineare Anordnung ihrer Grundobjekte im Speicher. Sie werden durch die Verwendung von eckigen Klammern (**[]**) nach dem Variablennamen erzeugt. Innerhalb der Klammern wird die Größe des Vektors (Feldes) in Anzahl Elementen des definierten Typs angegeben:

```
char s;          /* 1  Zeichen */
char sf[20];     /* 20 Zeichen */
int i;           /* 1  ganze Zahl */
int iff[100];    /* 100 ganze Zahlen (= 200 Bytes) */
```

Der Zugriff auf die einzelnen Vektorelemente erfolgt durch Angabe des Vektornamens gefolgt vom Index in eckigen Klammern:

```
int j=5, k;
sf[8] = 'a';
sf[j] = sf[19];
k     = iff[j];
```

Das **1.Element** hat immer den festgelegten Index **0**, das **N-te** den Index **N-1**. Vektorelemente können L-Werte sein, Vektornamen alleine nicht, da sie feste Adressen (die Adresse des 1. Vektorelementes) darstellen. Als dafür zu große Einheit können Vektoren **nicht** in der Speicherklasse **register** sein. Vektoren in der Speicherklassse **auto** können **nicht initialisiert** werden, dies ist nur bei Vektoren in den Speicherklassen **static** und **extern** möglich. Werden diese nicht initialisiert, so haben alle Elemente den Wert 0. Die Initialisierungswerte werden nach einem Gleichheitszeichen in geschweiften Klammern angegeben. Nicht initialisierte Elemente erhalten ebenfalls den Wert 0. Beispiele für Vektorinitialisierungen:

```
static int iff[10] = {
    1,2,3,4,5,6,7,8,9,10 };

static int jf[5] = {0,1,3};   /* restl. Elemente = 0 */

static int kf[] = {1,3,5,7,11,13};  /* Laenge aus Anzahl */
                                    /* Init-Elemente */
```

Nachfolgend einige Beispiel für Vektorverarbeitung:

```
/* Feld mit der Summe der natuerlichen Zahlen bis zu
   diesem Feldelement belegen */

# define MAX 100

int sum[MAX], i, j;

for (i=0,j=0; i<MAX; i++) {
    j = j + i + 1;
    sum[i] = j;
}
```

```
/* nochmals unter Auslassung des 1.Elementes */

# define MAX 101

int sum[MAX], i, j;

for (i=1,j=0; i<MAX; i++) {
    j = j + i;
    sum[i] = j;
}


/* Feld mit den Quadratzahlen belegen  1.Version */
#define MAX 100

int mul[MAX], i;
for (i=0; i<MAX; i++)
    mul[i] = (i+1) * (i+1);


/* Feld mit den Quadratzahlen belegen  2.Version */
#define MAX 100

int mul[MAX], i, j;
for (i=0,j=1; i<MAX; i++,j++) {
    mul[i] = j*j;
```

7.1.2 ZEICHENKETTEN

Zeichenketten sind eindimenionale Vektoren mit mehreren Besonderheiten. Sie können auch in der Speicherklasse **auto** initialisiert werden. Als Zeichenkette müssen sie mit '**\0**' abgeschlossen sein. Dies läßt sich aber auch automatisch mit Hilfe der vereinfachten Initialisierung erreichen:

```
char s[] = {'s','t','r','i','n','g','\0'};
char s[] = "string";       /* \0 intern angehaengt */
char s[10] = "string";     /* restl. Elemente mit 0 init. */
```

Ansonsten werden Zeichenketten wie normale Vektoren behandelt. Insbesondere ist der Zugriff auf Vektorelemente gleich:

```
s[0] hat den Wert 's'
s[3] hat den Wert 'i'
```

7.1.3 MEHRDIMENSIONALE VEKTOREN

Mehrdimenisonale Vektoren werden durch 2 oder mehr Paare eckiger Klammern nach dem Objektnamen definiert (Dabei stehen zwischen den Klammern keine Kommas.). Ein 2-dimensionaler Vektor ist ein Vektor, dessen Komponenten Vektoren sind, für höhere Dimensionen gilt Entsprechendes. Die Angabe der Elementanzahl der ersten Dimension kann (in Funktionen) fehlen, die der anderen Dimensionen nicht, da sonst die Indexberechnung nicht richtig arbeiten kann.

```
/* Deklaration extern oder in der aufrufenden Funktion */
int md[5][10];
funct(md,5);

/* in der Funktion */
void funct(mdf,anzahl)   /* in anzahl wird die aktuelle */
int mdf[][10];           /* Groesse der 1. Dimension */
int anzahl;              /* übergeben */
{
    int i, j;
    for (i=0; i<anzahl; i++)
        for (j=0; j<10; j++)
            mdf[i][j] = i + j;
}
```

Die gleiche Vektorbelegung wie mit diesem kurzen Programm kann man auch durch Initialisierung erreichen. Diese müßte dann wie folgt aussehen:

```
static int md[5][10] = {
    {0,1,2,3,4,5,6,7,8,9},
    {1,2,3,4,5,6,7,8,9,10},
    {2,3,4,5,6,7,8,9,10,11},
    {3,4,5,6,7,8,9,10,11,12},
    {4,5,6,7,8,9,10,11,12,13}
};
```

Die Angaben für die 2. (und weitere) Dimension werden also jeweils in eigene geschweifte Klammern geschrieben. Innerhalb der geschweiften Klammern gilt wiederum, daß evtl. nicht initialisierte Elemente auf 0 gesetzt werden. Man kann die inneren geschweiften Klammern auch weglassen, dann wird der ganze Vektor Element für Element initialisiert.

Beispielprogramm P7-1.C

```
static int day_tab[2][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};

/* Tag im Jahr aus Monat und Tag bestimmen */
day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i=1; i < month; i++)
        day += day_tab[leap][i];
    return (day);
}
```

```
/* Monat und Tag aus Tag im Jahr */
month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i=1; yearday > day_tab[leap][i]; i++)
        yearday -= day_tab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

7.2 ZEIGER

7.2.1 ZEIGER UND ADRESSEN

Vektoren werden über ihre Anfangsadresse angesprochen und auch mittels dieser Anfangsadresse an Funktionen übergeben. Die **Anfangsadresse eines Vektors** steht fest, ist also eine **Konstante**. Im Gegensatz dazu ist ein **Zeiger** eine **Variable**, die die Adresse eines (beliebigen) anderen Objektes enthält. Man kann auf dieses Objekt indirekt über einen Zeiger zugreifen.

Die Adresse eines Objektes erhält man durch Anwendung des Adreßoperators **&**. **&** kann auf Variablen und Vektorelemente angewendet werden, nicht aber auf Vektornamen selbst (Warum? Ein Vektorname **hat** keine Adresse, er **ist** eine Adresse!). Ebenso haben natürlich Variablen in der Speicherklasse **register** keine Adressen.

Beispiele für die Anwendung des Adreßoperators und das Speichern der Adresse in einem Zeiger:

```
px = &x;     /* px erhält als Wert die Adresse von x */
pf = &f[5];  /* pf erhält als Wert die Adresse des
                6. Elementes von f */
```

Der Zugriff auf ein Objekt, dessen Adresse in einer Variablen (einem Zeiger) steht, geschieht mittels des Operators ***** (Inhalt von):

```
y = *px     /* y erhält den Wert des Objektes, dessen
               Adresse in px steht */
px = &x;
y = *px;    /* y = x; */
```

Zeiger zeigen immer auf Objekte eines bestimmten Typs. Sie müssen daher deklariert werden. Auch in der Zeigerdefinition wird der Operator ***** verwendet. Zeigerdefinitionen kann man als Muster verstehen:

```
int *pk;    /* pk ist Zeiger auf int */
char *zs;   /* zs ist Zeiger auf char */

int x, y, *px;
px = &x;    /* Adressen */
y = *px;    /* Werte */
```

Die Kombination ***Zeiger** kann in Ausdrücken überall dort auftreten, wo auch das Objekt, auf das der Zeiger zeigt, selbst stehen könnte:

```
y = *px + 10;
y = *px + *px;
printf("%d\n", *px);
*px = 0;
py = px;    /* falls py auch Zeiger auf int */
```

Bei der Verwendung des Operators ***** muß man die Operatorrangfolge und -assoziativität genau beachten. Dies erscheint zunächst etwas schwierig, da dieser Operator ungewohnt ist. Hier einige Beispiele mit dem ***** Operator und anderen Operatoren:

```
y = *px + 1;    /* Inhalt von px plus 1 */
y = *(px+1);    /* Inhalt der Adresse px+1 */
*px += 1;       /* Inhalt von px = Inhalt von px plus 1 */
(*px)++;        /* Inhalt von px inkrementieren */
*px++;          /* wie *(px++); (Assoziativität)
                   Inhalt der Adresse px; px = px plus 1*/
*++px;          /* Inhalt der Adresse px+1; px = px plus 1 */
```

Zeiger haben nur dann sinnvolle Werte, wenn sie die Adresse eines Objektes oder 0 enthalten. Für den Zeigerwert 0 ist garantiert, daß er nirgends hinzeigt (**NULL** bzw. NIL(in PASCAL)).

Beispielprogramm P7-2.C

```
/* Funktion zum Vertauschen ihrer Argumente */

/* 1. Versuch, FALSCH, da Werte nur innerhalb */
/* von swap getauscht */
/* in der aufrufenden Funktion */
swap(a,b);          /* Werte übergeben */


/* die Tauschfunktion */
swap(int x,int y);     /* FALSCH */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}


/* 2. Versuch, RICHTIG, da Objekte, auf die die */
/* Adressen zeigen, ausgetauscht werden */
/* in der aufrufenden Funktion */
swap(&a,&b);        /* Adressen übergeben */

/* die Tauschfunktion*/
swap(int *px,int *py);     /* *px und *py austauschen */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

7.2.2 ZEIGER UND VEKTOREN

Zeiger und Vektoren hängen sehr eng zusammen. Das sieht man schon daran, daß ein Vektorname genau wie ein Zeiger auch eine Adresse ist (allerdings eine Adreßkonstante). Als Konsequenz aus dem engen Zusammenhang zwischen Vektoren und Zeigern ergibt sich die für den Anfänger

zunächst etwas verwirrende Tatsache, daß alle Operationen mit Vektorindizes auch mit Zeigern formuliert werden können. Dies ist meist effizienter, aber für Anfänger schwerer zu verstehen. Ausgehend von den Definitionen und Anweisungen:

```
int a[10];        /* int Vektor mit 10 Elementen */
int *pa, y;       /* Zeiger auf int Objekte */
pa = &a[0];       /* pa hat Adresse 1. Vektorelement */
pa = a;           /* pa hat ebenfalls Adresse 1. Element */
y = *pa;          /* y = 1. Element von Vektor a */
```

sind die folgenden Angaben äquivalent:

```
a[i]        *(a+i)     /* i + 1tes Element des Vektors a */
&a[i]        (a+i)     /* Adresse des i + 1ten Elementes */
pa[i]       *(pa+i)
```

Statt Vektorname und Indexausdruck kann immer auch ein Zeiger mit Abstand stehen. Als Konsequenz aus Vektorname (=Adresskonstante) und Zeiger (=Adressvariable) ergeben sich folgende erlaubten und nicht erlaubten Zusweisungen:

```
pa = a;   /* erlaubt */          a = pa;   /* falsch */
pa++;     /* erlaubt */          a++;      /* falsch */
                                 pa = &a;  /* falsch */
```

Beispielprogramm P7-3.C

```
strlen(char *s)   /* Laenge der Zeichenkette s */
{
    int n;

    for (n=0; *s != '\0'; s++)
        n++;
    return (n);
}
```

7.2.3 ZEIGERARITHMETIK

Mit Zeigern können bestimmte arithmetische Operationen und Vergleiche durchgeführt werden. Es sind natürlich nur die Operationen erlaubt, die zu sinnvollen Ergebnissen führen. Zu Zeigern dürfen ganzzahlige Werte addiert und es dürfen ganzzahlige Werte subtrahiert werden. Zeiger dürfen in- und dekrementiert werden und sie dürfen voneinander subtrahiert werden (Dies ist i.a. nur sinnvoll, wenn beide Zeiger auf Elemente des gleichen Objektes (z.B. Vektor) zeigen.).

Man kann Zeiger mittels der Operatoren **>**, **>=**, **<**, **<=**, **!=** und **==** miteinander vergleichen. Wie bei der Zeigersubtraktion ist das aber i.a. nur dann sinnvoll, wenn beide Zeiger auf Elemente des gleichen Vektors zeigen. Eine Ausnahme bildet hier der Zeigerwert NULL.

Alle anderen denkbaren arithmetischen und logischen Operationen (Addition von 2 Zeigern, Multiplikation, Division, Shifts oder Verwendung von logischen Operatoren, sowie Addition und Subtraktion von **float** oder **double** Werten) sind mit Zeigern **nicht** erlaubt.

Wie funktioniert nun aber die Zeigerarithmetik? Sei p ein Zeiger und n eine ganze Zahl, dann bezeichnet **p+n** das n-te Objekt im Anschluß an das Objekt, auf das p gerade zeigt. Es wird also nicht der Wert n zu p direkt addiert, sondern n wird vorher mit der Typlänge des Typs, auf den p zeigt, multipliziert. Dieser Typ wird aus der Deklaration von p bestimmt.

Beispielprogramm P7-4.C

```
strlen(char *s)    /* Laenge der Zeichenkette s  2.Version */
{
    char *p = s;

    while (*p != '\0')
        p++;
    return (p-s);
}
```

Beispielprogramm P7-5.C

```
/* einfache Speicherverwaltung */

#define NULL 0   /* Zeigerwert fuer Fehleranzeige */
#define ALLOCSIZE 1000     /* verfuegbarer Platz */

static char allocbuf[ALLOSIZE];
static char *allocp = allocbuf;   /* naechste freie
                                     Position */

char *alloc(int n)   /* liefert Zeiger auf Platz fuer
                        n Zeichen */
{
    if (allocp+n <= allocbuf+ALLOCSIZE) {   /* reicht */
        allocp += n;
        return (allocp - n);      /* alter Zeiger */
    } else                        /* nicht genug Platz */
        return (NULL);
}


free(char *p)     /* Speicher ab p freigeben */
{
    if (p >= allocbuf && p < allocbuf+ALLOCSIZE)
```

```
        allocp = p;
}
```

Beispielprogramm P7-6.C

```
/* 4 Versionen von strcpy: Kopieren eines Strings */
/* 1. t nach s kopieren Version mit Vektoren */
strcpy(char s[],char t[])
{
    int i;

    i = 0;
    while ((s[i]=t[i]) != '\0')
        i++;
}



/* 2. t nach s  kopieren  1.Version mit Zeigern */
strcpy(char *s,char *t)
{
    while ((*s=*t) != '\0') {
        s++;
        t++;
    }
}



/* 3. t nach s kopieren  2.Version mit Zeigern */
strcpy(char *s,char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}



/* 4. t nach s kopieren  3.Version mit Zeigern */
strcpy(char *s,char *t)
{
    while (*s++ = *t++)
        ;
}
```

Beispielprogramm P7-7.C

```
/* 2 Versionen von strcmp: Stringvergleich */
strcmp(char s[],char t[])   /* liefert <0 wenn s<t, 0 wenn */
                            /* s==t und >0 wenn s>t */
{
    int i;

    i = 0;
    while (s[i] == t[i])
        if (s[i++] == '\0')
            return (0);
    return (s[i] - t[i]);
}



strcmp(char *s,char *t)   /* liefert <0 wenn s<t, 0 wenn */
                          /* s==t und >0 wenn s>t */
{
    for ( ; *s == *t; s++,t++)
        if (*s == '\0')
            return (0);
    return (*s - *t);
}
```

7.2.4 Zeiger auf Funktionen

In C ist eine Funktion keine Variable, sondern der Funktionsname ist (wie ein Vektorname) eine Adreßkonstante. Es können aber Zeiger auf Funktionen definiert werden, die man dann wie Variablen verwenden kann. Damit wird es möglich, Funktionen an Funktionen zu übergeben.

Ein Zeiger auf eine Funktion wird so vereinbart:

```
(* Funktionsname) ()     /* Zeiger auf eine Funktion */
```

Im Gegensatz zu:

```
* Funktionsname ()       /* Funktion, die einen
                             Zeigerwert liefert */
```

Die Möglichkeit und der Gebrauch von Zeigern auf Funktionen wird am besten an einem Beispiel klar:

```
/* in der aufrufenden Funktion */
int strcmp(), numcmp(), swap(), tausche();

sort (numcmp,swap);     /* sortiere mittels der Funktionen */
                        /* numcmp und swap */
sort (strcmp,swap);     /* sortiere mittels der Funktionen */
                        /* strcmp und swap */
sort (numcmp,tausche);  /* sortiere mittels der Funktionen */
                        /* numcmp und tausche */
```

```
/* in der gerufenen Funktion */
sort(int (*comp)(),int (*exch)())
/* comp und exch sind Zeiger auf Funktionen */
{
    if ((*comp)(v[j],v[j+gap]) <= 0)  /* die Funktion, auf */
                                /*  die comp zeigt, wird mit */
                                /* den Parametern v[j] und */
                                /* v[j+gap] aufgerufen */
        break;
    (*exch)(&v[j],&[j+gap]);
}
```

7.3 ANWENDUNGEN

7.3.1 VEKTOREN VON ZEIGERN UND ZEIGER AUF ZEIGER

Genauso wie man Vektoren aus den Grunddatentypen (char, int, float und double) bilden kann, kann man dies auch mit Zeigern tun. Ein Vektor von Zeigern wird so definiert:

```
* Vektorname []
```

gelesen von rechts nach links (wegen des Vorranges von []): Vektor von Zeigern. Dagegen ist

```
(* Vektorname) []
```

ein Zeiger auf einen Vektor.

Zeiger auf Zeiger sind eine äquivalente Formulierung für Vektoren von Zeigern

```
int *iff[];       /* sind äquivalent, da iff die
int **iff;           Adresse des Vektors enthält */
```

Das folgende Beispiel zeigt, wie man einen Vektor von Zeigern initialisieren kann:

Beispielprogramm P7-8.C

```
char *month_name(int n)   /* liefert Name des n. Monats */
{
    static char *name[] = {   /* Vektor von Zeigern */
        "falscher Monat",     /* String ist ein char- */
        "Januar",             /* Vektor und daher durch */
        "Februar",            /* seine Anfangsadresse */
        "Maerz",              /* charakterisiert */
        "April",
        "Mai",
        "Juni",
        "Juli",
        "August",
        "September",
        "Oktober",
        "November",
        "Dezember"
    };
    return ((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Beispielprogramm P7-9.C

```
/* Programm zum Einlesen, alphabetischen Sortieren
   und Ausgeben von Eingabezeilen */
#define NULL 0
#define LINES 100      /* maximale Anzahl Zeilen */

main()     /* Eingabe Zeilen sortieren */
{
    char *lineptr[LINES];   /* Zeiger auf Textzeilen */
    int nlines;             /* Anzahl gelesener Zeilen */

    if ((nlines = readlines(lineptr,LINES)) >= 0) {
        sort(lineptr,nlines);
        writelines(lineptr,nlines);
    } else
        printf ("input too big to sort\n");
}


#define MAXLEN 1000     /* Maximallaenge einer Zeile */

/* Eingabezeilen einlesen um sie zu sortieren */
readlines(char *lineptr[],int maxlines)
{
    int len,nlines;
    char *p, *alloc(), line[MAXLEN];

    nlines = 0;
    while ((len = getline(line,MAXLEN)) > 0)
        if (nlines >= maxlines)
            return(-1);
        else if ((p=alloc(len)) == NULL)
            return(-1);
        else {
            line[len-1] = '\0'; /* Zeilentrenner entfernen */
            strcpy(p,line);
            lineptr[nlines++] = p;
        }
    return(nlines);
}


/* Zeilen ausgeben */
writelines(char *lineptr[],int nlines)
{
    int i;

    for (i=0; i < nlines; i++)
        printf("%s\n",lineptr[i]);
}
```

```
/* Zeichenketten v[0],...,[n-1] in aufsteigender Reihen- */
/* folge sortieren, es werden die Zeichenketten */
/* verglichen und ihre Zeiger entsprechend sortiert */
sort(char *v[], int n)
{
    int gap, i, j;
    char *temp;

    for (gap=n/2; gap > 0; gap /= 2)
        for (i=gap; i < n; i++)
            for (j = i-gap; j >= 0; j -= gap) {
                if (strcmp(v[j],v[j+gap]) <= 0)
                    break;
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

7.3.2 ZEIGER UND MEHRDIMENSIONALE VEKTOREN

Ein mehrdimensionaler Vektor ist ja ein Vektor, dessen Elemente selbst wieder Vektoren sind (siehe Abschnitt 7.1.3). Man kann solche Vektoren auch mit Zeigern realisieren. Betrachten wir dazu folgendes Beispiel:

```
int md[10][10];   /* 2 dimensionaler Vektor mit insgesamt
                     100 int-Elementen */
int *mp[10];      /* Vektor mit 10 Zeigern auf int-Objekte */
```

Die Ausdrücke

```
md[5][5]     und     mp[5][5]
```

sind beide möglich und bezeichnen das gleiche Element.

md ist ein Vektor mit 10 Elementen, die selbst wiederum Vektoren mit je 10 **int** Elementen sind. Insgesamt enthält **md** also 100 Elemente, für die auch der Speicherplatz bereitgestellt wird.

mp ist ein Vektor mit 10 Elementen, die Zeiger auf **int** Objekte sind (Speziell könnten diese natürlich auch auf je einen **int** Vektor mit je 10 Elementen zeigen.). Im Falle von **mp** wurden aber nur 10 Speicherplätze für die Zeiger angelegt, die noch nicht initialisiert sind. Im Vergleich zu **md** wird hier für das gleiche Feld insgesamt mehr Speicherplatz und evtl. eine Initialisierung benötigt. Dafür hat man aber 2 Vorteile:

a) wegen einfacherer Adressierung (meist) schnellere Programmausführung
b) Zeiger können auf verschieden lange Vektoren zeigen (höhere Dimensionen der Vektoren haben immer eine feste Länge)

7.3.3 ARGUMENTE DER KOMMANDOZEILE

Bisher haben wir die Funktion **main** als parameterlos behandelt. Im allgemeinen hat diese Funktion aber 2 Parameter **argc** und **argv**, die Angaben über die Kommandoparameter beim Programmaufruf enthalten.

argc ist ein **int** Parameter und enthält die Anzahl der Paramter der Kommandozeile einschließlich des Programmaufrufes selbst (hat also immer mindestens den Wert 1).

argv ist ein **Vektor** mit Zeigern auf **Zeichenketten**. Diese Zeichenketten enthalten die Aufrufparameter der Kommandozeile, wobei

der **1. Parameter** der **Programmaufruf** selbst ist. Letztes Argument ist die Vektorkomponente argv[argc - 1].

Beispiel:

```
/* Programmaufruf auf Betriebssystemebene */
echo Hier ist Echo!

/* Werte von argc und argv im aufgerufenen Programm */
argc = 4
argv[0] = "echo"
argv[1] = "Hier"
argv[2] = "ist"
argv[3] = "Echo!"
```

Beispielprogramm P7-10.C

```
/* 3 Versionen von echo */
/* Echo der Aufrufargumente 1.Version */
main(int argc,char *argv[])
{
    int i;

    for (i=1; i < argc; i++)
        printf("%s%c",argv[i],(i<argc-1) ? ' ' : '\n');
}



/* Echo der Aufrufargumente 2.Version */
main(int argc,char *argv[])
{
    while (--argc > 0)
        printf("%s%c",*++argv,(argc > 1) ? ' ' : '\n');
}



/* Echo der Aufrufargumente 3.Version */
main(int argc,char *argv[])
{
    while (--argc > 0)
        printf((argc > 1) ? "%s " : "%s\n",*++argv);
}
```

Beispielprogramm P7-11.C

```
/* Programm zum Finden von Zeilen, die ein anzugebendes
   Suchmuster enthalten bzw. nicht enthalten
   Aufruf: find [-n] [-x] suchmuster */

#define MAXLINE 1000

/* Zeilen mit Suchmuster finden */
main(int argc,char *argv[])
{
    char line[MAXLINE], *s;
    long lineno = 0;
    int except = 0, number = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        for (s=argv[0]+1; *s != '\0'; s++)
            switch (*s) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n",*s);
                argc = 0;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line,MAXLINE) > 0) {
            lineno++;
            if ((index(line, *argv) >= 0) != except) {
                if (number)
                    printf("%ld: ",lineno);
                printf("%s",line);
            }
        }
}
```

8. ZUSAMMENGESETZTE DATENSTRUKTUREN

8.1 STRUKTUREN

Während Vektoren eine Zusammenfassung von Objekten gleichen Typs sind, handelt es sich bei einer Struktur um eine Zusammenfassung von Objekten möglicherweise verschiedenen Typs zu einer Einheit (vergl. **record** in PASCAL). Die Verwendung von Strukturen bietet Vorteile bei der Organisation komplizierter Daten. Beispiele sind Personaldaten (Name, Adresse, Gehalt, Steuerklasse, usw.) oder Studentendaten (Name, Adresse, Studienfach, Note).

8.1.1 VEREINBARUNG VON STRUKTUREN

Strukturen werden mit Hilfe des Schlüsselwortes **struct** vereinbart

struct *Strukturname* { *Komponente(n)* } *Strukturvariable(n) Init.* **;**

Strukturname ist optional und kann nach seiner Definition für die Form (den Datentyp) dieser speziellen Struktur verwendet werden, d.h. als Abkürzung für die Angaben in den geschweiften Klammern. **Strukturkomponenten** werden wie normale Variable vereinbart. **Struktur-** und **Komponentennamen** können mit anderen Variablennamen identisch sein ohne daß Konflikte auftreten, da sie vom Compiler in einer separaten Tabelle geführt werden.

Durch die Angabe einer (oder mehrerer) **Strukturvariablen** wird diese Struktur erzeugt (d.h. Speicherplatz dafür bereitgestellt). Strukturvereinbarungen ohne Angabe einer Strukturvariablen legen nur die Form (den Prototyp) der Struktur fest.

Strukturen können wie Vektoren nur initialisiert werden, wenn sie **global** oder **static** sind.

Beispiele für Strukturvereinbarungen:

```
struct datum {    /* legt die Form der Struktur datum fest */
    int tag;
    int monat;
    int jahr;
    int jahrestag;
    char mon_name[4];
};

struct datum {     /* erzeugt die Strukturen geb_dat */
    int tag;       /* und heute mit der Form datum */
    int monat;
    int jahr;
    int jahrestag;
    char mon_name[4];
} geb_dat, heute;

struct datum geb_dat,heute;   /* erzeugt ebenfalls die */
                              /* Strukturen geb_dat und */
                              /* heute mit der Form datum */

struct datum heute = {26,9,1987,0,"jun"};   /* erzeugt */
                              /* die Struktur heute der */
                              /* Form datum mit den ange- */
                              /* gebenen Initialisierungen */
```

Strukturen können als Elemente ebenfalls wieder Strukturen enthalten (allerdings nicht sich selbst) und Strukturen können zu Vektoren zusammengefaßt werden:

```
struct kunde {
    char name[NAMLAE];
    char adresse[ADRLAE];
    int kund_nr;
    struct datum liefer_dat;
    struct datum rech_dat;
    struct datum bez_dat;
};

struct kunde kunde1, kunde2, ... ;

struct kunde kunden[KUNANZ];
```

8.1.2 ZUGRIFF AUF STRUKTURELEMENTE

Mit Strukturen sind nur 2 Operationen möglich:

a) Zugriff auf Strukturelemente (direkt und indirekt)
b) (Anfangs-)Adresse bestimmen

Die Adresse wird wie gewohnt mit dem **&** Operator bestimmt. Für den Elementzugriff gibt es zwei eigene Operatoren. Der direkte Zugriff wird dabei mit dem Punktoperator **.** nach folgendem Schema durchgeführt:

Strukturvariable **.** *Komponente*

Beispiele für direkten Zugriff:

```
geb_dat.jahr
heute.tag
heute.mon_name[0]
kunde1.kund_nr
kunden[3].name
kunde2.liefer_dat.monat
```

Für den **indirekten** Zugriff muß die Adresse der Struktur in einem dafür geeigneten Zeiger stehen. Dann kann mit Hilfe des Zeigeroperators -> (Minuszeichen Größerzeichen) auf die Strukturelemente zugegriffen werden:

Strukturzeiger -> *Komponente*

Beispiele für indirekten Zugriff:

```
struct datum *pdat;
struct kunde *pkun;
pdat = &heute;
pkun = &kunden[4];

pdat->tag
pdat->mon_name[1]
pkun->adresse
pkun->rech_dat.monat
```

Bei diesen Zugriffen muß man immer den Vorrang und die Assoziativität der Operatoren **.** und **->** beachten (höchster Vorrang).

Beispiele für Operatorvorrang:

```
struct {
```

```
        int x;
        int *y;
    } *p;

    ++p->x     /* x = x + 1, da implizit ++(p->x) */
    (++p)->x   /* p zeigt auf nächste Struktur, */
               dann Zugriff auf x */
    (p++)->x   /* Zugriff auf x, dann p auf nächste Struktur */
    p++->x     /* wie eben, Warum? */
```

Die gerade gezeigten Beispiele für Ausdrücke würden in einem "echten" Programm allerdings zu schwerwiegenden Laufzeitfehlern führen. Warum?
Es wurde zwar ein Zeiger auf eine Struktur vereinbart, nicht aber die Struktur selbst, für die folglich auch kein Speicherplatz bereitgestellt wird. Daher ist sie natürlich auch nicht mit Werten vordefiniert.
Merke: Mit Zeigern kann man nur arbeiten, wenn sie auch auf eine "vernünftige" Stelle zeigen.

Beispielprogramm P8-1.C

```
/* Realisierung von Tag im Jahr bestimmen mit
   einer Struktur */

/* 1. Möglichkeit mit alter Funktion  day_of_year */
struct datum d;

d.jahrestag = day_of_year(d.jahr,d.monat,d.tag);


/* 2. Möglichkeit mit modifizierter Funktion */
struct datum rech_dat;

rech_dat.jahrestag = day_of_year(&rech_dat);


day_of_year(struct datum *pd)
/* Tag im Jahr aus Monat und Tag bestimmen */
{
    int i, leap, day;

    day = pd->tag;
    leap = pd->jahr%4 == 0 && pd->jahr%100 != 0
           || pd->jahr%400 == 0;
    for (i=1; i < pd->monat; i++)
        day += day_tab[leap][i];
    return (day);
}
```

```
/* Dasselbe für month_day */

month_day(struct datum *pd)
/* Monat und Tag aus Tag im Jahr */
{
    int i, leap;

    leap = pd->jahr%4 == 0 && pd->jahr%100 != 0
           || pd->jahr%400 == 0;
    pd->tag = pd->jahrestag;
    for (i=1; pd->tag > day_tab[leap][i]; i++)
        pd->tag -= day_tab[leap][i];
    pd->monat = i;
}
```

Beispielprogramm P8-2.C

```
/* Vorkommen reservierter Woerter zaehlen */
#include <stdio.h>

#define MAXWORD 20
#define NKEYS (sizeof(keytab) / sizeof(struct key))
#define LETTER 'a'
#define DIGIT '0'

struct key {   /* global, daher initialisierbar */
    char *keyword;
    int keycount;
} keytab[] = {
    "break",0,
    "case",0,
    "char",0,
    "continue",0,
    /* ... */
    "unsigned",0,
    "while",0
};


main()     /* reservierte Worte in C zaehlen */
{
    int n,t;
    char word[MAXWORD];

    while ((t=getword(word,MAXWORD)) != EOF)
        if (t == LETTER)
            if ((n=binary(word,keytab,NKEYS)) >= 0)
                keytab[n].keycount++;
    for (n=0; n < NKEYS; n++)
        if (keytab[n].keycount > 0)
            printf("%4d %s\n",
                keytab[n].keycount, keytab[n].keyword;
}
```

```
/* word in tab[0],...,tab[n-1] finden */
binary(char *word,struct key tab[],int n)
{
    int low,high,mid,cond;

    low = 0;
    high = n-1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond=strcmp(word,tab[mid].keyword)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return(mid);
    }
    return(-1);
}


/* naechstes Wort aus der Eingabe holen */
getword(char *w,int lim)
{
    int c,t;

    if (type(c= *w++ = getch()) != LETTER) {
        *w = '\0';
        return(c);
    }
    while (--lim > 0) {
        t = type(c= *w++ = getch());
        if (t != LETTER && t != DIGIT) {
            ungetch(c);
            break;
        }
    }
    *(w-1) = '\0';
    return(LETTER);
}


/* Typ eines ASCII Zeichens liefern */
type(int c)
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
        return(LETTER);
    else if (c >= '0' && c <= '9')
        return(DIGIT);
    else
        return(c);
}
```

8.1.3 REKURSIVE STRUKTUREN

Rekursive Strukturen sind Strukturen, die zumindest ein Element enthalten, das auf eine andere Struktur gleichen Typs verweist. Sie werden sehr oft benötigt (z.B. verkettete Listen, Datenbäume). Eine Struktur kann sich zwar nicht selbst enthalten, wohl aber einen Zeiger auf eine Struktur gleichen Typs.

```
struct datum {
    int tag;
    int monat;
    int jahr;
    int jahrestag;
    char mon_name[4];
    struct datum heute;     /* FALSCH */
};


struct datum {
    int tag;
    int monat;
    int jahr;
    int jahrestag;
    char mon_name[4];
    struct datum *heute;      /* RICHTIG */
};
```

Zur Illustration soll das folgende Beispielprogramm für einen Binärbaum dienen.

Beispielprogramm P8-3.C

```
/* binaerer Baum */
struct tnode {
    char *word;   /* zeigt auf den Text */
    int count;    /* Haeufigkeit */
    struct tnode *left;   /* linker Nachfolger */
    struct tnode *right;  /* rechter Nachfolger */
};

#define MAXWORD 20


main()   /* Haeufigkeit von Worten zaehlen */
{
    struct tnode *root, *tree();
    char word[MAXWORD];
    int t;

    root = NULL;
    while ((t = getword(word,MAXWORD)) != EOF)
        if (t == LETTER)
            root = tree(root,word);
    treeprint(root);
}
```

```
/* w bei oder nach p einfuegen */
struct tnode *tree(struct tnode *p,char *w)
{
    struct tnode *talloc();
    char *strsave();
    int cond;

    if (p == NULL) {   /* ein neues Wort */
        p = talloc();  /* einen neuen Knoten anlegen */
        p->word = strsave(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w,p->word)) == 0)
        p->count++;     /* Wort wird wiederholt */
    else if (cond < 0)  /* kleiner, links darunter */
        p->left = tree(p->left,w);
    else                /* groesser, rechts darunter */
        p->right = tree(p->right,w);
    return(p);
}


/* Baum p rekursiv ausgeben */
treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n",p->count,p->word);
        treeprint(p->right);
    }
}


/* Platz für eine Struktur besorgen */
struct tnode *talloc()
{
    char *alloc();

    return((struct tnode *) alloc(sizeof(struct tnode)));
}


/* Zeichenkette s in Sicherheit bringen */
char *strsave(char *s)
{
   char *p, *alloc();
   if (p=alloc(strlen(s)+1)) != NULL)
      strcpy(p,s);
   return (p);
}
```

8.2 VARIANTEN (UNIONS)

Während eine Struktur mehrere Variablen (verschiedenen Typs) enthält, ist eine Variante eine Variable, die (aber natürlich nicht gleichzeitig) Objekte verschiedenen Typs speichern kann. Verschiedene Arten von Datenobjekten können so in einem einzigen Speicherbereich maschinenunabhängig manipuliert werden. (Eine solche Eigenschaften ist zum Beispiel sehr nützlich beim Aufbau von Symboltabellen in Übersetzern.)

Eine Variante wird durch das Schlüsselwort **union** vereinbart. Die weitere Syntax ist wie bei **struct**.

```
union u_tag {     /* Uniontyp u_tag */
    int ival;
    float fval;
    char *pval;
} uval;           /* Unionvariable uval */
```

Im Falle einer Variante (Union) muß man sich natürlich im Programm in einer anderen Variablen merken, welcher Datentyp in der Variante gerade abgelegt ist.

Beispiel:

```
if (utype == INT)
    printf("%d\n",uval.ival);
else if (utype == FLOAT)
    printf("%f\n",uval.fval);
else if (utype == STRING)
    printf("%s\n",uval.pval);
else
    printf("bad type %d in utype\n",utype);
```

Eine Struktur kann innerhalb einer Varianten vorkommen und umgekehrt:

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *pval;
    } uval;
} symtab[NSYM];

symtab[i].uval.ival    /* ival */
*symtab[i].uval.pval   /* 1. Zeichen von pval */
```

Ein wichtiges Beispiel für Struktur- und Variantenvereinbarung ist die Definition der Strukturen WORDREGS und BYTEREGS sowie der Varianten REGS für MS-DOS Funktionsaufrufe:

```
struct WORDREGS {
    unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
    unsigned int cflag;
};
struct BYTEREGS {
    unsigned char al,ah;
    unsigned char bl,bh;
    unsigned char cl,ch;
    unsigned char dl,dh;
};
union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};

union REGS inregs,outregs;

inregs.x.bx = 0x12;    /* BX Register auf Hex 12 stellen */
inregs.h.ah = 0x10     /* AH Register auf Hex 10 stellen */
c = outregs.x.cx       /* CX Register nach c kopieren */
```

8.3 BITFELDER

Zur platzsparenden Abspeicherung von Daten werden in bestimmten Fällen mehrere Objekte in einem Maschinenwort zusammengefaßt. Ein Objekt ist dann durch ein oder mehrere Bits diese Maschinenwortes repräsentiert (Beispiele: Initialisierungwerte für peripere Bausteine, Tabelle der ASCII-Zeichen mit ihren Eigenschaften (alpha, digit, hexdigit, print, usw.))

Für die Verarbeitung solcher Daten stehen die Operatoren für die bitweisen logischen Verknüpfungen zur Verfügung. Bestimmte Ausdrücke werden dabei häufig verwendet:

```
#define ALPHA 01
#define DIGIT 02
#define HEX 04

int attr;

attr |= ALPHA | HEX;       /* setzt Bits ALPHA und HEX */
attr &= ~(ALPHA | HEX);    /* löscht Bits ALPHA und HEX */
if ((attr & (ALPHA|HEX)) == 0)  /* genau dann wahr, wenn */
                           /* ALPHA und HEX gleich 0 sind */
```

Alternativ dazu bietet C die Möglichkeit, Bitwerte in Maschinenworten direkt zu definieren. Dazu wird die **struct**-Anweisung in abgewandelter Form verwendet:

```
struct {
    unsigned int is_alpha: 1;  /* Bit 0 */
    unsigned int is_digit: 1;  /* Bit 1 */
    unsigned int is_hex: 1;    /* Bit 2 */
                 : 3;          /* 3 Bit freilassen */
    unsigned int zei_satz: 2;  /* Bits 6 und 7 */
} attr;
```

Die Bitfeldvariable ist vom Typ **int**. Passen nicht alle Bitfelder in ein **int** Objekt, so wird dieses bis zur nächsten int Grenze erweitert. Die Zahl hinter dem **Doppelpunkt** ist die Anzahl der Bits dieser Komponente. Die Angabe nur eines Doppelpunktes ohne Komponentennamen ist erlaubt (namenlose Komponente) und kann als Platzhalter verwendet werden. Eine Bitbreite von 0 erzwingt die Fortsetzung des Bitfeldes auf der nächsten **int** Grenze.

Der Zugriff erfolgt wie bei Strukturen, die Komponenten sind hier kleine ganze Zahlen ohne Vorzeichen. Die 3 Bitmanipulationen mit bitweisen logischen Operatoren von oben kann man mit Bitfeldern so formulieren:

```
attr.is_alpha = attr.is_hex = 1;
attr.is_alpha = attr.is_hex = 0;
if ((attr.is_alpha==0) && (attr.is_hex == 0))
```

Achtung: Bitfelder haben keine Adressen und es gibt keine Vektoren von Bitfeldern!

8.4 AUFZÄHLUNGEN

In neueren Implementationen von C ist auch der Datentyp **Aufzählung** realisiert. Eine Aufzählungsvereinbarung wird durch das Schlüsselwort **enum** eingeleitet, es folgt der Aufzählungstypname, die in geschweiften Klammern eingeschlossen Aufzählungsliste und schließlich die Aufzählungsvariable, die eine der in der Aufzählungliste genannten Konstanten als Wert haben kann.

enum *Aufzählungstyp* { *Aufzählungsliste* } *Aufzählungsvar.* **;**

Beispiele:

```
enum farben { gelb, gruen, blau, rot} farbe;
enum farben col, *pcol;
```

Die Werte der Konstanten beginnen links bei 0 und erhöhen sich nach rechts jeweils um 1 (gelb=0, gruen=1, blau=2, usw.). Einer Aufzählungskonstanten kann aber auch direkt eine **int** Konstante zugewiesen werden, die Werte der weiter rechts stehenden Konstanten werden dann wieder je um 1 von diesem Wert erhöht.

```
enum farben {gelb,gruen=3,blau,rot,schwarz=8,braun};
```

braun hat also den Wert 9.

8.5 "TYPEDEF"

Mit **typedef** kann man neue Datentyp**namen** definieren (Nicht aber neue Datentypen!). typedef ist #define ähnlich, aber weitergehender, da erst vom Compiler verarbeitet und nicht nur einfacher Textersatz. Die Anwendungen von typedef liegen darin, ein Programm gegen Protabilitätsprobleme abzuschirmen und für eine bessere interne Dokumentation

Beispiel für Portabilitätsprobleme (int**):**

```
typedef int LENGTH;
typedef char *STRING;
LENGTH len, maxlen;
LENGTH *lengths[];
STRING p, alloc();
```

Beispiel für interne Dokumentation:

```
typedef struct tnode {
    char *word;
    int count;
    struct tnode left*;
    struct tnode *right;
} TREENODE, *TREEPTR;


typedef int (*PFI)();   /* das kann der Präprozessor nicht */
                        /* auswerten, Verwandschaft mit */
                        /* Prototyping bei Funktionen */
                        /* PFI ist der Datentyp für einen */
                        /* Zeiger auf eine Funktion, die */
                        /* ein int Resultat liefert */
```

9. ARBEITEN MIT DATEIEN

9.1 GRUNDLAGEN

Alle bisher besprochenen Programme waren lediglich in der Lage, von der Konsole (Tastatur bzw. Bildschirm) zu Lesen bzw. zu Schreiben. Aber unter UNIX (auch unter MS-DOS ab Version 2.0) ist eine Umleitung dieser Standard Ein-/Ausgabe möglich. Die Umleitung bezieht sich auf die Aufrufe der Funktionen **getchar**, **putchar**, **printf** und **scanf**.

Eine Umleitung der Standard Ein-/Ausgabe erfolgt mit den Umleitungssymbolen **<**, **>** und **>>** bzw. durch Pipe Bildung mit **|** beim Aufruf des Programmes.

Beispiele:

```
prog1                  /* prog1 liest von Tastatur und
                          schreibt auf Bildschirm */
prog1 >hugo            /* prog1 liest von Tastatur und
                          schreibt auf Datei hugo, die neu
                          angelegt wird */
prog1 <steuer          /* prog1 liest von Datei steuer und
                          schreibt auf Bildschirm */
prog1 <steuer >>hugo   /* prog1 liest von Datei steuer und
                          schreibt auf Datei hugo weiter */
prog1|prog2            /* prog1 liest von Tastatur und gibt
                          Ausgabedaten an prog2 weiter
                          prog2 liest Daten von prog1
                          und schreibt auf Bildschirm */
```

Die Umleitung erfolgt transparent, d.h. das Programm merkt nichts davon; die Umleitungsangaben tauchen auch nicht in **argv** auf.

Beim Arbeiten mit Dateien muß die Include-Datei **stdio.h** dazugeladen werden. Dies erreicht man mittels

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

stdio.h enthält wichtige Definitionen von Konstanten und einer Struktur für den Dateizugriff.

9.2 ÖFFNEN UND SCHLIEßEN VON DATEIEN

Natürlich kann man von einem C Programm aus auch auf jede beliebige Datei zugreifen. Für diese Zugriffe muß die Datei geöffnet und nach Abschluß der Dateiarbeiten wieder geschlossen werden. Zur Identifizierung einer Datei dient dabei ein sogenannter **Filepointer**, der beim erfolgreichen Öffnen der Datei als Wert der Öffnungsfunktion zurückgegeben wird.

Der **Filepointer** ist ein Zeiger auf eine Struktur, die Angaben über den Bearbeitungszustand der Datei enthält. Die Einzelheiten dieser Struktur, die in **stdio.h** definiert ist, sind für uns nicht wichtig. Die Vereinbarung:

```
FILE  *fp, *fopen();
```

definiert **fp** als einen Zeiger auf die Struktur FILE und **fopen()** als eine Funktion, die ein solches Resultat (einen Pointer auf eine Struktur FILE) liefert.

Eine Datei wird mittels

```
fp = fopen(name,mode);
```

geöffnet. **name** ist der Dateiname (genauer Pfadname, Achtung Sonderbedeutung von Backslash in C!) als Zeichenkette und **mode** kann die Werte **"r"** (Lesezugriff), **"w"** (Schreibzugriff) und **"a"** (anfügender Schreibzugriff) haben. Nach erfolgreichem Öffnen der Datei wird von fopen ein Zeiger auf eine Dateistruktur zurückgeliefert, ansonsten der Wert **NULL** (wie EOF in stdio.h definiert). War die Datei nicht schon vorhanden, so wird sie angelegt, allerdings nur bei Schreibzugriff (Das Lesen von nicht existierenden Dateien ist natürlich ein Fehler).

Eine geöffnete Datei wird bei Programmende oder -abbruch mit exit() automatisch geschlossen. Dies ist insbesondere bei Schreibzugriffen wichtig, da nur dann der Restpuffer auch wirklich in die Datei gelangt. Generell sollte man aber jede nicht mehr benötigte Datei mit

```
fclose(fp);
```

wieder schließen. **fp** muß dabei ein vorher von **fopen()** zurückgebener Filepointer sein.

Die Namen **stdin**, **stdout** und **stderr** sind vordefinierte konstante Filepointer, die man ohne Öffnen der zugehörigen Dateien verwenden kann, da das Öffnen beim Programmstart automatisch besorgt wird. Es gilt folgende Zuordnung:

```
stdin   /* Standardeingabe (Tastatur) Umlenkung möglich */
stdout  /* Standardausgabe (Bildschirm) Umlenkung möglich */
stderr  /* Fehlerausgabe (Bildschirm) keine Umlenkung */
```

9.3 DATEI EIN-/AUSGABE

Zur zeichenweisen Ein-/Ausgabe von/auf eine Datei, die über einen Filepointer identifiziert wird, stehen die Routinen **getc** und **putc** zur Verfügung.

```
c = getc(fp);     /* Lesen wie bei getchar */
putc(c,fp);       /* Schreiben wie bei putchar */
```

Sonderfall:

```
c = getc(stdin);     /* getchar */
putc(c,stdout);      /* putchar */
```

Zeilenweise Ein-/Ausgabe kann mittels **fgets** und **fputs** erledigt werden.

```
fgets(zeile,ZEILAE,fp);
```

liest die nächste Eingabezeile (bis zum und einschließlich des Zeilenendezeichens) von **fp** in den Vektor **zeile**. Es werden maximal **ZEILAE-1** Zeichen gelesen und **zeile** immer mit **'\0'** abgeschlossen. Funktionswert von **fgets** ist der Zeigerwert **zeile** bzw. **NULL**, wenn das Dateiende erreicht ist (vgl. getline()).

```
fputs(zeile,fp);
```

schreibt den Inhalt von **zeile** auf die Datei **fp**.

Beispielprogramm P9-1.C

```
/* Realisierung von fgets und fputs */
#include <stdio.h>

/* hoechstens n Zeichen ueber iop einlesen */
char *fgets(char *s,int n,register FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c=getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return((c == EOF) && cs == s) ? NULL : s);
}


/* Zeichenkette s ueber iop ausgeben */
fputs(register char *s,register FILE *iop)
{
    register int c;

    while (c = *s++)
        putc(c,iop);
}
```

Beispielprogramm P9-2.C

```
/* Realisierung des cat Kommandos */
#include <stdio.h>

/* cat: Dateien nacheinander ausgeben */
main(int argc,char *argv[])
{
    FILE *fp, *fopen();

    if (argc == 1)   /* ohne Argumente */
        filecopy(stdin);   /* Standard Eingabe kopieren */
    else
        while (--argc > 0)
            if ((fp=fopen(*++argv,"r")) == NULL) {
                fprintf(stderr,
                    "cat: can't open %s\n",*argv);
                exit(1);
            } else {
                filecopy(fp);
                fclose(fp);
            }
    exit(0);
}


filecopy(fp)   /* Datei fp als Standard Ausgabe ausgeben */
FILE *fp;
{
    int c;

    while ((c=getc(fp)) != EOF)
        putc(c,stdout);
}
```

Für die formatierte Datei Ein-/Ausgabe werden **fprintf** und **fscanf** verwendet. Sie arbeiten genauso wie **printf** und **scanf**, haben jedoch zusätzlich als ersten Parameter den Filepointer **fp**.