Programmieren in C

Kevelaer im Mai 1997

Dipl.-Ing. Reinhard Peters
Ber. Ing. für Informationsverarbeitung

Heideweg 60
47623 Kevelaer-Keylaer

Telefon 02832/6678
E-Mail: rpeters@pikt.de

www.pikt.de

Inhaltsverzeichnis

1 Einführung 5

2 Aufbau eines C-Programmes 6

2.1 Beispiel für ein 'C'-Programm 8

3 Wörter in C-Programmen 9

3.1 Trennzeichen 9
3.2 Namen 10
3.3 Reservierte Wörter 10
3.4 Konstanten 11
3.4.1 Ganzzahlige Konstanten 11
3.4.2 Gleitkommakonstanten 12
3.4.3 Zeichenkonstanten 12
3.4.4 Zeichenkettenkonstanten 13
3.5 Operatoren 14

4 Ausdrücke 16

4.1 L-Value 16
4.2 Unäre Operatoren 16
4.2.1 Logische Negation 16
4.2.2 Einer-Komplement (Bitweise Negation) 17
4.2.3 Inkrement, Dekrement 17
4.2.4 Cast-Operator 18
4.2.5 Sizeof-Operator 18
4.2.6 Adreßoperator 18
4.2.7 Verweis 18
4.3 Binäre Operatoren 19
4.3.1 Arithmetische Operatoren 19
4.3.2 Shift-Operatoren 20
4.3.3 Vergleichsoperatoren 20
4.3.4 Logische Operatoren (bitweise) 21
4.3.5 Logische Operatoren 21
4.3.6 Bedingungsoperator 22
4.3.7 Zuweisungsoperatoren 22
4.3.8 Komma-Operator 23

5 Elementare Datentypen 25

5.1 Deklarationen 30
5.2 Mixed mode 30
5.3 Weitere elementare Datentypen 32

6 Anweisungen 34

6.1 Leere Anweisung 34
6.2 Verbundanweisung 35
6.3 Zuweisung 36
6.4 if-Anweisung 36
6.5 switch-Anweisung 39
6.6 while-Anweisung 41
6.7 do .. while-Anweisung 43
6.8 for-Anweisung 44
6.9 break-Anweisung 45

6.10 continue-Anweisung 45
6.11 return-Anweisung 46
6.12 goto-Anweisung....46

7 Standard-Ein- und Ausgabe....48

7.1 printf( ) 49
7.2 putchar( ) 52
7.3 puts( ) 53
7.4 scanf( ) 55
7.5 getchar( ) 57
7.6 gets( ) 58
7.7 Variationen der vorgestellten Funktionen 58

8 Funktionen 59

8.1 Funktionsdefinition....59
8.2 Kommunikationskanäle zwischen Funktionen 61
8.3 Funktionsprototypen 63
8.4 Rekursion 65

9 Vektoren (Felder) 66

9.1 Initialisierung....67
9.2 Adressierung 68
9.3 Vektoren als Funktionsparameter....69

10 Zeiger....71

10.1 Zeiger auf Funktionen....73
10.2 Cast-Operator für Zeigertypen....74
10.3 Zusammenfassung....74

11 Strukturen....76

11.1 Dynamische Datenstrukturen 77

12 Bitfelder 83

13 Unions....85

13.1 Typdeklaration mit typedef 86

14 Dateien....87

14.1 Mit Hilfe der Systemaufrufe 87
14.2 Mit Hilfe der Standardbibliothek....91

15 Kommandozeilenparameter....95

16 Speicherklassen 98

16.1 Variablen 99
16.2 Funktionen....101
16.3 Modularisieren eines Programmes....101

17 Der Preprozessor....104

17.1 Die Anweisungen #define und #undef....105
17.2 Die Anweisung #include 107
17.3 Bedingte Compilierung 108

17.4 Die Anweisungen #line und #pragma 109

18 Die Standard-Bibliotheken 110

18.1 Die Include-Datei assert.h 112
18.2 Die Include-Datei ctype.h 112
18.3 Die Include-Datei float.h 114
18.4 Die Include-Datei limits.h 115
18.5 Die include-Datei math.h 116
18.6 Die Include-Datei setjmp.h 118
18.7 Die Include-Datei signal.h 118
18.8 Die Include-Datei stdarg.h 119
18.9 Die Include-Datei stdio.h 120
18.10 Die Include-Datei stdlib.h 125
18.11 Die Include-Datei string.h 128
18.12 Die Include-Datei time.h 131

19 Schlußwort 133

20 Quellennachweis 133

1 Einführung

Ziel des Kurses soll es sein, gemeinsam mit den Kursteilnehmern die wesentlichen Kennzeichen der Programmiersprache ‘C’ bzw. ‘C++’ und ihre Unterschiede zu anderen Programmiersprachen zu erarbeiten. Die Kursteilnehmer sollen zum Abschluß des Kurses in der Lage sein, selbstständig Probleme in der Programmiersprache ‘C’ bzw. ‘C++’ zu bearbeiten bzw. bestehende Programme ihren Bedürfnissen anzupassen.

Entwicklung

‘C’ ist eine Programmiersprache, die Anfang der 70'er Jahre entwickelt wurde, und die mit der Verbreitung des Betriebssystems UNIX zunehmend an Bedeutung gewonnen hat. UNIX wurde 1969 von Ken Thompson in den Bell-Laboratories der Firma AT&T (American Telefon and Telegraph, vergleichbar mit Siemens in Deutschland) entwickelt. 1971 wurde dieses Betriebssystem von Dennis Ritchie in 'C' formuliert, der gleichzeitig dazu diese Programmiersprache erst entwickelte. Hierher rührt es auch, daß C immer so eng mit UNIX verknüpft wird und bis vor wenigen Jahren nahezu unbeachtet geblieben ist. Inzwischen hat ‘C’ auch im PC-Bereich an Beudtung gewonnen, da es mittlerweile zur Standard-Programmiersprache im Bereich der Windows-Anwendungen geworden ist. ‘C’ ist auch die Basis für die objektorientierte Programmiersprache ‘C++’ alle ‘C’-Konstruktionen können Sie auch in den Quelltexten für ‘C++’-Programme verwenden. Dies hat den Vorteil, daß Sie bei der Umstellung auf ‘C++’ vorhandenen ‘C’-Code weiter benutzen können. ‘C++’ wurde von seinem Erfinder Bjarne Stroustrup für die Programmierung von ereignisgesteuerten Simulationen entwickelt. Aus einem ‘C’ mit Klassen wurde in einem evolutionären Prozeß ‘C++’. Gleichzeitig mit der Entwicklung von C++ wurde auch die Programmier-sprache ‘C’ weiterentwickelt. Insbesondere das amerikanische Normungs-Institut ANSI hat hier einen wichtigen Beitrag geleistet. Beide Entwicklungen haben sich gegenseitig beeinflußt.

Einordnung

'C' ist eine Programmiersprache, die von Ihren Entwickler für die Erstellung eines Betriebssystemes konzipiert wurde. Trotzdem ist 'C' nicht als eine anwendungs-orientierte Programmiersprache wie etwa COBOL oder FORTRAN zu verstehen und schon garnicht als eine Lehr- und Lernsprache wie etwa PASCAL zu sehen. Vielmehr ist 'C' eine Art Universalsprache.

Merkmale

'C' ist eine maschinennahe Sprache. Dies erkennt man an den Operatoren und den Datentypen. Komplexe Datentypen wie Zeichenketten (Strings) oder logische Datentypen (Boolean) gibt es nicht. Dafür gibt es den Datentyp Zeiger (Pointer) auf den sogar Rechenoperationen zugelassen sind. Auch die Ausdrücke und Operatoren erinnern oftmals stark an, von jedem Mikroprozessor her bekannte, Maschinenbefehle.

Sprachumfang

Der Sprachumfang von 'C' ist relativ gering. Viele, von anderen Sprachen her bekannte, Standardoperationen und Anweisungen werden in 'C' durch Funktionsaufrufe abgewickelt, so auch die Ein- und Ausgabeanweisungen. Diese Standardfunktionen finden Sie in den entsprechenden Bibliotheken.

VariablenTypprüfung

Im Gegensatz zu Pascal findet in 'C' beim Compilieren keine Typüberprüfung

Typprüfung

Im Gegensatz zu Pascal findet in 'C' beim Compilieren keine Typüberprüfung statt (Bei ‘C++’ sind die Regeln etwas strenger, hier werden auch Typ-Überprüfungen vorgenommen). Dies erfordert natürlich Disziplin vom Programmierer (Für die Typüberprüfung gibt es bei den meisten Systemen ein Dienstprogramm mit dem Namen lint, bei Borland-C oder MS-Visual-C++ lassen sich Überprüfungen durch den Compiler innerhalb der Entwicklungsumgebung einschalten). Wesentliche Bestandteile eines 'C'-Programms sind die Funktionen, und zwar nur Funktionen, wobei genau eine Funktion den Namen main hat, diese ist dann quasi das Hauptprogramm.

Preprozessor

Ein wesentlicher Bestandteil eines 'C'-Compilers ist der sogenannte Preprozessor. Er erlaubt mit Hilfe der Preprozessoranweisungen das Einbinden von Dateien (Include), das Ersetzen von Text und die Benutzung

von Makros. Der Preprozessor ist quasi ein Unterprogramm des Compilers, das vor dem eigentlichen Kompilieren den Quelltext überarbeitet.

Standards

Wesentlicher Vorteil von 'C' ist, daß die in 'C' geschriebenen Programme weitestgehend portabel sind. Für fast alle bekannten Rechner existieren heute bereits 'C'-Compiler und dank des geringen Sprachumfangs ist ein 'C'-Compiler auch relativ schnell zu implementieren. Außerdem ist 'C' weitestgehend standardisiert. Wobei man folgende Ebenen unterscheidet:

- Kerninghan & Ritchie-Standard (K & R)
- UNIX-Standard
- ANSI-Standard
- C++

K & R-Standard

Kerninghan und Ritchie definierten in ihrem Buch „The C programming language“ den ersten Standard. Dieses Buch heißt in der deutschen Ausgabe „Programmieren in C“ und ist 1983 beim Hanser-Verlag erschienen. Inzwischen ist eine neue Auflage des Buches erschienen, die auch den ANSI-Standard berücksichtigt (Hanser 1990). Die Definitionen von Kerninghan und Ritchie wurden zur Grundlage aller nachfolgend entwickelten C-Compiler.

UNIX-Standard

Auf UNIX-Systemen hat sich eine besondere Variante des K & R-Standards etabliert. Dieser Standard unterscheidet sich K & R-Standard im Wesentlichen durch andere bzw. erweiterte Funktionsbibliotheken.

ANSI-Standard

Darüber hinaus gibt es den sogenannten ANSI-Standard (ANSI - amerikanisches Normungsinstitut, vergleichbar mit dem DIN in Deutschland). Dieser Standard wird auch von den meisten heute erhältlichen Compilern gehalten. Er bietet zusätzlich zu den zuvor definierten Standards Mechanismen zur Syntaxprüfung.

C++-Standard

C++ ist die objektorientierte Weiterentwicklung von C. Die eigentliche Programmiersprache C ist eine echte Untermenge von C++. Neben den zusätzlichen Mechanismen für die objektorientierte Programmierung ist ein wesentliches Merkmal von C++, daß im Gegensatz zum ursprünglichen C strenge Konventionen für die Behandlung der Datentypen gelten. Die gängigen Entwicklungs-Umgebungen für Windows, z. B. Borland-C++ und MS-Visual C++ erfüllen heute alle diesen erweiterten Standard, zum Teil jedoch mit hersteller-spezifischen Besonderheiten, insbesondere was die vordefinierten Klassen-Bibliotheken angeht.

Portierung

Wenn man also dem Standard entsprechende Programme schreibt, dürfte das größte Problem bei der Portierung auf andere Systeme das Übertragen der Quelltexte sein.

Mit dem zunehmenden Bekanntheitsgrad des Betriebssystems UNIX fand auch 'C' eine immer größere Verbreitung. Heute ist 'C' auch aus der Welt der Personalcomputer und der Steuerungsrechner nicht mehr wegzudenken. Viele bekannte Programmpakete werden heute in 'C' erstellt, sicherlich auch mit Blick auf die bessere Portabilität.

2 Aufbau eines C-Programmes

Module

'C'-Programme bestehen aus Modulen. Die Module sind dadurch gekennzeichnet, daß für jedes Modul eine eigene Datei zur Verfügung steht. Kleine und mittlere 'C'-Programme bestehen in der Regel aus einem Modul.

Ein Modul kann enthalten:

- Preprozessoranweisungen (erstes Zeichen in jeder Zeile: #)
- Globale Vereinbarungen (global für das jeweilige Modul)
- Funktionen

Funktionen

Unterprogramme heißen in 'C' grundsätzlich Funktionen. Eine Funktion muß mindestens existieren. Diese Funktion heißt main und ist quasi das Hauptprogramm.

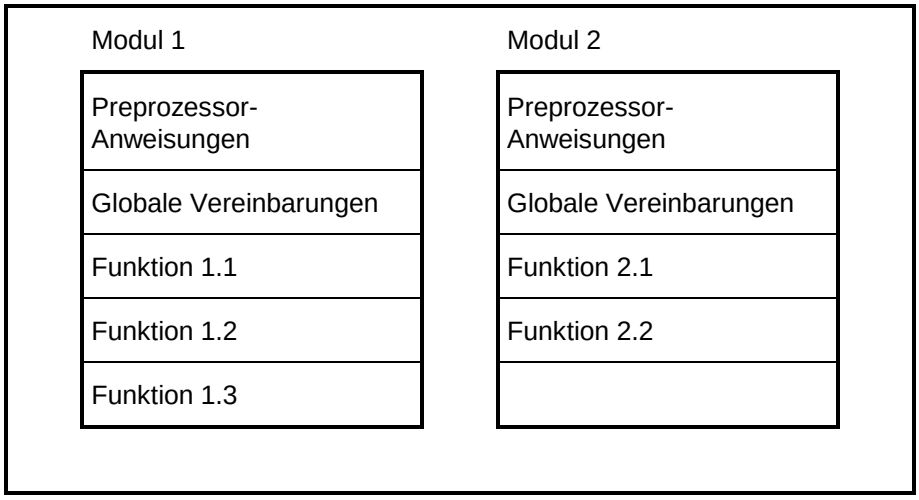
main

Die Anordnung der Funktion main innerhalb des Quelltextes spielt keine Rolle, d.h. speziell sie braucht nicht wie etwa in Pascal-Programmen am Ende der Liste aller Funktionen zu stehen. Allerdings ist es üblich die Funktion main am Anfang anzuordnen.

Prozeduren

Funktionen, die keinen Funktionswert zurückliefern sollen (d.h. wie Prozeduren gehandhabt werden sollen), können mit dem Typ void versehen werden. Da Funktionen in 'C' auch Ausdrücke sein können, können sie letzlich auch wie die von Pascal her bekannten Prozeduren eingesetzt werden.

Beispiel für den Aufbau eines 'C'-Programmes:



Jede der Funktionen könnte die Funktion main sein. Jede Funktion könnte jede Funktion mit Ausnahme von main aufrufen (d.h. auch Rekursionen sind erlaubt). Die Funktion main kann nicht von sich selbst oder einer anderen Funktion aufgerufen werden, sie wird vielmehr durch das Betriebssystem beim Starten des Programms aufgerufen.

2.1 Beispiel für ein 'C'-Programm

Beispiel

Dieses Programm wird im Buch von Kernighan & Ritchie ebenfalls als Einführungsbeispiel benutzt und zeigt bereits einige wichtige für alle 'C'-Programme geltende Merkmale.

```
1.  /* Berühmtes 'C'-Programm */
2.
3.  #include  <stdio.h>
4.
5.  main()
6.  {
7.    printf("hello, world\n");
8.  }
```

Wirkung

Durch das Programm wird der Text 'hello, world' mit einem abschließenden CR-LF (Carriage Return - Line Feed = Wagenrücklauf und Zeilenvorschub) auf dem Standardausgabegerät (in der Regel der Bildschirm) ausgegeben.

Die Zeilennummern dienen nur zur besseren Beschreibung des Programmes bei den nachfolgenden Erläuterungen, sie gehören nicht zum C-Quelltext.

Layout/Format

'C'-Programme sind (wie Pascal-Programme) formatfrei, d.h. sie sind nicht, wie etwa Fortran-Programme, an bestimmte Formatvorgaben gebunden. Sie können also die Anweisungen und Wörter beliebig im Quelltext plazieren und an jeder beliebigen Stelle Kommentare einfügen.

Die Zeile 1 enthält einen Kommentar, dies wird durch die Zeichenkombination /* bzw. */ angezeigt.

Die Zeile 3 enthält eine Preprozessoranweisung, das erkennen wir an dem führende #-Zeichen. Es wird die Header-Datei mit den Funktionsprototypen für die Standard-Ein- und Ausgabe (stdio.h) eingebunden.

Die Zeilen 5 bis 8 enthalten die Funktion main. Wobei als erstes in Zeile 5 der Name der Funktion angegeben wird. Sie erkennen dies an den runden Klammern, diese müssen immer angegeben werden, auch wenn keine Parameter übergeben werden sollen. Die geschweiften Klammern entsprechen dem BEGIN ({) und dem END (}) in Pascal. Sie zeigen den Anfang und das Ende eines Blockes (hier Funktionsblock) an.

Die Zeile 6 enthält die Funktion zur formatierten Ausgabe. Diese Funktion ist in der Datei stdio.h beschrieben und wird hier als Standardfunktion ähnlich der write-Anweisung in Pascal eingesetzt. Der auszugebende Text steht in doppelten Anführungszeichen (keine Hochkommas). Die Zeichenfolge \n ist ein Steuerzeichen, das einen Zeilenvorschub (CR-LF; n=newline) bewirkt. Alle Steuerzeichen werden durch das sogenannte ESCAPE-Zeichen (\) eingeleitet. Weitere Steuerzeichen werden wir später kennenlernen.

Aufgabe 1

Versuchen Sie mit dem an Ihrem Arbeitsplatz vorhandenen Entwicklungssystem das oben angegebene Beispielprogramm einzugeben, übersetzen und ausführen zu lassen. Diese Aufgabe soll Sie in den Umgang mit Ihrem Entwicklungssystem einführen.

3 Wörter in C-Programmen

Wörter

'C'-Programme bestehen aus Wörtern. Die Wörter sind quasi die Atome eines 'C'-Programmes.

Wir unterscheiden:

- Preprozessoranweisungen
  Diese beginnen immer mit einem #-Zeichen, z.B.: #include, #define. Beachten Sie bitte, daß zwischen dem #-Zeichen und der eigentlichen Preprozessoranweisung kein Leerzeichen stehen darf.
- 5 weitere Wortklassen (eigentliche Wörter)
  Es sind dies:
  - Trennzeichen
  - Namen
  - reservierte Wörter
  - Konstanten
  - Operatoren

Syntaxdiagramme

Diese Wörter werden in eine sinnvolle Folge gebracht und bilden dann das Programm. Die Regeln, die die Folgen der Wörter bestimmen, sind die Syntaxregeln. Sie werden in Form von Syntaxdiagrammen oder seltener in der sogenannten Backus-Nauer-Form (BNF) angegeben. Syntaxdiagramme zu 'C' finden Sie als Anlage unter dem in Kapitel 1 genannten Buch von Kerninghan und Ritchie. Eine, der Backus-Nauer-Form ähnliche, Form wird in der Sprachbeschreibung von 'C' benutzt, die ebenfalls Bestandteil des Buches von Kerninghan und Ritchie ist. Wir wollen nun die einzelnen Wortklassen etwas genauer betrachten.

3.1 Trennzeichen

Trennzeichen

Es sind dies:

- Blank (Leerzeichen)
- Tab (Tabulator)
- Zeilentrenner
- Kommentare

Trennzeichen sind Zeichen, die vom Compiler ignoriert werden. Daraus folgt, daß ein C-Programm nicht an ein bestimmtes Format gebunden ist, denn die Trennzeichen werden vom Compiler nicht bewertet, anders als z.B. bei Fortran.

Kommentare werden zwischen die beiden Zeichenkombinationen /* und */ geschrieben.

Beispiel

Beispiel für einen Kommentar:

```
/* Ein beruehmtes C-Programm */
```

3.2 Namen

Ein Name kann nur aus Buchstaben, Ziffern und dem Unterstrich (_) bestehen. Ein Name muß mit einem Buchstaben oder einem Unterstrich beginnen.

Beispiel

Beispiele für gültige Namen :

```
x
_x
x1
 _x1
otto_karl
__c
```

Hinweis

Beachten Sie, daß 'C' case-sensitiv ist, d. h. in 'C' wird, im Gegensatz zu Pascal oder zu MS-DOS, Groß- und Kleinschreibung unterschieden.

Beispiel

Die Bezeichner

```
null
NULL
Null
NUll
nuLL
```

sind 5 verschiedene Namen!

Hinweis

In der Regel werden alle Namen klein geschrieben. Selbstdefinierte Konstanten werden groß geschrieben. Vom System bereitgestellte Konstanten beginnen oftmals mit einem Unterstrich.

Bei einigen älteren 'C'-Compilern werden nur die ersten 8 Stellen des Namens zur Identifizierung genutzt. Speziell bei Borland-C++ und MS-Visual-C++ werden die ersten 32 Stellen ausgewertet.

3.3 Reservierte Wörter

Reservierte Wörter

Vom Programmierer vergebene Namen (Wörter) dürfen nicht mit den, im folgenden aufgeführten, reservierten Wörtern übereinstimmen.

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

C++

Hier werden die meisten von Ihnen schon viele alte Bekannte wiedererkennen. Desweiteren gibt es zusätzliche reservierte Wörter für ‘C++’-Compiler:

class	**const**	**delete**	**friend**	**inline**
inline	**new**	**operator**	**private**	**protected**
public	**signed**	**this**	**virtual**	**volatile**

Auf die Bedeutung der Wörter an passender Stelle näher eingegangen.

Zusätzlich sind speziell in Borland-C++ bzw. MS-Visual-C++ folgende weiteren Wörter reserviert:

asm	**cdecl**	**far**	**fortran**	**huge**
near	**pascal**	**interrupt**		

Außerdem sind folgende Bezeichner für die Prozessorregister reserviert:

_cs	**_ds**	**_es**	**_ss**	**_AH**
_AL	**_AX**	**_BH**	**_BL**	**_BX**
_CH	**_CL**	**_CX**	**_DH**	**_DL**
_DX	**_BP**	**_DI**	**_SI**	**_SP**

Die für Ihren Compiler gültige Liste mit reservierten Wörtern entnehmen Sie bitte dem Programmierhandbuch zu Ihrem Compiler. So sind z. B. bei MS-Visual-C++ zahlreiche weitere Wörter zur Steuerung des Verhaltens des Compilers definiert. Meist besteht jedoch keine Gefahr die entsprechenden Namen versehentlich zu benutzen, da diese von doppelten Unterstrichen eingeleitet werden, z. B. __saveregs, _segment, __cplusplus, __MSDOS, __time usw.

3.4 Konstanten

Wir unterscheiden folgende Arten von Konstanten:

- ganzzahlige Konstanten
- Gleitkommakonstanten
- Zeichenkonstanten
- Zeichenkettenkonstanten (Strings)

3.4.1 Ganzzahlige Konstanten

Beispiel

Beispiele für ganzzahlige Konstanten sind:

```
13
1
5432
-789
017
01000
0x45
0xaf
```

Die ersten vier Zahlen sind Beispiele für Konstanten, wie wir sie auch von anderen Programmiersprachen her kennen. Sie beschreiben eine ganzzahlige Zahl im Dezimalsystem. Die Konstanten, die mit einer führenden Null beginnen, sind Konstanten im Oktalsystem (Basis 8). Sie dürfen nur Ziffern von 0 bis 7 enthalten. Die mit 0x beginnenden Konstanten beschreiben Ziffern im Hexadezimalsystem (Basis 16). Sie dürfen alle Ziffern und die Buchstaben a bis f (bzw. A bis F, entspricht 10 bis 15) enthalten.

Abhängig vom Wert der Konstanten werden für diese 16 oder 32 Bit Speicherplatz reserviert. Durch Anfügen eines l bzw. L kann die Speicherung im 32-Bit-Format (long) erzwungen werden, auch wenn der Wert kleiner als 32768 ist.

3.4.2 Gleitkommakonstanten

Beispiele für Gleitkommakonstanten sind:

```
11.0
-0.032
.456
456.
1.50000e+5
5E-10
```

Alle Konstanten entsprechen den dargestellten Zahlen im Dezimal-System. Die E-Formate sollten von anderen Programmiersprachen und vom Taschenrechner her bekannt sein. Man kann nichts falsch machen, wenn man die von Pascal her bekannten Konventionen übernimmt. Es gibt allerdings Unterschiede. So können Gleitkommakonstanten mit einem führenden Punkt beginnen oder auf einen Punkt enden. Dies ist in Pascal nicht erlaubt.

Gleitkommakonstanten enthalten in jedem Fall einen Punkt oder ein e bzw. E, durch diese Zeichen unterscheiden sie sich von den ganzzahligen Konstanten. Gleitkommakonstante werden grundsätzlich im 64-Bit-Format gespeichert. In Ansi-C kann durch Nachstellen eines f bzw. F auch das 32-Bit-Format erzwungen werden.

3.4.3 Zeichenkonstanten

Beispiele für Zeichenkonstanten sind:

```
'a'
'x'
'Z'
'3'
'\n'
```

Zeichen werden im Speicher genau durch 8 Bit repräsentiert. Bei MS-DOS-Rechnern findet in der Regel der sogenannte IBM-Zeichensatz Anwendung. Der IBM-Zeichensatz ist ein erweiterter ASCII-Zeichensatz ist. Einige Steuerzeichen lassen sich mit Hilfe des ESCAPE-Zeichen (\) direkt eingeben. Es sind dies:

`\a`	Piepton (Turbo-C)	`\b`	Backspace
`\f`	Formfeed (Seitenvorschub)	`\n`	CR-LF
`\r`	CR	`\t`	Tab (Tabulator, horizontal)
`\v`	Tabulator, vertikal	`\'`	'-Zeichen
`\"`	"-Zeichen	`\\`	\-Zeichen

Außerdem besteht die Möglichkeit nach dem \-Zeichen den ASCII-Code anzugeben und so Zeichen auszugeben, die nicht über die Tastatur eingegeben werden können.

Allgemeine Form `\ooo` oder `\xhh`

Dabei bedeuten:

- `ooo` dreistellige Oktalzahl
- `hh` zweistellige Hexadezimalzahl

Beispiel

`\007` = `\x07` = `\a` (Piepton)

`\375` = `\xFD` = Ω (ASCII-Code = 234)

3.4.4 Zeichenkettenkonstanten

Beispiele für Zeichenkettenkonstanten sind:

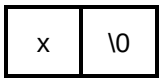
```
"x"
"dies ist eine Zeichenkette \n"
```

Eine Zeichenkettenkonstante ist eine Folge von beliebigen Zeichen wie sie auch unter 3.4.3 verwendet werden können. Es können also auch die Steuerzeichen in einer Zeichenkette benutzt werden. Die Zeichenkette benötigt intern gerade soviel Bytes, wie sie Zeichen enthält plus einem zusätzlichen Zeichen, das das Ende der Zeichenkette anzeigt ('\0', ASCII-Zeichen mit dem Code 0). Eine Zeichenkettenkonstante ist also immer mindestens ein Byte lang (leere Zeichenkette). Man nennt diese Art von Zeichenkettenkonstanten auch nullterminierte Zeichenketten.

Hinweis ***'x' und "x" sind nicht dasselbe. Intern wird 'x' als ein Byte abgespeichert (Zeichen x), während "x" in zwei Byte (Zeichen x und Zeichen \0) abgespeichert wird.***

Abspeicherung von 'x' in genau einem Byte:

Abspeicherung von "x" in zwei Byte. Der Zeichenkette wird als Erkennungszeichen für das Ende immer ein '\0' angehängt.



Aufgabe 2 Was wird ausgegeben ?

a) `printf ( "\157" );`

b) `printf ( "\0157" );`

3.5 Operatoren

Rangfolge

Operatoren haben in 'C' eine genau definierte Rangfolge. Wir unterscheiden:

- primäre Operatoren
- unäre Operatoren
- binäre Operatoren

Die Reihenfolge der Aufzählung spiegelt auch die Rangfolge wieder.

Assoziativität

Außerdem unterscheidet man rechts- und linksassoziative Operatoren.

Beispiel

```
y = x = a + b + c
```

In diesem Ausdruck gibt es zwei Operatoren + und =. Der Ausdruck wird wie folgt ausgewertet: Zuerst wird a und b addiert, danach dieses Ergebnis zu c addiert und anschließend wird dieses Ergebnis x und danach y zugewiesen. Das + ist ein linksassoziativer Operator, deshalb wird der Ausdruck von links nach rechts ausgewertet. Das = dagegen ist rechtsassoziativ, deshalb wird das Ergebnis zuerst dem x und dann dem y zugewiesen. Der Ausdruck wird von rechts nach links ausgewertet. Wenn man bewußt Klammern setzen würde, um die Rangfolge der Operatoren zu beschreiben, so würde dies so aussehen:

```
(y= (x= ( (a+b) +c) ) ) )
```

Die folgende Tabelle zeigt mit der höchsten Priorität oben beginnend die möglichen Operatoren in ihrer Rangfolge. Wobei Operatoren mit dem gleichen Rang in einer Zeile dargestellt sind.

Primäre Operatoren	`( ) [ ] ->`	linksassoziativ
Unäre Operatoren	`! ~ ++ -- - (typ) * & sizeof()`	rechtsassoziativ
Binäre Operatoren	`* / %`	linksassoziativ
	`+ -`	linksassoziativ
	`<< >>`	linksassoziativ
	`< > <= >=`	linksassoziativ
	`== !=`	linksassoziativ
	`&`	linksassoziativ
	`^`	linksassoziativ
	`|`	linksassoziativ
	`&&`	linksassoziativ
	`||`	linksassoziativ
	`?.. :..`	rechtsassoziativ
	`op=`	rechtsassoziativ
	`,`	linksassoziativ

Für typ kann jeder beliebige Datentyp eingesetzt werden. (typ) ist der sogenannte Cast-Operator.

Für op können eingesetzt werden: `+ - * / % >> >> & | ^`.

Die Bedeutung der einzelnen Operatoren wird im Kapitel 4 erläutert.

Wichtig ist diese Tabelle für die Bestimmung der Rangfolge der Operatoren.

Aufgabe 3

Setzen Sie Klammern in den Ausdrücken, um die Rangfolge der Operatoren zu verdeutlichen.

a) `a += *a + b`

b) `z += y = x = c * a + b`

c) `z = y = a == b == c`

C++

Für C++ sind noch einige zusätzliche Operatoren definiert. Deren Bedeutung wird im Zusammenhang mit den Ausführungen zu den Klassen, bzw. den Funktionen zur Speicherreservierung näher erläutert. Dies sind die Operatoren:

`.* ->* :: new delete`

4 Ausdrücke

Einfache Ausdrücke sind aus Namen und Operatoren zusammengesetzt. Darüber hinaus gibt es komplexere Ausdrücke, die in anderen Kapiteln näher beschrieben werden. Ausdrücke können außerdem enthalten:

- (Ausdruck)
- Name(Argumentliste) Funktionen
- Name[Ausdruck] Vektoren
- Ausdruck.Name Strukturen
- Ausdruck->Name Verweise

Wichtig: Jeder Ausdruck besitzt einen Wert.

Logische Ausdrücke Wie Sie sehen werden, gibt es auch logische Operationen, obwohl wir in Kapitel 1 gesehen, daß es keinen speziellen logischen Datentyp wie etwa Boolean gibt. Die logischen Werte sind wie folgt definiert:

- true jede Zahl <> 0
- false die Zahl 0

Logische Operationen liefern als Ergebnis entweder 0 oder 1.

4.1 L-Value

L-Value Ein L-Value ist ein Ausdruck oder eine Variable, der bzw. die, vereinfacht gesagt, auf der linken Seite eines Zuweisungsoperators stehen darf (L=left, links). Ein L-Value repräsentiert einen Speicherbereich, in den Werte übertragen werden können. Einem L-Value kann also ein Wert zugewiesen werden.

Bei manchen Operatoren ist die Angabe eines L-Values als Operand zwingend vorgeschrieben.

4.2 Unäre Operatoren

Die unären Operatoren haben nur einen Operanden. Das klassische Beispiel für einen unären Operator ist das Minuszeichen, das aus einer positiven eine negative Zahl macht und umgekehrt.

4.2.1 Logische Negation

Der logische Negationsoperator wandelt einen Wert ungleich Null in eine Null und den Wert Null in Eins.

Beispiele

`9`	entspricht true
`!9 = 0`	entspricht false
`!!9 = !(!9) = !0 = 1`	entspricht true

4.2.2 Einer-Komplement (Bitweise Negation)

Das Einerkomplement entspricht der bitweisen Negation einer Größe. In der binären Darstellung werden alle Nullen zu Einsen und alle Einsen zu Nullen. Das folgende Beispiel soll dies anhand einer char-Größe (8 Bit) verdeutlichen.

Beispiele

3	entspricht	0000 0011	
~3	entspricht	1111 1100	(= -4)

4.2.3 Inkrement, Dekrement

Der Inkrement- bzw. Dekrement-Operator addiert zum Operanden bzw. subtrahiert vom Operanden eine Eins. Wir unterscheiden die folgenden Operatoren:

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

Außerdem unterscheiden wir je nach dem, ob der Operator vor oder hinter dem Operanden steht:

`++k`	Preinkrement	`k++`	Postinkrement
`--k`	Predekrement	`k--`	Postdekrement

Die beiden Variationen der Operatoren (Pre... und Post...) unterscheiden sich durch den Zeitpunkt zu dem die Veränderung des Operanden stattfindet. Wenn die Operatoren innerhalb eines komplexeren Ausdruckes eingesetzt werden, verhalten sie sich wie folgt:

- Bei Pre... wird zuerst der Wert des Operanden verändert und dann der Wert des Ausdruckes bestimmt.
- Bei Post... wird der Wert des Ausdruckes bestimmt und danach erst der Operand verändert.

Wichtig: Diese Operation ist nur mit einem L-Value als Operand möglich.

Beispiele

'C'	Pascal
`k=3;`	`k:=3;`
`y=++k;`	`k:=k+1;`
	`y:=k;`

⇒ y=4 und k=4

`k:=3;`	`k:=3;`
`y=k++;`	`y:=k;`
	`k:=k+1;`

⇒ y=3 und k=4

4.2.4 Cast-Operator

Mit dem Cast-Operator können Sie den Typ einer Variablen oder eines Ausdruckes anpassen.

Beispiel

```
(double) a
```

Wenn z.B. die Variable a den Typ int hat (16 Bit, ganzzahlig), dann kann mit dem obigen Beispiel eine Typanpassung vorgenommen werden. Der ganze Ausdruck (also mit dem Cast-Operator) hat dann den Typ double (64 Bit, Gleitkomma). Der Typ der Variablen, auf die der Cast-Operator angewendet wird, wird dadurch nicht verändert.

4.2.5 Sizeof-Operator

Mit dem Sizeof-Operator können Sie den Speicherbedarf eines Ausdruckes, einer Variablen oder eines Datentyps in Byte ermitteln.

Beispiele

```
sizeof(int)
sizeof(a)
```

Dieses Beispiel sieht zwar so aus, als handele es sich bei sizeof um eine Funktion, es ist aber ein Operator.

Im ersten Beispiel hat der Ausdruck den Wert 2, denn Variablen vom Typ int benötigen zur Speicherung 16 Bit, dies entspricht 2 Byte. Im zweiten Beispiel wollen wir annehmen, daß die Variable a den Typ double (64 Bit, Gleitkomma) hat. Dann hat der Ausdruck den Wert 8.

4.2.6 Adreßoperator

Der Adreßoperator liefert die Adresse einer Speicherstelle.

Beispiel

```
&a
```

Dieser Ausdruck liefert die Adresse der Variablen a.

Wichtig: Der Operand muß ein L-Value sein.

4.2.7 Verweis

Der Verweisoperator ist der zum Adreßoperator inverse Operator. Er dereferenziert eine Adresse und liefert somit den Wert, der in der angegebenen Speicherstelle abgelegt ist.

Beispiel

```
*p
```

Wenn p ein Zeiger ist (also die Adresse einer Speicherstelle), so hat der Ausdruck *p den Wert der Zelle auf die p zeigt.

Beispiel

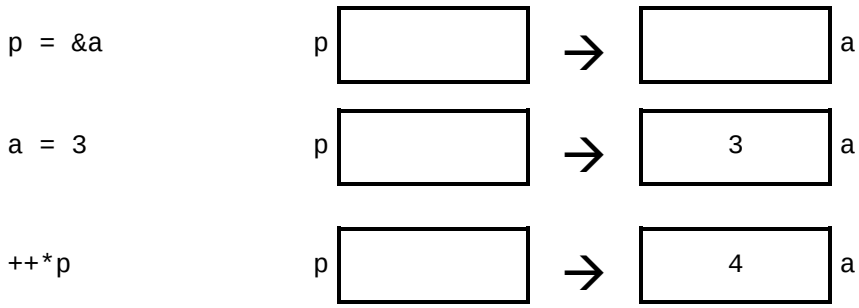
p sei ein Zeiger auf int

a sei eine int-Variable

Dazu ist folgende Deklaration nötig:

```
int a, *p;
```

Es sollen nun die nachfolgenden Operationen durchgeführt werden:



Durch die erste Zuweisung wird die Adresse der Variablen a in der Zeigervariablen p gespeichert. Damit zeigt p auf die Speicherzelle der Variablen a. Durch die zweite Zuweisung wird der Variablen a der Wert 3 zugewiesen. Durch die dritte Zuweisung wird der Wert in der Speicherzelle, auf die p zeigt, inkrementiert. Da die beiden Operatoren * und ++ den gleichen Rang haben und rechtsassoziativ sind, hat der Operator * Vorrang.

4.3 Binäre Operatoren

Die binären Operatoren haben immer zwei Operanden. Klassisches Beispiel für einen binären Operator ist das Pluszeichen mit dem zwei Zahlen addiert werden. Ein Ausdruck bestehend aus zwei Operanden und dem Pluszeichen bildet eine Summe. Der Wert des gesamten Ausdruckes ist die Summe der beiden Operanden.

4.3.1 Arithmetische Operatoren

Die arithmetischen Operatoren dienen zur Bildung arithmetischer Ausdrücke, z. B. für Berechnungen. Folgende Operatoren sind in C verfügbar:

```
*   /   %
+   -                (haben niedrigere Priorität)
```

Die Operatoren *, /, + und - dürften Ihnen bekannt sein. Der einzige Operator, der einer Erläuterung bedarf ist der Operator %. Ein Ausdruck mit diesem Operator liefert den Divisionsrest.

Beispiel

```
113  %  5        ergibt 3
```

Dieser Operator entspricht dem mod in Pascal.

4.3.2 Shift-Operatoren

Diese Operatoren ermöglichen die bitweise Verschiebung um die angegeben Zahl von Stellen. Auf der linken Seite steht der zu verschiebende Operand. Auf der rechten Seite steht die Anzahl der Stellen um die der linke Operand verschoben wird. Wir unterscheiden zwei Operatoren:

```
<<    >>
```

Die Pfeile bestimmen die Schieberichtung. Es erfolgt eine bitweise Verschiebung um die angegebene Zahl von Stellen.

Beispiel (für 8 Bit)

```
k = 2;
y = 3;                  ⇒    y = 00000011
y = y << ++k;           ⇒    y = 00011000
```

⇒ y hat den Wert 24

Hinweis Eine Verschiebung um eine Stelle nach links entspricht der Multiplikation mit 2, um 2 Stellen der Multiplikation mit 4, um 3 Stellen der Multiplikation mit 8 usw.; eine Verschiebung nach rechts entspricht der Division durch 2.

4.3.3 Vergleichsoperatoren

Die Vergleichsoperatoren dienen zum Vergleichen der beiden Operanden. Ausdrücke mit Vergleichsoperatoren liefern logisch zu interpretierende Ergebnisse. Die Ausdrücke haben je nach Ergebnis des Vergleichs entweder den Wert Null oder den Wert Eins. Die folgenden Operatoren stehen zur Verfügung:

```
>    <    >=   <=
==   !=              (haben niedrigere Priorität)
```

Die Operatoren haben folgende Bedeutung:

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

Wichtig: Ein einfaches Gleichheitszeichen (=) erfüllt nicht die Funktion von == !
Hier liegt eine sehr häufige Fehlerquelle, gerade für Umsteiger von anderen Programmiersprachen.

4.3.4 Logische Operatoren (bitweise)

Die bitweisen, logischen Operatoren dienen der bitweisen Verknüpfung zweier Größen. Diese Operatoren werden vorzugsweise bei der systemnahen Programmierung zum Ausmaskieren oder Abprüfen einzelner Bits eingesetzt. Folgende Operatoren sind verfügbar (in der Reihefolge ihrer Priorität):

&	UND
^	Exklusiv ODER (XOR)
|	ODER

Beispiel Ergebnisse der Operationen am Beispiel von 4-Bit-Zahlen:

	a&b	a|b	a^b
a	1010	1010	1010
b	1100	1100	1100
Ergebnis	1000	1110	0110

4.3.5 Logische Operatoren

Die logischen Operatoren dienen der Verknüpfung von logischen Ausdrücken. Bei diesen Operatoren werden die Operanden logisch interpretiert (Null entspricht false und ungleich Null entspricht true). Entsprechend sind die Ergebnisse dieser Ausdrücke in Abhängigkeit von ihren Operanden 0 oder 1. Folgende Operatoren stehen zur Verfügung:

&&	UND
||	ODER

Beispiel

```
(3 > 4) && (4 < 5) || (7 > 6 )
```

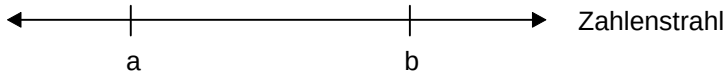
Der &&-Operator hat einen höheren Rang als ||-Operator. Deshalb wird zunächst der linke Teilausdruck ausgewertet. Der Ausdruck (3 > 4) ist falsch und liefert als Ergebnis eine logisch zu interpretierende Null. Der Klammerausdruck (4 < 5) ist wahr und liefert eine logisch zu interpretierende Eins. Die UND-Verknüpfung der beiden Ausdrücke liefert eine logisch zu interpretierende Null, da eine UND-Verknüpfung nur dann wahr ist, wenn beide Operanden wahr sind. Das Ergebnis des linken Teilausdruckes wird danach mit dem Ergebnis des letzten Klammerausdruckes ODER-verknüpft. Dieser hat den logisch zu interpretierenden Wert Eins. Die ODER-Verknüpfung ergibt somit eine Eins, da eine ODER-Verknüpfung immer dann wahr ist, wenn mindestens einer der Operanden wahr ist. Das Ergebnis des Gesamtausdrucks ist damit dann eine logisch zu interpretierende Eins.

4.3.6 Bedingungsoperator

Der Bedingungsoperator ermöglicht die wahlweise Ausführung eines Teilausdrucks. Dies wird von der Bedingung, die vor einem Fragezeichen stehen muß, gesteuert. Hinter dem Fragezeichen steht der Ausdruck, der ausgewertet wird, wenn die Bedingung erfüllt ist. Hinter einem Doppelpunkt steht der Teilausdruck, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Der gesamte Ausdruck hat den Wert des ausgewerteten Ausdruckes.

Beispiel

```
x = x > b ? b : x > a ? x : a
```



Nach der Auswertung des Ausdruckes hat x in jedem Fall einen Wert der in dem Intervall [a,b] liegt. Für den Fall, daß x > b ist, erhält der Ausdruck, rechts neben dem Zuweisungsoperator =, den Wert b. Andernfalls muß der zweite Bedingungsausdruck noch ausgwertet werden. Bei diesem Ausdruck erhält der Gesamtausdruck den Wert x für den Fall, daß x größer a ist, sonst erhält er den Wert a.

4.3.7 Zuweisungsoperatoren

Der Zuweisungsoperator dient dem Übertragen von Werten in die durch den linken Operanden bezeichnete Speicherzelle. Er entspricht im wesentlichen der Pascal-Anweisung `:=`. Der Zuweisungsoperator in C bietet aber noch einige zusätzliche Möglichkeiten:

- Es können mehrere Zuweisungsoperatoren in einem Ausdruck auftreten:

Beispiel

```
x = y = z = 3
```

oder zur besseren Übersicht

```
( x = ( y = ( z = 3 ) ) )
```

Alle Variablen bekommen den Wert 3. Der Operator ist rechtsassoziativ.

- Der Zuweisungsoperator kann in Verbindung mit einem der folgenden Operatoren eingesetzt werden:

```
+  -  *  /  %  <<  >>  &  |  ^
```

Beispiel

`x += y` hat denselben Effekt wie `x = y + x`

`k *= 3` hat denselben Effekt wie `k = 3 * k`

4.3.8 Komma-Operator

Der Kommaoperator hat den niedrigsten Rang. Er dient, dazu in einen Ausdruck mehrere Ausdrücke einzubringen. Der Kommaoperator hat die folgende Form:

```
a1 , a2
```

Zuerst wird a1 ausgewertet und danach a2. Der Gesamtausdruck hat den Wert von a2. Der Operator ist linksassoziativ, d. h. ein Ausdruck mit mehreren Kommaoperatoren wird von links nach rechts ausgewertet. Der Gesamtausdruck hat dann den Wert des letzten Teilausdrucks.

Beispiel

```
b = 10;
c = 20;
x = ( ++b , c-- );
```

⇒ b=11 c=19 und x=20

Aufgabe 4

Das folgende Programm ist gegeben. Überlegen Sie zuerst welche Ausgabe auf dem Bildschirm erscheinen wird. Anschließend prüfen Sie dies mit einem eigenen Programm nach!

```
main ( )
{
  int x=61, y=39, z1, z2, z3;
  z1 = x & y >> ( x > y );
  z2 = x & y >> x > y;
  z3 = ++y ^ x-- << 2;
  printf ( "x=%d  y=%d  z1=%d  z2=%d  z3=%d\n",
            x, y, z1, z2, z3 );
}
```

Aufgabe 5

Der größere der beiden Werte von a und b (beide als int vereinbart), soll ausgegeben werden. Bestimmen Sie in printf ( ) den fehlenden Ausdruck (???)!

```
printf ( "%d",  ???  );
```

Aufgabe 6

Wie die vorhergehende Aufgabe jedoch nun mit drei Werten a, b und c, wobei zweckmäßig eine Hilfsvariable eingesetzt wird. Wenn Sie die Aufgabe mit Hilfsvariable gelöst haben, überlegen Sie sich eine Möglichkeit den Ausdruck auch ohne Hilfsvariable zu formulieren.

Aufgabe 7

Schreiben Sie die folgenden Programmstücke 'C'-gerecht in einem Ausdruck!

8.1 (Pascal)

```
y := y * ( a + 1 ) / b;
b := b - 1;
```

8.2 (Pascal)

```
k := k - 1;
x := x * ( k + y ) / z;
z := z + 1;
```

Aufgabe 8

Geben Sie zu jedem nachfolgenden Beispiel die Werte von a, b und c sowie die Werte des Ausdruckes an, unter der Voraussetzung, daß vor der Auswertung eines jeden Ausdruckes gilt:

```
int a = 14, b = 25, c = 1;
```

(a,b und c sind ganzzahlige Variablen mit der Wortbreite 16 Bit)

	Wert von:			
Auszuwertender Ausdruck	a	b	c	Ausdruck
`!c`				
`~c`				
`++c`				
`c++`				
`--a`				
`b--`				
`sizeof(a)`				
`&a`				
`(4 * b + c) / a`				
`(4 * b + c) % a`				
`a << 2`				
`b >> 1`				
`a < b`				
`a > b`				
`c <= c`				
`c >= b`				
`a & b`				
`a | b`				
`a ^ b`				
`a = b = c`				
`a += b`				
`a *= -b`				
`a > c && b > a`				
`c > a || a > b`				
`(a <= b) ? ++a : ++b`				
`c-- ? a++ : b--`				
`a++ , b--`				
`c = a++ , b + c`				

5 Elementare Datentypen

Wir unterscheiden, wie auch in anderen Programmiersprachen, zwischen strukturierten und unstrukturierten Datentypen. Die unstrukturierten Datentypen nennen wir auch elementare Datentypen. In 'C' gibt es die folgenden elementaren Datentypen:

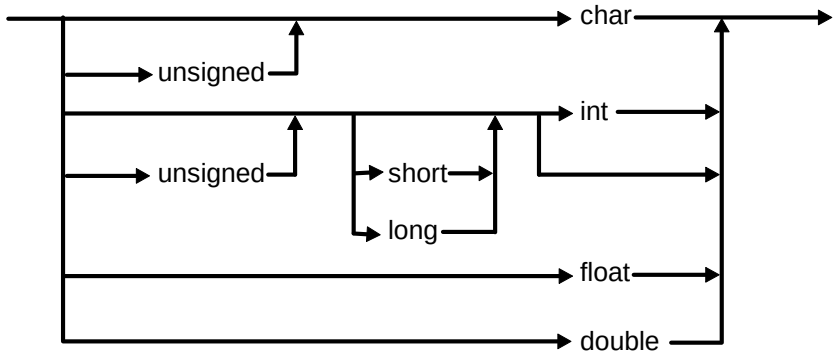
- char → ganzzahlig 8 Bit
- int → ganzzahlig 16 bzw. 32 Bit (implementationsabhängig)
- float → Gleitkommazahl (32 Bit)
- double → Gleitkommazahl (64 Bit)

Wichtig: Alle 4 Datentypen repräsentieren Zahlen. Beachten Sie bitte, daß auch char eine Zahl ist. Die Ausgabefunktionen können jedoch auf Wunsch einen char-Typ auch als Zeichen ausgeben. Dies bedeutet auch, daß Rechenoperationen mit dem char-Typ zugelassen sind.

Modifizierer Die oben aufgeführten Grunddatentypen lassen sich durch folgende Zusätze, die sogenannten Modifizierer, noch genauer spezifizieren:

- short
- long
- unsigned

Dazu beachten Sie bitte das folgende Diagramm:



Zulässige Typangaben wären also z.B. :

```
unsigned short int  oder  long int
```

aber auch diese Angaben sind zulässig:

```
unsigned,   unsigned short oder long
```

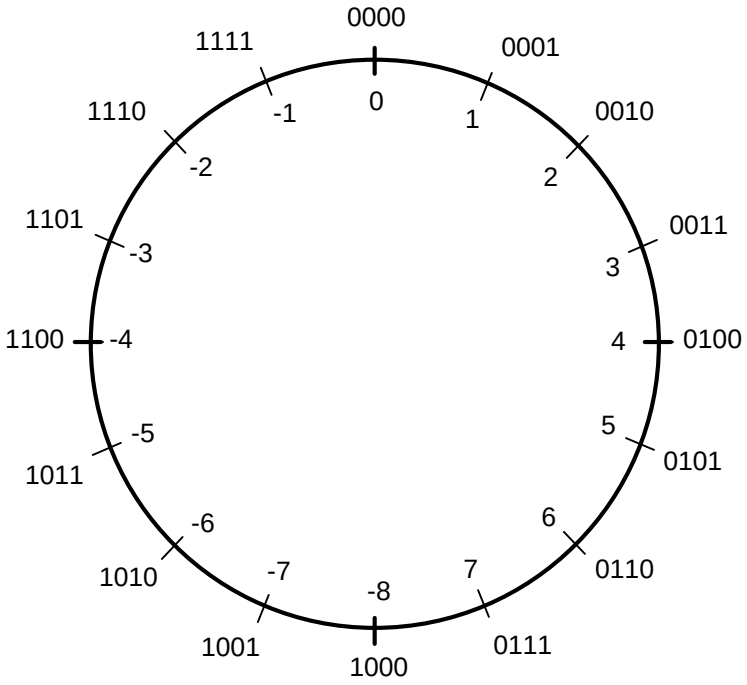
Wenn unsigned, short oder long alleine auftreten, so wird automatisch int als dahinterstehend angenommen.

Wenn kein Typ angegeben wird, wie es gemäß dem Diagramm bei int möglich ist, wird vom Compiler automatisch int als Datentyp angenommen. Diese Möglichkeit ist allerdings nur bei Funktionsdefinitionen zulässig.

Short ist immer das 16-Bit-Format, während long für das 32-Bit-Format steht. Ob int im 16- oder 32-Bit-Format gespeichert wird, hängt von dem verwendeten Compiler ab. Bei den meisten Compilern für DOS- und Windows-Systeme haben die Angaben short, short int und int denselben Effekt, d. h. int wird im 16-Bit-Format gespeichert.

Zweierkomplement

Die ganzzahligen Datentypen werden in der Regel in der Zweierkomplement-Darstellung abgespeichert. Der nachfolgend dargestellte Zahlenkreis soll diese Form der Codierung von ganzen Zahlen in einem 4-Bit-Format erläutern:



Diese Art der Zahlencodierung sieht zunächst ungewöhnlich aus. Man hätte sich in einem ersten Ansatz sicherlich einfachere Lösungen denken können. Ein wesentlicher Vorteil der Zweierkomplementdarstellung jedoch ist, daß die bekannten Rechenoperationen auf diese Zahlen genauso angewandt werden können, wie auf die natürlichen Zahlen.

Umrechnung

Die Umrechnung einer dezimalen Zahl in die duale Darstellung kann nach folgenden Schema erfolgen:

```
6 : 2 = 3 Rest 0
3 : 2 = 1 Rest 1
1 : 2 = 0 Rest 1
-------------------
Duale Darstellung  0  1  1  0
```

Die umzuwandelnde Zahl wird solange durch zwei geteilt, bis sich als Ergebnis eine Null ergibt. Die Reste von unten nach oben abgeschrieben ergeben die duale Darstellung der Zahl. Eventuell müssen führende Nullen ergänzt werden.

Die Umrechnung von positiven nach negativen Zahlen kann erfolgen, indem von rechts beginnend alle Nullen von der umzuwandelnden Zahl abgeschrieben werden. Die erste Eins von rechts bleibt stehen, alle weiteren Stellen müssen invertiert werden.

Beispiel

Positive Zahl	6:	0110
Negative Zahl	-6:	1010

Bei Verwendung der beschriebenen Zahlenformate ergeben sich die folgenden Wertebereiche:

-32768	bis	32767	für das 16-Bit-Format
-2147483648	bis	2147483647	für das 32-Bit-Format

Mit der Angabe unsigned erhält man vorzeichenlose Zahlen mit folgenden Wertebereichen:

0	bis	65535	für das 16-Bit-Format
0	bis	4294967295	für das 32-Bit-Format

IEEE-Format

Gleitkommazahlen werden beim PC in der Regel im IEEE-Format abgespeichert (IEEE Amerikanische Ingenieurvereinigung, vergleichbar mit dem VDE in Deutschland). Dieser Quasi-Standard wird von vielen Gleitkommaprozessoren unterstützt, so auch vom NPX 80x87 bzw. Den modernen 486er- und Pentium-Prozessoren.

Beim IEEE-Format werden die Zahlen wie bei der Anzeige im Taschenrechner in drei Teilen dargestellt.

Beispiel

- 0.178125 E 3 diese Zahl entspricht der Zahl -178.125

Bei dieser Darstellungsform handelt es sich um normierte Gleitkommadarstellung. Die drei Teile aus denen sich die Zahl zusammensetzt heißen:

- Vorzeichen (- oder +),
- Mantisse (0.178125) und
- Exponent (E 3)

Der Exponent bezieht sich bei der Darstellung im Taschenrechner auf die Basis 10. Die Mantisse muß im Beispiel dreimal mit 10 multipliziert werden und man erhält die gewohnte Darstellung der Zahl. Die dreimalige Multiplikation mit 10 entspricht einer Verschiebung des Kommas um drei Stellen nach rechts.

Bei der normierten Darstellung wird die Mantisse immer so hingeschrieben, daß sie mit 0. beginnt. Wenn eine so dargestellte Zahl nun gespeichert werden muß, dann kann man sich das Abspeichern, des 0. und des E sparen. Wenn die Zahlen immer in der normierten Form dargestellt sind, so sind diese beiden Punkte bekannt und somit redundant.

Die Abspeicherung von Gleitkommazahlen im Rechner erfolgt nach demselben Schema, nur daß hierbei die Mantisse dual codiert wird und die Exponenten sich auf die Basis 2 beziehen. Wir wollen uns die Darstellung einer Gleitkommazahl an einem Beispiel im 32 Bit-IEEE-Format verdeutlichen.

Beispiel

Zunächst ist die Mantisse der Zahl zu ermitteln. Vorkomma- und Nachkommateil der Zahl müssen getrennt umgewandelt werden. Das Schema für die Umwandlung des Vorkommateils kennen wir bereits.

178	:	2	=	89	Rest	0
89	:	2	=	44	Rest	1
44	:	2	=	22	Rest	0
22	:	2	=	11	Rest	0
11	:	2	=	5	Rest	1
5	:	2	=	2	Rest	1
2	:	2	=	1	Rest	0
1	:	2	=	0	Rest	1

Damit ergibt sich für den Vorkommateil: 10110010

Die Umwandlung des Nachkommateils erfolgt durch fortlaufende Multiplikation mit 2 solange, bis sich hinter dem Komma eine Null ergibt:

0.125	·	2	=	0.25
0.25	·	2	=	0.5
0.5	·	2	=	1.0

Es wird jeweils nur mit dem Nachkommateil weitergerechnet. Die Nullen und Einsen in den Vorkommateilen der Ergebnisse werden von oben nach unten abgeschrieben und ergeben dann den Nachkommateil der Dualzahl.

Damit ergibt sich als Nachkommateil: .001

Die Zahl 178.125 dual codiert ergibt also:

10110010.001

Nun wird diese Zahl in die normierte Form gebracht und zwar so, daß immer 1. vorne steht. Diese 1. wird später nicht mit abgespeichert (man spricht hierbei vom sogenannten hidden bit).

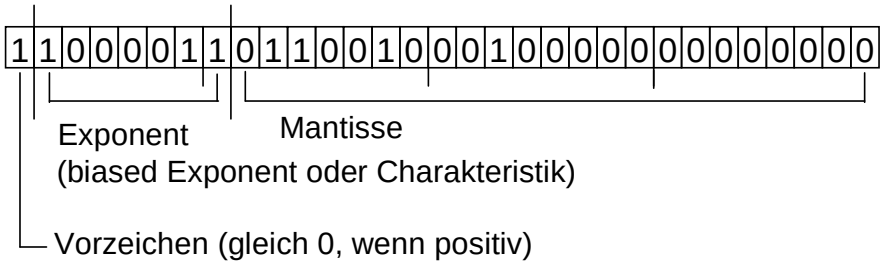
$1.0110010001 \cdot 2^7$

Nun muß der Exponent (7) noch dual codiert werden. Da auch negative Exponenten möglich sind, muß wieder eine Form zur Darstellung negativer Zahlen gefunden werden. Dies erfolgt durch das Bilden eines sogenannten biased exponent. Hierbei handelt es sich um eine Art Nullpunktverschiebung. Es wird das Bias addiert und Null ist nicht mehr Null sondern 127.

Bilden des Exponenten:

7	ist dual codiert	00000111
127	ist dual codiert	01111111
		10000110

Damit ergibt sich nun folgende Darstellung der Zahl:



Bei dem 64-Bit-Gleitkomma-Format (double) besteht die Mantisse aus 52 Bit (plus hidden bit) und der Exponent aus 11 Bit mit einem Bias von 1023.

Damit ergeben sich folgende Wertebereiche für die Gleitkommazahlen:

$1.5 \cdot 10^{-45}$	bis	$3.4 \cdot 10^{38}$	für das 32-Bit-Format
$5.0 \cdot 10^{-324}$	bis	$1.7 \cdot 10^{308}$	für das 64-Bit-Format

Aufgeführt sind jeweils die kleinste und die größte darstellbare Zahl. Die Angaben gelten sowohl für die negativen als auch für die positiven Zahlen. Die Genauigkeit der Zahlen entspricht beim 32-Bit-Format der von 7 bis 8 Dezimalstellen und beim 64-Bit-Format der von 15 bis 16 Dezimalstellen.

Wichtig:

Diese Form der Zahlencodierung hat zur Folge, daß Zahlen unter Umständen nicht hundertprozentig genau abgespeichert werden können. Dadurch können Ungenauigkeiten bei Berechnungen entstehen. Bei Verwendung von Gleitkommazahlen sind insbesondere die Vergleichsoperatoren mit Bedacht anzuwenden. Das nachfolgende Programm sieht auf den ersten Blick korrekt aus und Sie werden annehmen, daß die Schleife nach vier Durchläufen terminiert wird.

```
main( ) {
  float x = 10; int n = 1;
  while( x != 0.001 ) {
    x /= 10;
    printf( "%d", n++ );
  }
}
```

Die Schleife ist vielmehr eine Endlosschleife. Besser ist in solchen Situationen der Einsatz von Größer- oder Kleiner-Operatoren (z. B. x > 0.001).

5.1 Deklarationen

Jede Variable, die benutzt werden soll, muß in 'C' genau wie in Pascal vorher mit einem Typ vereinbart werden. Dies gilt auch für die Funktionen und Funktionsparameter.

Beispiel

'C'

```
int add( int a, int b )
{
  int c;
  c = a + b;
  printf( "c = %8d\n", c );
}
```

Pascal

```
PROCEDURE add( a,b : integer );
  VAR c : integer;
BEGIN
  c := a + b;
  writeln ('c = ', c:8 );
END;
```

void

Diese Funktion kann wie eine Prozedur eingesetzt werden. Sie liefert keinen Wert zurück, dazu wäre eine zusätzliche Anweisung notwendig. In diesem Fall ist die Typangabe also überflüssig und in der Tat kann man eine Typangabe bei Funktionen auch weglassen. Es wird dann allerdings automatisch int als Typ für die Funktion angenommen. Möchte man, wie hier, eine Funktion wie eine Prozedur einsetzen, so empfiehlt es sich explizit anzugeben, daß kein Funktionswert erwünscht ist. Dies geschieht mit dem leeren Typ void (leer, unwirksam). Es wäre also im obigen Beispiel zu empfehlen, die Funktion add wie folgt zu deklarieren:

```
void add ( int a, int b )
```

Zum Aufruf dieser Unterprogramme ist in beiden Programmiersprachen dieselbe Anweisung notwendig z. B. :

```
add ( 23, 12 )        oder    add ( r, b+1 )
```

Initialisierung

Darüber hinaus können in 'C' die Variablen mit der Deklaration (Vereinbarung) auch initialisiert werden (mit einem Wert vorbelegt werden). Dies geschieht durch eine entsprechende Zuweisung bei der Deklaration.

Beispiel

```
int x1=32, x2=34, x3=0;
float f1=.5;
char s1=32, s2='\n', s3='A';
```

5.2 Mixed mode

In 'C' dürfen innerhalb eines Ausdruckes die Datentypen gemischt auftreten. Es wird dann bei der Compilierung eine automatische Typanpassung vorgenommen. Das nachfolgende Schema zeigt, die Möglichkeiten der Typumwandlung. In einem Ausdruck mit zwei unterschiedlichen Typen wird die Operation immer mit dem Typ vorgenommen, der in diesem Schema weiter rechts steht.

char ⇒

short ⇒ int ⇒ long ⇒

unsigned ⇒ unsigned long ⇒

float ⇒ double

Beispiel

```
char      c;
unsigned  u;
float     f;
int       i;

i  =  f  *  ( c  +  u );
               |______|
               unsigned
         |_________________|
               double
|_________________________|
            int
```

Beachten Sie bitte, daß hier double angedeutet ist, obwohl f den Typ float hat. In 'C' werden gemäß Kerninghan und Ritchie alle Gleitkomma-berechnungen im double-Format ausgeführt, um Rundungsfehler während der Berechnung zu vermindern. Bei der Zuweisung wird das Ergebnis automatisch wieder an die aufnehmende Variable angepasst.

Im Gegensatz zu Pascal ist ein solcher Ausdruck erlaubt und man erhält beim Compilieren keine Fehlermeldung. Allerdings sind solche Ausdrücke mit Vorsicht zu genießen, da es durch die internen Umwandlungen zu Rundungsfehlern und Bereichsüberschreitungen kommen kann. Bei modernen ‘C++’-Compilern, werden diese in Abhängigkeit von der Warnstufe an solchen Stellen auch Hinweismeldungen ausgeben.

Auch bei Funktionsaufrufen wird eine automatische Typ-Umwandlung vorgenommen. Die Ausnutzung der automatischen Typumwandlung führt in vielen Fällen zu schwer nachvollziehbaren Programmen. Deshalb sollte man von der Möglichkeit der Steuerung von Typumwandlungen Gebrauch machen. Damit ist es auch möglich die oben angesprochenen Hinweis-meldungen zu vermeiden.

Mit Hilfe des Cast-Operator `(typ)` kann man die Umwandlung steuern:

Beispiel

```
i = (int) f * ( c + (int) u );
```

In diesem Beispiel finden alle Operationen im int-Format statt.

Wichtig

Manchmal sind Operationen aufgrund von Rundungsfehlern durch die automatische Typanpassung nicht kommuntativ. Beachten Sie dazu das folgende Beispiel:

```
int i, a = 33, b = 5, c = 10;

i = a / b * c
```

Der obere Ausdruck ergibt für i = 60, während der nachfolgende Ausdruck

```
i = a * c / b
```

für i = 66 ergibt.

5.3 Weitere elementare Datentypen

Mit der Spracherweiterung von ‘C’ sind 1976 zwei weitere elementare Datentypen definiert worden. Dies sind:

- void
- enum

Den Typ void haben wir bereits im Zusammenhang mit einem Beispiel kennengelernt. Es ist dies ein leerer Datentyp, der bei Funktionen, die wie Prozeduren gehandhabt werden sollen, zum Einsatz kommt. Sie vermeiden damit entsprechende Hinweismeldung, wegen fehlender Festlegung des Funktionswertes oder Nichtverwendung des Funktionswertes. Der Typ enum entspricht dem Aufzählungstyp in Pascal:

Beispiel

'C'

```
enum name { fritz, susi, otto, erna };
enum name person1, person2;
```

Pascal

```
TYPE name = ( fritz, susi, otto, erna );
VAR  person1, person2 : name;
```

Man könnte in 'C' auch schreiben:

```
enum name { fritz, susi, otto, erna } person1;
```

Wobei name hier auch weggelassen werden kann. Eine Vereinbarung eines Namens in einer solchen Zeile gibt nur Sinn, wenn der vereinbarte Typ an anderen Stelle nochmals gebraucht wird.

Die Werte des enum-Types sind in Wirklichkeit versteckte int-Werte. Sie entsprechen den Ordnungszahlen in Pascal, also 0,1,2,3 (bei 4 Elementen).

Man kann mit den enum-Werten rechnen!

Beispiel

```
printf ("%d", erna * otto );
```

Die interne Berechung mit erna = 3 und otto = 2 ergibt 3*2 = 6. Es wird also die Zahl 6 auf dem Bildschirm ausgegeben.

Es ist außerdem möglich den Konstanten andere als die Standardwerte zuzuweisen.

Beispiel

```
enum name { fritz=20, susi=19, erna, otto=42 };
```

$\Rightarrow$ fritz = 20, susi = 19, erna = 2 (Standardwert), otto = 42

Der Aufzählungstyp ist also eine andere Möglichkeit Konstanten zu definieren. Der Vorteil des enum-Typs ist in der Gruppierung verwandter Namen zu suchen und in der kompakteren Schreibweise als eine Reihe von define-Anweisungen.

Variablen und Konstanten des enum-Typs werden wie entsprechende int-Größen behandelt.

Aufgabe 9 Wie kann man, ohne eine Konstante zu verwenden, x = 0 realisieren?

Betrachten Sie dazu die Größe x bitweise!

Aufgabe 10 Es soll ein Programm geschrieben werden, das eine 1 ausgibt, wenn die int-Variable j eine einem Schaltjahr entsprechende Zahl enthält, andernfalls soll eine 0 ausgegeben werden.

Benutzen Sie die logischen Operatoren!

Aufgabe 11 Folgende Deklaration wird vorausgesetzt:

```
int n;
float r;
```

Welchen Datentyp hat dann der folgende Ausdruck?

```
n > 0 ? n : r
```

Überlegen Sie sich, wie Sie Ihre Überlegungen mit Hilfe eines kleinen Programmes überprüfen können!

Aufgabe 12 Vertauschen Sie die Werte zweier Variablen a und b innerhalb eines Ausdrucks (die Verwendung einer Hilfsvariablen ist zweckmäßig, aber nicht nötig). Welchen Wert hat ihr Ausdruck?

Aufgabe 13 Schreiben Sie ein 'C'-Programm, das anzeigt, ob beim Rechtsshift von negativen Zahlen eine 1 oder eine 0 nachgeschoben wird!

Erwünschte Ausgabe:

```
-->1    bzw.    -->0
```

6 Anweisungen

Anweisungstypen

Die Anweisungen steuern den Programmablauf. In 'C' unterscheiden wir folgende Anweisungstypen.

- Leere Anweisung
- Verbundanweisung
- Zuweisung
- if-Anweisung
- switch-Anweisung
- break-Anweisung
- while-Anweisung
- do .. while-Anweisung
- for-Anweisung
- continue-Anweisung
- return-Anweisung
- goto-Anweisung

Die einzelnen Anweisungstypen werden in den folgenden Kapiteln besprochen. Grundsätzlich kann man sagen: Ein Programm besteht aus einer Folge von Anweisungen.

6.1 Leere Anweisung

Die leere Anweisung ist in 'C' das Semikolon. Hinter jeder Anweisung muß eine in 'C' ein Semikolon folgen. Einzige Ausnahme ist die Verbundanweisung. Hinter der Verbundanweisung ist kein Semikolon verlangt.

Vergleich

Pascal	C
; ;	;
leere Anweisung zwischen den Semikolons	leere Anweisung ist das Semikolon

Bei der leeren Anweisung findet keine Aktion statt. Es wird kein Code und keine Rechenzeit benötigt.

Die leere Anweisung kann eingesetzt werden, wenn aus syntaktischen Gründen zwar eine Anweisung erforderlich ist, aber keine Aktion ausgeführt werden muß.

6.2 Verbundanweisung

Die Verbundanweisung dient dazu mehrere Anweisungen eines beliebigen Typs zu einer Anweisung zusammenzufassen. Sie wird eingesetzt, wenn aus syntaktischen Gründen nur eine Anweisung zulässig ist, aber mehrere Anweisungen ausgeführt werden müssen.

Syntax

```
{
 lokale Vereinbarung;
 Anweisung1;
 ...
 ...
 AnweisungN;
}
```

Beispiel

```
{            /* entspricht BEGIN in Pascal */
 int i,k;    /* lokale Vereinbarung von i, k */
 i = 31;     /* 1. Anweisung */
 k = ++i;    /* 2. Anweisung */
 x = k * i;  /* x ist global definiert */
}            /* entspricht END in Pascal */
```

Verbundanweisungen können geschachtelt werden. Man beachte, daß auch vor der abschließenden Klammer ein Semikolon stehen muß. Anders als in Pascal, wo vor dem abschließenden END das Semikolon entfallen kann.

Außerdem können anders als in Pascal lokale Vereinbarungen im Block der Verbundanweisung erfolgen. Sie können dadurch bei mehreren ineinander geschachtelten Blöcken auch blocklokale Variablen definieren.

Beispiel

```
{             /* Block1 */
 int a,b,c;
 {            /* Block2 */
  int a;
  {           /* Block3 */
   int b;
  }
 }
}
```

Die in Block1 definierte Variable a ist nur in Block1 bekannt, da sie in Block2 und Block3 durch die in Block2 definierte Variable a überlagert wird.

Die in Block1 definierte Variable b ist in Block1 und Block2 bekannt, nicht aber in Block3.

Die in Block1 definierte Variable c hat in allen drei Blöcken Gültigkeit.

Die in Block2 definierte Variable a gilt in Block2 und Block3, die in Block3 definierte Variable b nur in Block3.

6.3 Zuweisung

Die Zuweisung dient dazu einer Variablen (einem L-Value) den Wert eines Ausdruckes zuzuweisen. Die Zuweisungsoperatoren sind uns bereits bekannt.

`=` bzw. `op=`

Ein Ausdruck mit einem Zuweisungsoperator wird dadurch zur Zuweisung, daß er mit einem Semikolon abgeschlossen wird.

Syntax

```
Zuweisungs-Ausdruck;
```

Beispiel

```
x =  24;
x += 24;
x =  i + k;
```

In diesen Beispielen wird der Variablen x ein neuer Wert zugewiesen. Auf der linken Seite können aber auch Ausdrücke stehen, die einen L-Value repräsentieren.

Beispiel

```
*p = 24;
```

Wenn p ein Zeiger auf die Variable x ist, dann hat diese Zuweisung denselben Effekt wie das erste Beispiel.

Als eine besondere Art der Zuweisung kann man die Verwendung der Inkrement- bzw. Dekrement-Operatoren interpretieren.

Beispiel

```
i++;
++i;
```

Hierbei ist die Post- bzw. Prefix-Notation ohne Bedeutung.

6.4 if-Anweisung

Die if-Anweisung ist eine von zwei Verzweigungsanweisungen in C. Sie ermöglicht es in Abhängigkeit vom steuernden Ausdruck entweder die eine oder die andere Anweisung ausführen zu lassen.

Syntax

```
if (ausdruck)
  anweisung1;
else
  anweisung2;
```

oder

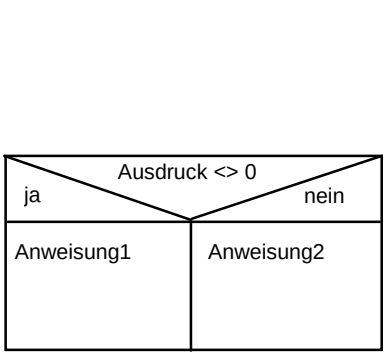
```
if (ausdruck)
  anweisung;
```

Wenn ausdruck ungleich 0 ist (entspricht true), dann wird anweisung1 bzw. anweisung ausgeführt. Wenn ausdruck gleich 0 ist (entspricht false), dann wird im ersten Fall anweisung2 und im zweiten Fall keine Anweisung ausgeführt.

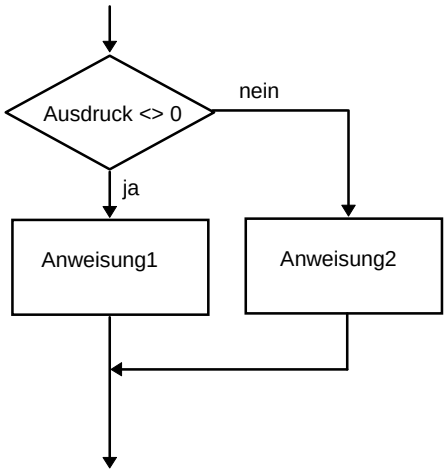
Im Gegensatz zu Pascal sind folgende Punkte zu beachten:

- Der Ausdruck hinter dem if muß in Klammern stehen.
- Vor dem else muß ein Semikolon stehen, es sei denn, es wird dieVerbundanweisung eingesetzt.
- Es gibt kein then.

Die if-Anweisung kann in Diagrammen wie folgt dargestellt werden:



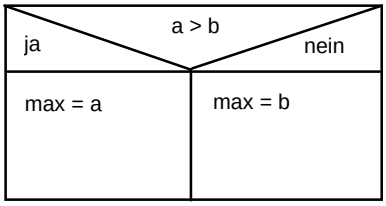
Struktogramm



Programmablaufplan (PAP)

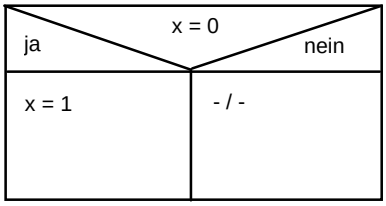
Beispiel

Die nachfolgenden Beispiele zeigen jeweils ein Struktogramm und die entsprechende C-Formulierung.




```
if ( a>b )
  max = a;
else
  max = b;
```

In diesem Beispiel wird der Variablen max der jeweils größere Wert der beiden Variablen a und b zugewiesen. Dies ließe sich in 'C' eleganter mit dem Bedingungsoperator lösen.




```
if ( x==0 )
  x = 1;
```

In diesem Beispiel wird auf Gleichheit mit einer Konstanten geprüft. Man beachte das hier der Vergleichsoperator == eingesetzt wird. Die Formulierung (x=0) wäre in 'C' kein Fehler, allerdings würde dann die Anweisung nie ausgeführt, da die Bedingung immer 0, d. h. false, wäre.

Insbesondere Prüfungen auf Gleichheit oder Ungleichheit mit Null lassen sich in 'C' dadurch eleganter formulieren, daß man den Wert der Variablen direkt interpretiert.

Beispiel

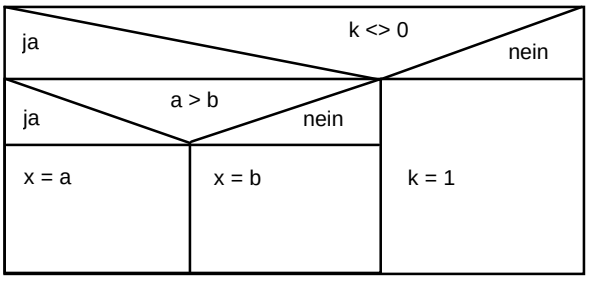
`if ( k )` entspricht in der Wirkung `if ( k != 0 )`

`if ( !k )` entspricht in der Wirkung `if ( k == 0 )`

In 'C' sind auch verschachtelte if-Anweisungen möglich.

Beispiel

```
if ( k )
  if ( a>b )
    x = a;
  else
    x = b;
else
  k = 1;
```



Häufig findet man bei mehrfach verschachtelten Anweisungen auch folgende Formulierung:

Beispiel

```
if ( a )
  x = 1;
else if ( b )
  x = 2;
else if ( c )
  x = 3;
else if ( d )
  x = 4;
```

6.5 switch-Anweisung

Die switch-Anweisung ist eine Verzweigungsanweisung für Mehrfachverzweigungen. In Abhängigkeit vom steuernden Ausdruck werden jeweils andere Anweisungen ausgeführt.

Syntax

```
switch (ausdruck)
{
  case konstante1 : anweisung(en)1;
  case konstante2 : anweisung(en)2;
  case konstante3 : anweisung(en)3;
  .
  .
  case konstanteN : anweisung(en)N;
  default : anweisung(en);
}
```

Diese Anweisung ähnelt der CASE-Anweisung in Pascal oder dem ON GOTO in Basic. Es besteht allerdings ein wichtiger Unterschied zu der entsprechenden Pascal-Anweisung. Es werden nähmlich alle nach der entsprechenden Konstante folgenden Anweisungen ausgeführt, also auch die Anweisungen, die hinter den nachfolgenden Konstanten stehen. case ist quasi eine Einsprungadresse. Der default-Teil ist optional, er kann also weggelassen werden. Gibt es keine Übereinstimmung mit einer Konstanten, so wird nichts bzw. der default-Teil ausgeführt.

Beispiel

```
int i=4,k=3;
switch ( k++ )
{
  case 1  : i++;
  case 2  : i--;
⇒ case 3  : i *= 2;
  case 4  : i += 3;
  default : i -= 7;
}
```

Es gelangen in diesem Fall die folgenden Anweisungen zur Ausführung:

```
k++;
i *= 2;
i += 3;
i -= 7;
```

Falls mit mehreren Konstanten dieselben Anweisungen zur Ausführung gebracht werden sollen, so muß für jede Konstante case aufgeführt werden.

Beispiel

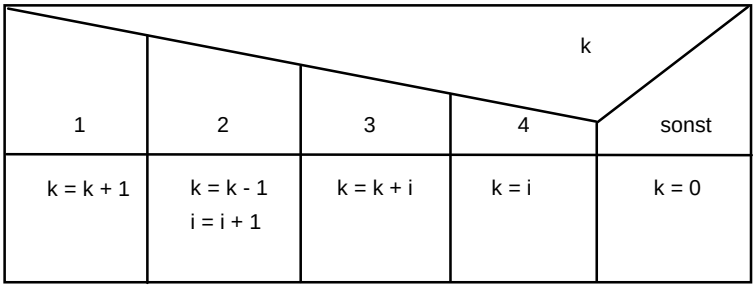
```
switch ( k+i )
{
  case 4 :
  case 5 :
  case 9 : anweisung1
  case 8 : anweisung2
}
```

Wünschenswert wäre nun noch eine Struktur, die es ermöglicht, genau wie in Pascal nur eine Anweisung bzw. einen Block für jede Konstante ausführen zu lassen. Hierfür können Sie die break-Anweisung einsetzen, die quasi einen Sprung an das Ende eines Blockes ermöglicht.

Beispiel

```
switch ( k )
{
  case 1  : k++; break;
  case 2  : k--; i++; break;
  case 3  : k = k + i; break;
  case 4  : k = i; break;
  default : k = 0;
}
```

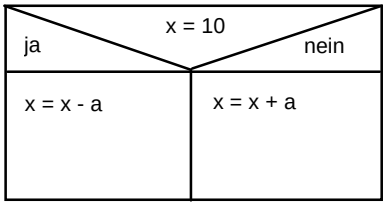
Beachten Sie bitte, daß bei der switch-Anweisung nach dem case keine Verbundanweisung folgen muß, sondern eine beliebig lange Folge von Anweisungen folgen kann.



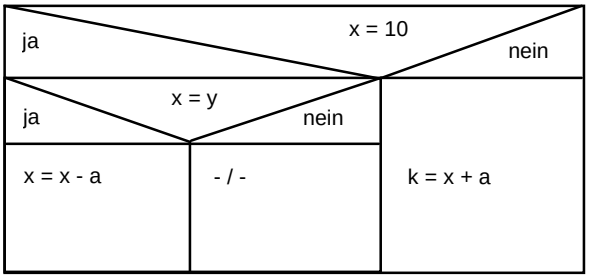
Aufgabe 14

Setzen Sie die beiden nachfolgenden Struktogramme in 'C'-Anweisungen um. Benutzen Sie dazu alle vorgestellten entscheidungsabhängigen Anweisungstypen (if, switch und Zuweisung mit Bedingungsoperator) !

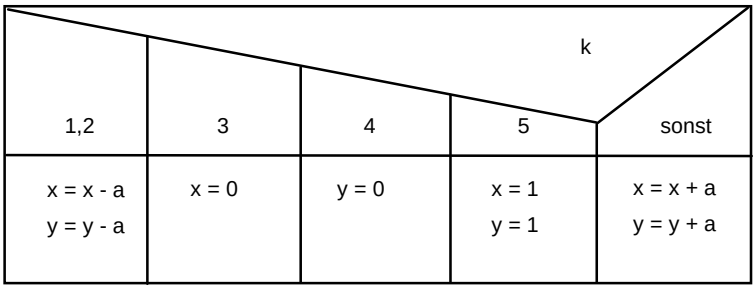
14.1



14.2



14.3



6.6 while-Anweisung

Die while-Anweisung ist die erste von drei Wiederholungsanweisungen in C, die wir kennenlernen wollen. Bei der while-Anweisung handelt es sich um eine sogenannte abweisende Schleife. Die erste Auswertung des steuernden Ausdrucks wird vor dem ersten Schleifendurchlauf durchgeführt.

Syntax

```
while (ausdruck)
  anweisung;
```

Falls der Ausdruck ungleich 0 ist (entspricht true), dann wird die Anweisung ausgeführt. Die Anweisung kann jede beliebige andere Anweisung, speziell natürlich auch eine Verbundanweisung sein. Nach der Ausführung erfolgt dann eine erneute Überprüfung des Ausdruckes. Es ist darauf zu achten, daß der Ausdruck irgendwann den Wert 0 erreicht, sonst entsteht eine Endlosschleife.

Beispiel

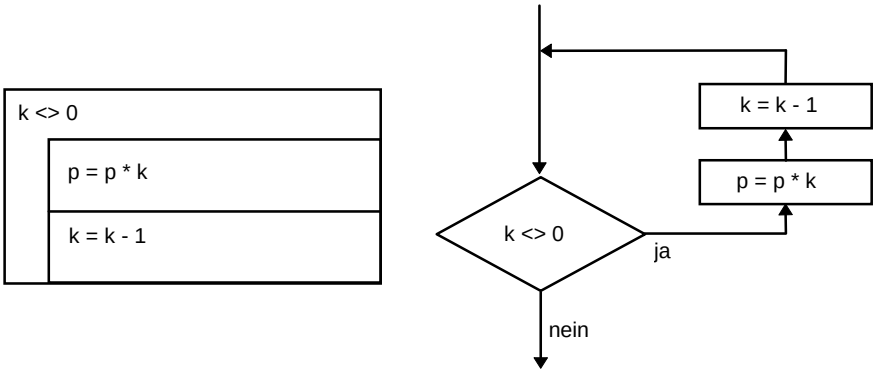
Die Wirkung der folgenden drei Beispiele sollten Sie sich anhand von Programmablaufplänen und eines Handtests nochmals verdeutlichen.

Folgende Definitionen werden für alle Beispiele vorausgesetzt:

```
long p=1;
int  k=3;
```

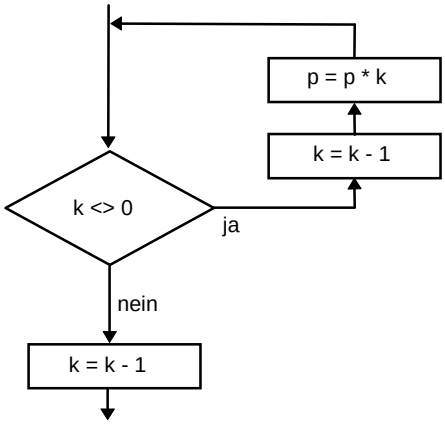
Beispiel a

```
while ( k )
  p *= k--;
```



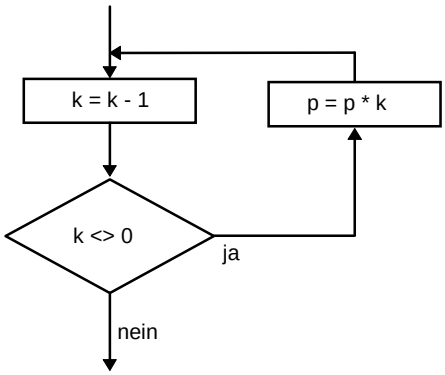
Beispiel b

```
while ( k-- )
  p *= k;
```



Beispiel c

```
while ( --k )
  p *= k;
```



Die drei C-Formulierungen sehen sich auf den ersten Blick ziemlich ähnlich und man möchte annehmen, daß alle drei Formulierungen dieselben Ergebnisse liefern. Wie uns die Ergebnisse nach einem Handtest zeigen, ist dies jedoch nicht der Fall.

Beispiel a	⇒	k = 0	p = 6
Beispiel b	⇒	k = -1	p = 0
Beispiel c	⇒	k = 0	p = 2

Mit diesen Beispielen sollten Sie auf die Gefahren aufmerksam gemacht werden, die sich in der kompakten Schreibweise von C verbergen können. Sie sollten sich gerade bei der Programmierung von Schleifen immer mit Handtests von der korrekten Funktion Ihrer Codierung überzeugen.

break

Die im Zusammenhang mit der switch-Anweisung vorgestellte break-Anweisung kann auch mit der while-Anweisung eingesetzt werden. Durch den Einsatz der break-Anweisung kann das Ende des Blockes vorzeitig erreicht werden, d.h. die Schleife vorzeitig verlassen werden.

Beispiel

```
int k=10;
while ( 1 )
{
  printf ("%d\n",k);
  if (k-- == 5) break;
}
```

Bei diesem Beispiel wird die Terminierung der Schleife nur durch die break-Anweisung erreicht. Anweisungen, die hinter der break-Anweisung stehen, werden nicht mehr ausgeführt.

6.7 do .. while-Anweisung

Die do .. while-Anweisung ist die zweite der vorgestellten Wiederholungsanweisungen in C. Sie unterscheidet sich von der while-Anweisung dadurch, daß die Anweisung im Schleifenkörper vor dem ersten Test des steuernden Ausdrucks einmal ausgeführt wird.

Syntax

```
do
  anweisung;
while (ausdruck);
```

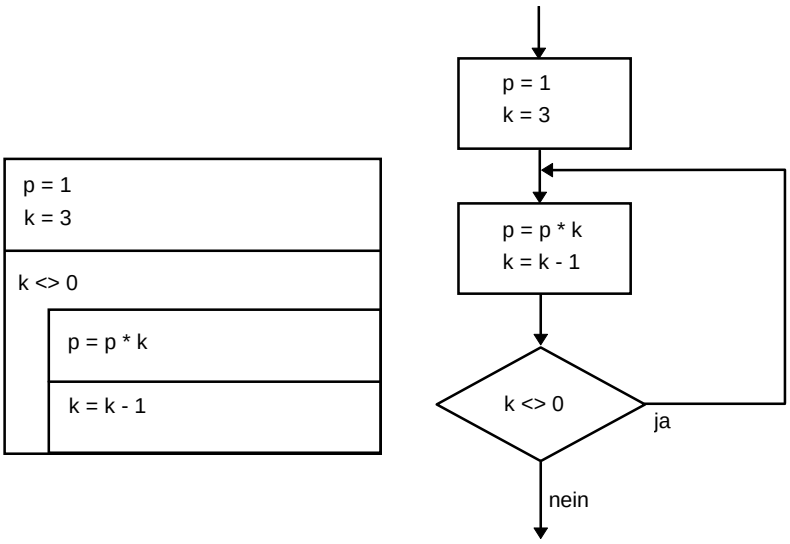
Bei der do .. while-Anweisung wird zuerst die Anweisung ausgeführt und dann der steuernde Ausdruck überprüft. Ist der Ausdruck ungleich 0, so wird die Anweisung wiederholt, ist der Ausdruck gleich 0, so wird die do .. while-Anweisung verlassen.

Im Prinzip entspricht das der REPEAT .. UNTIL-Anweisung in Pascal mit einem wichtigen Unterschied. Der Ausdruck muß nämlich false sein, damit die Schleife verlassen wird.

Beispiel

Die Wirkung des folgenden Beispiels sollte man sich anhand eines Programmablaufplanes und eines Handtests nochmals verdeutlichen.

```
long p=1;
int  k=10;

do
  p *= k--;
while ( k );
```



6.8 for-Anweisung

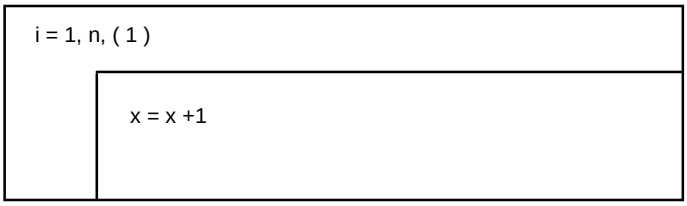
Die for-Anweisung ist ebenfalls eine Wiederholungsanweisung, die bevorzugt eingesetzt wird, wenn die Zahl der Schleifendurchläufe im voraus bekannt bzw. berechenbar ist.

Syntax

```
for ( ausdruck1; ausdruck2; ausdruck3 )
  anweisung;
```

Die for-Anweisung entspricht einer wie folgt angeordneten while-Anweisung:

```
ausdruck1;
while ( ausdruck2 )
{
  anweisung;
  ausdruck3;
}
```



Struktogramm der for-Anweisung zu den nachfolgenden Beispielen.

Beispiel

Pascal

```
FOR i :=  1 TO n DO
  x := x + 1
```

Mögliche Formulierungen in C:

```
for ( i=1; i<=n; i++ ) x++;

for ( i=1; x++,i<=n; i++ );

i=1;
for ( ; x++,i<=n; i++ );
```

Die einzelnen Ausdrücke innerhalb der Klammer können auch leer sein.

Im Extremfall können sogar alle Ausdrücke leer sein. Die for-Anweisung wäre dann eine Endlosschleife, die durch eine break-Anweisung terminiert werden könnte. Die Semikolons müssen jedoch in jedem Fall angegeben werden.

Beispiel

```
for ( ; ; )
{
  ..... ;
}
```

Interessant ist auch die Möglichkeit mehrere Laufvariablen in einer Schleife zu kombinieren.

Beispiel

```
for ( i=10,k=1; i; k++,i-- )
{
  ..... ;
}
```

Außerdem beachten Sie bitte, daß Sie durch den dritten Ausdruck die Schrittweiten der Laufvariablen bestimmen können.

6.9 break-Anweisung

Mit der break-Anweisung können die switch-, while-, do .. while- oder for-Anweisungen vorzeitig beendet werden. Es wird in der Abarbeitung mit der nach der switch- bzw. Wiederholungsanweisung nachfolgenden Anweisung fortgefahren.

Syntax

```
break;
```

Die break-Anweisung entspricht einem unbedingten Sprung hinter das Ende eines Blockes. Sie ist insoweit mit einer goto-Anweisung zu vergleichen. Ihr Einsatz sollte auf die Situationen begrenzt bleiben, die dem Prinzip der strukturierten Programmierung nicht widersprechen, z. B. innerhalb der switch-Anweisung.

6.10 continue-Anweisung

Die continue-Anweisung findet in den Wiederholungsanweisungen while, do..while und for Verwendung. Es wird damit eine unmittelbare Überprüfung der Abbruchbedingung veranlaßt. Bei der for-Anweisung wird vor der Überprüfung zuerst noch der ausdruck3 ausgewertet.

Syntax

```
continue;
```

Die break-Anweisung entspricht einem unbedingten Sprung vor das Ende eines Blockes. Die in einer Schleife nach der continue-Anweisung stehenden Anweisungen werden nicht mehr ausgeführt.

Beispiel

```
int k = 100, sum = 0;
while ( k-- )
{
  if ( k % 3 ) continue;
  sum += k;
}
```

In diesem Beispiel wird die Summe aller durch 3 teilbaren Zahlen, die kleiner als 100 sind berechnet.

Wenn diese Anweisung eingesetzt wird, entstehen oft Konstruktionen, die nicht der strukturierten Programmierung entsprechen. Daher sollte die continue-Anweisung mit Bedacht eingesetzt werden.

6.11 return-Anweisung

Die return-Anweisung dient zum Beenden einer Funktion (auch der Funktion main). Mit Hilfe des Ausdruckes kann der Funktionswert bestimmt werden.

Syntax

```
return;    oder      return ausdruck;
```

Beispiel

```
int eins ()
{
  return 1;
}
```

In diesem Beispiel wird durch den Aufruf der return-Anweisung der Funktion der Funktionswert zugewiesen.

```
void nix ()
{
  if ( ausdruck )
    return;
  else
    anweisung1;
  anweisung2
}
```

In diesem Beispiel wird durch den Einsatz der return-Anweisung erreicht, daß für den Fall, daß ausdruck ungleich 0 ist, die Funktion nix( ) vorzeitig beendet wird. Die hinter der if-Anweisung stehende Anweisung anweisung2 wird nicht mehr ausgeführt. Die return-Anweisung entspricht einem goto zum Ende des Funktionsblocks.

Auch mit dieser Anweisung sind Programmkonstruktionen möglich, die dem Prinzip der strukturierten Programmierung widersprechen. Die return-Anweisung sollte deshalb nur einmal und zwar am Ende eines Funktionsblockes aufgerufen werden.

6.12 goto-Anweisung

Die goto-Anweisung ermöglicht einen direkten Sprung an eine beliebige Stelle innerhalb eines Funktionsblocks.

Syntax

```
goto marke;
...
...
marke: anweisung(en);
```

Durch die Anwendung der goto-Anweisung wird das Programm als nächstes die Anweisung ausführen, die unmittelbar hinter der zugehörigen Marke steht.

Die goto-Anweisung ist vor allen den Basic-Programmierern bekannt. Von ihrer Verwendung wird bei den strukturierten Programmiersprachen abgeraten, weil sie leicht zu sehr unübersichtlichen Codierungen führt (Spaghetti-Code).

Da in 'C' alle für die strukturierte Programmierung nötigen Anweisungstypen vorhanden sind, sollte man auf die Verwendung dieser Anweisung ganz verzichten. Eine sinnvolle Anwendungsmöglichkeit ergibt sich allerdings bei sehr stark ineinander verschachtelten Wiederholungsstrukturen. In diesen Fällen kann mit der goto-Anweisung ein Verlassen aller Schleifen durch eine einzige Anweisung erreicht werden.

Aufgabe 15

Schreiben Sie ein Programm, das die n-te Potenz einer Zahl x berechnet. Für den Fall, daß das Ergebnis nicht definiert ist, soll eine Fehlermeldung ausgegeben werden.

n soll eine ganze Zahl und x eine reelle Zahl sein.

Analysieren Sie zunächst die Aufgabenstellung und fertigen Sie zuerst ein Struktogramm des Lösungsweges an!

Aufgabe 16

Entwickeln Sie ein Programm, das die natürlichen Zahlen > 0 solange addiert, bis die Summe aller Zahlen 5050 oder größer geworden ist. Anschließend sollen die Summe der Zahlen und die Anzahl der addierten Zahlen ausgegeben werden!

Aufgabe 17

ASCII-Tabelle ausgeben

Schreiben Sie ein Programm, mit dem Sie die ASCII-Zeichen mit einem Code >=32 und <127 in 8 Spalten bzw. 7 Spalten auf dem Bildschirm ausgeben. Es soll jedes Zeichen mit dem dazugehörigen ASCII-Code in HEX- bzw. Dezimaldarstellung ausgegeben werden.

Die zugehörigen Formatelemente sind:

%c für Zeichen
%X für Hex-Zahlen
%d für Dezimalzahlen

7 Standard-Ein- und Ausgabe

Ein- und Ausgabeanweisungen sind nicht Bestandteil der Sprache 'C'. Vielmehr gibt es Standardbibliotheken, die Funktionen für die Ein- und Ausgabe zur Verfügung stellen.

Von der Vielzahl der zur Verfügung stehenden Funktionen wollen wir zunächst die folgenden betrachten:

Ausgabe

`printf (...);`	formatierte Ausgabe
`putchar (...);`	Ausgabe eines Zeichens
`puts (...);`	Ausgabe einer Zeichenkette

Eingabe

`scanf (...);`	formatierte Eingabe
`getchar (...);`	Einlesen eines Zeichens
`gets (...);`	Einlesen einer Zeichenkette

Alle diese Funktionen arbeiten mit der Standard-Ein- und Ausgabedatei. Ihre Funktionsprototypen sind in der Header-Datei stdio.h enthalten. Diese Datei muß, wenn die Funktionen benutzt werden, mit der Preprozessoranweisung include eingebunden werden. Bei MS-DOS hat die Standard-Ein- und Ausgabedatei den Namen CON (Console, also Tastatur und Bildschirm). An dieser Stelle sei darauf hingewiesen, daß mit Hilfe der Zeichen <, > und | auf Betriebssystemebene die Möglichkeit der Ein- und Ausgabeumleitung besteht. Dies gilt sowohl für MS-DOS- als auch für UNIX-Systeme.

Ausgabeumleitung

Wenn Sie die Ausgabe eines Programmes in eine andere Datei umleiten möchten, so ist dafür folgende Eingabe in der Kommandozeile des Betriebssystems nötig:

```
progname >datname
```

Damit wird die Ausgabe des Programms progname in die Datei datname umgelenkt.

Auf diese Weise bietet sich dem Benutzer eine einfache Möglichkeit die Ausgaben eines Progammes nachträglich mit einem Editor zu bearbeiten oder auf ein anderes Ausgabegerät zu lenken (z. B. Drucker, PRN).

Eingabeumleitung

Wenn Sie die Eingabe für ein Programm aus einer anderen Datei holen möchten, so ist dafür die folgende Eingabezeile nötig:

```
progname <datname
```

Damit werden alle vom Programm angeforderten Eingaben aus der angegebenen Datei geholt. Auf diese Weise können Sie Programme automatisiert ablaufen lassen, z. B. beim Einsatz von Programmen in Stapeldateien.

Beide Möglichkeiten der Umleitung lassen sich auch in einer Kommandozeile angeben.

Piping

Eine weitere Möglichkeit der Ein- und Ausgabeumleitung ist das sogenannte Piping. Damit wird ermöglicht, die Ausgabe eines Programmes als Eingabe an ein anderes Programm weiterzugeben.

```
progname | sort
```

In diesem Beispiel wird die Ausgabe des Programmes progname als Eingabe an das Programm sort weitergereicht. SORT ist ein Kommando des MS-DOS und ermöglicht das zeilenweise Sortieren der Eingabedatei. Solche Programme wie SORT nennt man Filter.

Filter

Filter kann man durch das folgende grundsätzliche Verhalten beschreiben: Sie bearbeiten eine Eingabedatei, die durch die Bearbeitung nicht verändert wird und erzeugen als Ergebnis eine neue Datei, die Ausgabedatei, die aus der verarbeiteten Eingabedatei entstanden ist.

Solche Filterprogramme lassen sich in 'C' sehr einfach mit den die Standard-Ein- und Ausgabe benutzenden Funktionen realisieren.

Im Gegensatz zu den oben aufgeführten Funktionen stehen Funktionen, die direkt in den Bildschirmspeicher schreiben (diese Art der Ausgabe ist zwar schneller, aber sie läßt sich nicht mehr umlenken). Außerdem gibt es Funktionen, die es ermöglichen, die Ein- und Ausgabedateien im Quelltext des 'C'-Programmes festzulegen, wie z.B. die speziellen Funktionen zur Behandlung der Dateien auf einem Plattenspeicher.

7.1 printf( )

Diese Funktion bietet die Möglichkeit der formatierten Ausgabe von Daten.

Die allgemeine Form der Funktion sieht wie folgt aus:

```
printf( Formatstring [,Ausdrücke] );
```

Formatstring

Der Formatstring ist dabei zwingend vorgeschrieben. Es muß also mindestens der Formatstring in der aktuellen Parameterliste angegeben werden. Ein Beispiel für die Anwendung der printf( )-Funktion mit dem Formatstring ist unser erstes Beispielprogramm:

```
printf ( "hello, world.\n" );
```

In diesem Beispiel ist der Formatstring eine Zeichenkettenkonstante, die auch Steuerzeichen (siehe auch Kapitel 3.4.3) enthalten darf. Der Formatstring darf auch eine Zeichenkettenvariable sein.

Formatelemente

Wenn mit der printf-Funktion außer dem Formatstring auch noch Werte formatiert ausgegeben werden sollen, dann muß für jeden Ausdruck ein entsprechendes Formatelement in dem Formatstring angegeben werden. Dabei ist es wichtig, daß der Typ des Ausdruck mit dem durch das Formatelement beschriebenen Typ übereinstimmt. Ist dies nicht der Fall so werden die Ausgaben verfälscht.

Der grundsätzliche Aufbau eines Formatelementes sieht wie folgt aus:

```
%[Flags][Breite][.Präzision][Typergänzung]Typ
```

Dabei bedeuten:

% Das Prozentzeichen kennzeichnet ein Formatelement. Ein Formatelement beginnt also immer mit diesem Zeichen. Soll im Formatstring wirklich ein Prozentzeichen angegeben werden, so muß zu diesem Zweck ein doppeltes Prozentzeichen (%%) eingesetzt werden.

Typ Folgende Ausgabetypen stehen zur Verfügung:

Typ	Parametertyp	Wirkung/Interpretation des Ausdruckes
d	integer	dezimal
i	integer	dezimal
o	integer	vorzeichenlos oktal
u	integer	vorzeichenlos dezimal
x	integer	vorzeichenlos hexadezimal (a..f)
X	integer	vorzeichenlos hexadezimal (A..F)
f	Gleitkomma	[-]dddd.dddd
e	Gleitkomma	[-]d.dddd e [+|-]ddd
g	Gleitkomma	das kürzere von f und e
E	Gleitkomma	wie e aber großes E
G	Gleitkomma	wie g aber großes E
c	integer	einzelnes Zeichen
s	Zeichenkette	Ausgabe der Zeichenkette bis zum Nullzeichen
p	Zeiger	gibt einen Zeiger aus, in Abhängigkeit von der Typergänzung im folgenden Format: Near: AAAA Far: SSSS:OOOO (Segment:Offset)

Breite Hier ist eine Zahl einzutragen, die die Breite des Feldes angibt, in dem die Ausgabe erfolgen soll. Alternativ dazu kann hier auch ein Sternchen (*) angegeben werden. In diesem Fall muß dem auszugebenen Ausdruck ein Ausdruck vom Typ int vorangestellt werden, der die Breite des Feldes bestimmt.

Präzision

Hier ist ebenfalls eine Zahl anzugeben, der ein Punkt vorangestellt werden muß. Alternativ dazu kann wieder das Sternchen eingesetzt werden. Diese Angabe beschreibt bei der Ausgabe von Gleitkommawerten die Anzahl der Nachkommastellen. Bei int-Zahlen wird die Anzahl der ausgegebenen Zeichen bestimmt, es werden also führende Nullen mit ausgegeben. Bei Zeichenketten bestimmt dieser Wert die Anzahl der auszugebenen Zeichen.

Flags

Sofern keine Flags gesetzt werden, erfolgt die Ausgabe rechtsbündig in einem Feld, das so breit ist, wie unter Breite angegeben ist. Mit Hilfe der Flags kann dies noch variiert werden.

- `-` Ermöglicht die linksbündige Ausgabe.
- `+` (nur bei Zahlenformaten) Es werden die Vorzeichen mit ausgegeben. Normalerweise geschieht dies nur bei negativen Zahlen. Mit dem Flag + also auch bei positiven Zahlen.
- `' '` (Leerzeichen, nur bei Zahlenformaten) den positiven Zahlen werden Leerzeichen vorangestellt.
- `#` Ermöglicht alternative Zahlenformate. Bei c, s, d, i und u bleibt diese Angabe ohne Wirkung. Bei den anderen Formaten erhält man die folgende Wirkung:
  - `o` Es wird der Ausgabe eine Null (0) vorangestellt.
  - `x,X` Es wird der Ausgabe 0x bzw. 0X vorangestellt.
  - `e,E,f` Es wird in jedem Fall ein Dezimalpunkt ausgegeben.
  - `g,G` Wie bei e,E zusätzlich werden folgende Nullen nicht unterdrückt.

Typergänzung

Folgende Angaben sind möglich:

- `h` Die Argumente werden als short interpretiert (Formate: d,i,o,u,x und X).
- `l` Die Argumente werden als long (Formate d,i,o,u,x und X) bzw. als double interpretiert (Formate e,E,f,g und G).
- `L` Die Argumente werden als double interpretiert (Formate e,E,f,g und G).
- `F` Zeiger werden bei der Ausgabe als FAR interpretiert.
- `N` Zeiger werden bei der Ausgabe als NEAR interpretiert.

Die Unterscheidung zwischen FAR und NEAR ist durch die Segmentierung bei den Intel-Prozessoren bedingt und deshalb in der Regel nur bei C-Compilern für PCs verfügbar.

Sie sehen, daß Sie vielfältige Möglichkeiten bei der Benutzung der printf-Funktion haben. Wenn an dieser Stelle zu jeder Möglichkeit ein Beispiel gegeben werden sollte, so würde dies den Rahmen des Skripts sprengen. Daher wollen wir uns auf einige exemplarische Beispiele beschränken:

Beispiel

Zeichenketten

```
char s[80];
strcpy ( s, "Eine Zeile" );
printf ( "%20s\n", s );
printf ( "%-20s\n", s );
printf ( "%20.8s\n", s );
```

Es ergibt sich folgende Ausgabe:

```
1234567890123456789
          Eine Zeile
Eine Zeile
            Eine Zei
```

numerische Ausgabe

```
unsigned u=0XAF2C;
int i=12;
float f=3.1459;
printf ( "%#10X\n", u );
printf ( "%10.5d KM\n", i );
printf ( "PI= %+10.4f\n", f );
printf ( "%*.*f %%\n", i, 2, f );
```

Es ergibt sich folgende Ausgabe:

```
1234567890123456789
    0XAF2C
     00012 KM
PI=    +3.1459
        3.15 %
```

Funktionswert

printf ist als Funktion vom Typ int deklariert und liefert als Funktionswert die Anzahl der ausgegebenen Zeichen.

7.2 putchar( )

Diese Funktion gibt ein einzelnes Zeichen auf der Standardausgabedatei aus. Diese Datei heißt in 'C' stdout und ist in der Regel mit der DOS-Datei CON identisch. Das Argument der Funktion muß eine Konstante, Variable oder ein Ausdruck vom Typ char sein. Als Funktionswert wird der ASCII-Code der ausgegebenen Zeichenkonstante zurückgegeben. Die Wirkungsweise der Funktion soll das folgende Beispiel verdeutlichen:

Beispiel

```
char x=65;
putchar ( x );
putchar ('A');
putchar ( 65 );
putchar ('\n');
putchar ( putchar ( putchar ( x ) ) );
putchar ('\n');
putchar ( 1+48 );
```

Es ergibt sich folgende Ausgabe:

```
12345678901234567890
AAA
AAA
1
```

Bei der Ausgabe können Fehlersituationen eintreten, z.B. wenn das Ausgabegerät nicht bereit ist, für einen solchen Fall liefert die Funktion den Funktionswert -1, das ist der Code für EOF (=End Of File, Ende der Datei). Der Funktionswert hat den Typ int.

7.3 puts( )

Diese Funktion dient zur Ausgabe einer Zeichenkette an die Standardausgabedatei (stdout). Im Gegensatz zur printf-Funktion wird bei puts( ) als letztes automatisch ein Zeilenvorschub mit ausgegeben.

Beispiel

```
puts ( "hello, world" );
printf ( "hello, world\n" );
```

Beide Funktionsaufrufe haben exakt dieselbe Wirkung. puts liefert als Funktionswert eine 0, wenn die Ausgabe erfolgreich war und einen Funktionswert ungleich 0, wenn die Ausgabe nicht geglückt ist. Der Funktionswert hat den Typ int.

Zeichenketten

An dieser Stelle soll ein Thema vorweg genommen werden, dem an anderer Stelle noch ein eigenes Kapitel gewidmet wird. Wenn man Zeichenkettenvariablen verwendet, so sind diese immer ein Zeiger auf eine Speicherzelle vom Typ char. Auch Felder (in 'C' spricht man von Vektoren) werden durch Zeiger repräsentiert (siehe auch 1. Beispiel unter 8.1). Einer Funktion, die mit Zeichenketten operiert, muß als aktueller Parameter immer der Zeiger auf eine solche Zeichenkette übergeben werden. Eine Zeigervariable auf ein Zeichen (Zeichenkette) wird wie folgt deklariert:

```
char *x;
char y[80];
```

Im ersten Fall wird ein Zeiger auf eine Speicherzelle vom Typ char deklariert. Im zweiten Fall ist y durch die Deklaration als Vektor (Feld) quasi automatisch eine Zeigervariable geworden. Allerdings wird y vom Übersetzer immer als Vektor interpretiert somit ist die Zuweisung einer Zeichenkette an y nicht möglich. Es muß hierfür eine Funktion (strcpy = STRing CoPY) eingesetzt werden. Während einer Variablen, die als Zeiger auf char deklariert ist, direkt eine Zeichenkette zugewiesen werden kann.

Beispiel

```
x = "erste Zeile";
strcpy ( y, "zweite Zeile" );
puts ( x );
puts ( y );
```

In der ersten Zeile wird durch die Zuweisung der Zeiger x auf den Anfang der Zeichenkettenkonstante "erste Zeile" gesetzt. In der zweiten Zeile wird mit Hilfe der Funktion strcpy( ) die Zeichenkette "zweite Zeile" in den den für y reservierten Speicherbereich kopiert. Mit der Funktion puts( ) werden die beiden Zeichenketten dann schließlich auf der Standardausgabe ausgegeben.

Aufgabe 18

Simulieren Sie mit einem Programm die Funktion:

```
printf ( "%d\n", i )
```

i soll in dem Programm als Variable definiert und mit einem geeigneten Wert vorbesetzt werden. Die Ausgabe soll mit putchar erfolgen.

Aufgabe 19

Ersetzen Sie in den beiden folgenden Anweisungen die Fragezeichen durch geeignete Ausdrücke!

19.1 `putchar ( ??? )`

Gewünschte Wirkung: ABC

19.2 `for ( ???; ???; ??? )`

Gewünschte Wirkung: ABC ... bis XYZ

Aufgabe 20

Benutzen Sie die printf-Funktion und die nachfolgend deklarierten Konstanten.

```
char  s[]="Kernighan & Ritchie";
int   i  = 4095;
float f  = 2.718281828;
```

Die folgende Ausgabe ist erwünscht:

```
12345678901234567890123456789
                4095
          0000004095
               07777
        0x0000000fff
FFF
2.71828e+000
          +2.7182817
        2.71828E+000
           Kernighan & Ritchie
Kernighan
                     Kernighan
                       Ritchie
```

7.4 scanf( )

Diese Funktion dient zur formatierten Eingabe, sie kann als das Gegenstück zu printf angesehen werden. Die allgemeine Form der Funktion sieht wie folgt aus:

```
scanf ( Formatstring, zeiger1, zeiger2, ... );
```

Der Formatstring ist auch bei dieser Funktion wieder zwingend vorgeschrieben.

zeiger1, zeiger2 usw. sind Adressen von Variablen bzw. Zeigervariablen die auf die Speicherzellen zeigen, die die eingegebenen Werte aufnehmen sollen.

Es gibt zwei Möglichkeiten diese Zeiger zu realisieren.

Beispiel

Mit dem Adreßoperator

```
int a,b,c,d;
scanf ( "%d%d%d", &a, &b, &c );
d = a + b + c;
printf ( "%d", d );
```

Mit Zeigervariablen

```
int *a,*b,*c,*d;
int u,v,w,x;
a = &u, b = &v, c = &w, d = &x;
scanf ( "%d%d%d", a, b, c );
*d = *a + *b + *c;
printf ( "%d", *d );
```

Im ersten Beispiel wird der Adreßoperator (&) benutzt, um die Adressen der Variablen a, b und c zu ermitteln. Während im zweiten Beispiel Zeigervariablen benutzt werden. Welche Variante benutzt werden soll, muß man im konkreten Einzelfall entscheiden.

Als Formatelemente können, die Formatelemente angegeben werden, die auch bei der printf-Funktion zugelassen sind. Zur Eingabe von Zeichenketten wäre folgende Programmsequenz denkbar:

```
char *x;
char y[80], z[80];
x = &z[0];
scanf ( "%s%s", x, y );
```

Hierbei fällt auf, daß auch bei y der Adreßoperator weggelassen wird. Wie im vorhergehenden Kapitel bereits erläutert, handelt es sich bei Vektoren (Feldern) auch um Zeiger.

Nun noch einige Anmerkungen zur Eingabe:

white spaces

Mit 'white spaces' sind die Zeichen gemeint, die als Trennzeichen zwischen den Eingaben erlaubt sind. Es sind dies:

<TAB> <SPACE> <RETURN>

Falls der Formatstring andere Zeichen als 'white spaces' enthält, so müssen diese bei der Eingabe mit eingegeben werden.

Beispiel

```
int a,b,c;
scanf ( "%d%d %d", &a, &b, &c );
```

Mögliche Eingaben wären:

```
1 2 3
```

oder

```
1
2
3
```

oder

```
1          2
3
```

Beispiel

```
int a;
scanf ( "a= %d", &a );
```

Hierbei muß 'a=' ebenfalls eingegeben werden:

```
a= 45
```

Würde 'a=' nicht eingegeben, so würde die Funktion scanf einen Fehler melden.

Funktionswert

scanf ist eine richtige Funktion, sie liefert als Funktionswert die Anzahl der eingelesenen Werte. Für den Fall, daß keine Werte eingelesen werden konnten, hat die Funktion den Wert 0 (Fehlerfall). Wird versucht über das Ende einer Datei hinauszulesen, so wird der Funktionswert EOF (-1) zurückgeliefert.

7.5 getchar( )

Diese Funktion holt ein Zeichen aus der Standardeingabedatei (stdin). Sie liefert als Funktionswert den ASCII-Code des Zeichens zurück bzw. EOF (-1), wenn das Dateiende erreicht wurde. Diese Funktion hat keinen Parameter, trotzdem müssen die (leeren) Klammern beim Aufruf angegeben werden:

Beispiel

```
int x;
char y;
x = getchar( );
y = getchar( );
```

Erforderliche Eingabe:

```
    ab <RETURN>
```

oder

```
a <RETURN>
b <RETURN>
```

Die Funktion gelangt erst dann zur Ausführung, wenn die <RETURN>-Taste gedrückt wird. Sind bis zum Drücken der <RETURN>-Taste mehrere Zeichen eingegeben worden, so befinden sich diese in einem 'C'-intern Puffer. Beim den nächsten Aufrufen von getchar werden zuerst alle Zeichen aus diesem internen Puffer herausgeholt. Erst wenn der Puffer wieder leer ist, wird wieder auf das Drücken der <RETURN>-Taste gewartet.

An dieser Stelle soll nun noch das Gerüst für Filterprogramme angeben werden, wie sie zum Anfang dieses Kapitels beschrieben wurden:

Beispiel

```
#include <stdio.h>

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

Dieses Programm liest zeilenweise die Eingabe von stdin und gibt diese dann unverändert an die Standardausgabe aus. Man gibt hierbei eine Zeile ein und schließt diese mit der <RETURN>-Taste ab, wodurch der gepufferten Eingabe das Ende signalisiert wird. Nun wird jedes Zeichen aus dem Puffer gelesen, c zugewiesen und mit putchar auf der Standardausgabe ausgegeben. Wenn EOF erreicht wird, wird die Schleife verlassen und das Programm beendet.

Beispiel

Ein möglicher Ablauf des Programms kann wie folgt aussehen:

```
Programmieren in C ist
Programmieren in C ist
schön
schön
^Z  ← zeigt bei MS-DOS das Dateiende an (Taste F6)
```

In Borland-C bzw. MS-C gibt es übrigens zwei ähnliche Funktionen, die allerdings nicht auf das Drücken der <RETURN>-Taste warten, also ein Zeichen direkt von der Tastatur abholen. Es sind dies die Funktionen getch bzw. getche. Wobei bei getch, die Zeichen nicht am Bildschirm angezeigt werden, während sie bei getche angezeigt (geechot) werden.

Hinweis

getchar und putchar wurden hier als Funktionen vorgestellt, das ist genau genommen falsch. Wie wir später sehen werden, handelt es sich bei getchar und putchar nicht um Funktionen sondern um Makros.

7.6 gets( )

Mit Hilfe dieser Funktion kann man auf bequeme Weise eine Zeichenkette einlesen. Die Zeichenkettenvariable muß wie scanf wieder als Zeiger auf char vereinbart werden. Im Gegensatz zu scanf ("%s"...) darf die Zeichenkette hierbei auch Leerzeichen und Tabulatoren enthalten. Das Ende der Zeichenkette wird durch das Drücken der <RETURN>-Taste angezeigt, intern wird der Zeichenkette dadurch das Endezeichen '\0' angehängt.

Funktionswert

Als Funktionwert liefert diese Funktion einen Zeiger auf die eingelesene Zeichenkette zurück. Für den Fall eines Fehlers oder bei Dateiende wird der Nullzeiger (Konstante NULL) zurückgeliefert.

7.7 Variationen der vorgestellten Funktionen

Alle unter den vorhergehenden Punkten aufgeführten Funktionen sind in den Standardbibliotheken mehrfach vorhanden. Sie unterscheiden sich von den hier vorgestellten Funktionen durch einen oder mehrere vorangestellte Buchstaben und, abhängig von diesem Buchstaben, dann auch in ihrer Wirkung. Ich möchte dies am Beispiel der printf-Funktion kurz zeigen:

(v)printf - print formated
Ausgaben erfolgen mit stdout, \n wird automatisch in CR/LF ungewandelt.

cprintf - console printf
Direkte Ausgabe in den Bildschirmspeicher (oder bei anderen Systemen an die Console). Zeilenvorschübe werden nicht verarbeitet.

(v)fprintf - file printf
Die Ausgaben erfolgen mit der angegebenen Datei.

(v)sprintf - string printf
Die Ausgabe gelangt in die angegebene Zeichenkette.

Mit vorangestelltem v benutzen die Funktionen eine andere Form der Parameterübergabe. An der oben aufgeführten Liste sehen nochmals die Leistungsfähigkeit und die Flexibiltät des Funktionskonzeptes bei 'C'. Ein Ausschöpfen aller denkbaren Möglichkeiten wird wohl erst mit einiger Programmiererfahrung möglich werden.

Aufgabe 21

Schreiben Sie ein Programm, das einzelne Zeichen über die Tastatur einlesen kann und diese zeilenweise wieder ausgibt.

Dabei soll zu jedem Zeichen der ASCII-Code in dezimaler, hexadezimaler und dualer Form ausgegeben werden. Durch Eingabe des Dateiende-zeichens kann das Programm beendet werden.

8 Funktionen

'C' bietet die Möglichkeit Programme zu modularisieren und in Unterprogramme zu zerlegen. Die Unterprogramme heißen in 'C' Funktionen. Es gibt in 'C' nur Funktionen und nicht wie in anderen Programmiersprachen noch Prozeduren oder Subroutines. Im Gegensatz zu Pascal darf eine Funktionsdeklaration keine weiteren Funktionsdeklarationen enthalten.

main

Auch das Hauptprogramm (main) ist eine Funktion, diese Funktion kann allerdings nur durch das Betriebssystem aufgerufen werden.

Über die Bibliotheken stellt 'C' je nach Implementation bereits eine große Menge von Standardfunktionen für die unterschiedlichsten Anwendungsfälle zur Verfügung. Diese Funktionen können in eigenen Programmen benutzt werden. Wichtige Voraussetzung dafür ist Einbinden der zugehörigen Header-Dateien (.h) mit den sogenannten Funktionsprototypen.

8.1 Funktionsdefinition

Eine Funktionsdefinition hat in 'C' die folgende allgemeine Form:

```
Ergebnistyp Funktionsname ( Parameterliste )
Vereinbarung der formalen Parameter
/* bei K&R, nicht bei ANSI */
{
  Funktionskörper
}
```

Alle Angaben außer Funktionsname, () und {} sind optional. Demnach wäre

```
xyz( ) { }
```

eine korrekte Funktionsdefinition.

Ergebnistyp

Diese Angabe zeigt an, welchen Datentyp der Funktionswert hat. Wenn diese Angabe entfällt, wird automatisch der Typ int angenommen. Den Funktionswert kann man mit der return-Anweisung zuweisen. Wird von der return-Anweisung kein Gebrauch gemacht bzw. wird die return-Anweisung ohne Parameter aufgerufen, dann ist der Funktionswert undefiniert. Die Funktion main hat als Ergebnistyp den Typ int. Unter DOS kann der Rückgabewert der Funktion main mit der Anweisung if errorlevel ausgewertet werden.

Wenn als Ergebnistyp der Typ void festgelegt wird, dann kann die Funktion keinen Funktionswert zurückliefern und nur wie eine Prozedur eingesetzt werden. Die Verwendung einer mit dem Ergebnistyp void definierten Funktion in einem Ausdruck ist also nicht erlaubt.

Beispiel

```
void test( )
{
  ...
}

test( );              ist ein zulässiger Aufruf
a = test( );          ist nicht zulässig
if ( test( ) )        ebenfalls nicht zulässig
```

Fast alle Funktionen liefern Ergebnisse an denen erkennbar ist, ob die Funktion einwandfrei gearbeitet hat. Daher finden Sie oftmals Aufrufe in der folgenden Form:

```
if ( func( ) )
```

oder

```
if ( func( ) == ... )
```

Parameterliste

Mit der Parameterliste wird festgelegt, wieviele Parameter eine Funktion hat und welche Namen diese Parameter haben. Bei der Parameterliste unterscheiden sich der K&R- und der ANSI-Standard. Beim K&R-Standard werden die Typen der Parameter im Anschluß an den Funktionskopf definiert, während beim ANSI-Standard die Parameter direkt in der formalen Parameterliste mit den Typangaben versehen werden. Man sollte bei der Definition die Vereinbarungen des ANSI-Standards vorziehen, da diese bei den gängigen Entwicklungssystemen auf PCs dem Compiler eine Typprüfung der Aufrufparameter ermöglichen.

Beispiel

```
int sum ( int a, int b )
/* Deklaration nach dem ANSI-Standard */
{
  return ( a + b );
}


int sum ( a, b )
int a, b;
/* Deklaration nach K&R */
{
  return ( a + b );
}
```

Wir unterscheiden zwischen formaler Parameterliste und aktueller Parameterliste. Mit formalen Parametern sind die Angaben bei der Definition der Funktion gemeint. Mit aktuellen Parametern sind die Angaben beim Aufruf der Funktion gemeint.

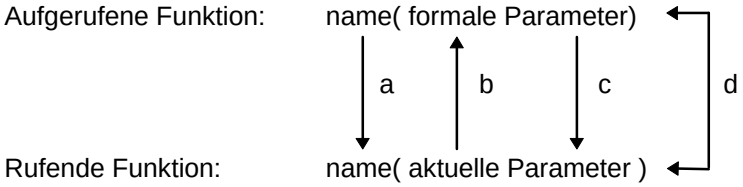
Funktionskörper

Der Funktionskörper kann alle bereits vorgestellten Anweisungen enthalten. Soll die Funktion einen Wert zurückliefern, dann muß der Funktionskörper mindestens eine return-Anweisung enthalten. Nach Ausführung der return-Anweisung ist die Ausführung der Funktion beendet. Eine Funktion kann auf drei Arten beendet werden:

- Wenn die abschließende geschweifte Klammer erreicht wird.
- Mit return(ausdruck), wenn die Funktion einen Wert zurückgeben soll. Der Ausdruck muß den Typ Ergebnistyp haben.
- Mit return, falls kein Funktionswert zurückgegeben werden soll.

8.2 Kommunikationskanäle zwischen Funktionen

Die rufende und die aufgerufene Funktion können über vier verschiedene Kommunikationskanäle Werte austauschen.



Dabei werden die einzelnen Wege wie folgt realisiert:

a Ausgabe mit Hilfe des Funktionswertes
b Eingabe mit Hilfe der Parameterlisten
c Ausgabe mit Hilfe der Parameterlisten
d Austausch über globale Variablen

Beispiel Eingabe über Parameterliste, Ausgabe über Funktionswert

```
#include <stdio.h>

int sum ( int x, int y )
{
  return ( x + y );
}

main ()
{
  int a=12;
  printf ( "%d", sum ( a, 3 ) );
}
```

In diesem Beispiel wird die Übergabe von Werten an die Funktion über die Parameterliste abgewickelt. Die Werte der Parameter werden in den lokalen Datenbereich der Funktion kopiert (Werteparameter, call by value). Innerhalb der Funktion können die Parameter wie lokal definierte Variablen benutzt und verändert werden. Veränderungen der Werte werden nach dem Verlassen der Funktion nicht in die Aufrufparameter übertragen. Daher können Sie beim Aufruf einer Funktion mit so deklarierten Parametern Konstanten, Variablen oder Ausdrücke als aktuelle Parameter angegeben. Durch die Anweisung return wird der Funktionswert der Funktion festgelegt. Dieser wird dann im Hauptprogramm für die Ausgabe weiterverwendet.

Beispiel

Eingabe und Ausgabe über Parameterliste

```
#include <stdio.h>

void sum ( int x, int y, int *z )
{
  *z = x + y;
}

main ()
{
  int a=12,b;
  sum ( a, 3, &b );
  printf ( "%d", b );
}
```

In diesem Beispiel wird sowohl die Übergabe von Werten an die Funktion als auch die Rückgabe des Ergebnisses der Berechnung über die Parameterliste abgewickelt. Dabei müssen bei den Ausgabeparametern nicht deren Werte sondern deren Adressen übergeben werden. So erhalten Sie eine Referenz auf die zu verändernde Variable. Durch den Sternoperator können die Adressen (Zeiger) dereferenziert werden und so deren Werte verändert werden (Referenzparameter, call by reference).

In C++ ergeben sich noch weitere Möglichkeiten zur Deklaration der formalen Parameterliste, z. B. die Deklaration als const oder einen anderen Mechanismus für die Verwendung von Referenzparametern:

```
#include <stdio.h>

void sum ( const int x, const int y, int &z )
{
  z = x + y;
}

main ()
{
  int a=12,b;
  sum ( a, 3, b );
  printf ( "%d", b );
}
```

Die Schreibweise mit dem Adreßoperator vereinfacht die Schreibung innerhalb der Funktion und beim Aufruf. Allerdings lassen sich beim Aufruf Referenz- und Werteparameter nicht mehr unterscheiden. Durch die zusätzliche Deklaration als const wird der Parameter x bzw. y in der Funktion als Konstante behandelt, darf also nicht verändert werden.

Beispiel

Austausch mit Hilfe globaler Variablen

```
#include <stdio.h>

int z,x,y;

void sum ( void )
{
  z = x + y;
}

main ()
{
  x=12;
  sum ();
  printf ( "%d", z );
}
```

In diesem Beispiel wird sowohl die Übergabe der Werte an die Funktion als auch die Rückgabe des Ergebnisses der Berechung mit Hilfe globaler Variablen vorgenommen. Die Verwendung globaler Variablen sollte zur Vermeidung von Seiteneffekten nur dann eingesetzt werden, wenn Sie unvermeidbar oder eindeutig sinnvoll ist. Grundsätzlich sollte man also die andereren Möglichkeiten der Kommunikation zwischen Funktionen vorziehen (Stichwort: Information Hiding).

Die verschiedenen Kommunikationskanäle können beliebig kombiniert werden.

Prozeduren

Falls eine Funktion wie eine Prozedur aufgerufen wird, ihr Funktionswert also nicht benötigt wird, soll sie den Typ void besitzen.

Alle im Bereich einer Funktion deklarierten Variablen sind lokal und können auch bei Namensgleichheit nicht von anderen Funktionen erreicht werden.

In Standard-'C' nach Kerninghan & Ritchie überprüft der Compiler nicht, ob beim Aufruf Typ und Anzahl der aktuellen Parameter mit denen der formalen Parameter übereinstimmen. Daher muß man sehr sorgfältig beim Aufruf der Parameter verfahren und gegebenenfalls den cast-Operator einsetzen.

8.3 Funktionsprototypen

Die Funktionsprototypen gleichen den Funktionsdefinitionen, jedoch ohne Funktionskörper und geschweifte Klammern, dafür wird die Angabe eines Funktionsprototypes mit einem Semikolon abgeschlossen.

```
Ergebnistyp Funktionsname ( Parameterliste );
```

Die Funktionsprototypen werden bei ersten Sprachbeschreibung von K&R nicht angegeben, deshalb verfügen sehr viele UNIX-Systeme noch nicht über die nachfolgend beschriebenen Möglichkeiten. Die Funktionsprototypen sind ein Bestandteil des ANSI-Standards bzw. von 'C++'.

Die Verwendung der Funktionsprototypen erscheint jedoch sinnvoll, da mit Hilfe der Prototypen dem Compiler die Möglichkeit gegeben wird, die Funktionsaufrufe beim Übersetzen zu kontrollieren und gegebenenfalls Typanpassungen vorzunehmen. Die Include-Dateien zu den Standardbibliotheken (.h bzw. .hpp-Dateien) enthalten im wesentlichen diese Funktionsprototypen. Deswegen ist es auch sinnvoll diese immer einzubinden. Folgende Möglichkeiten sind bei den Prototypen gegeben:

- `int test ( int a, int b );`

  Dieses Beispiel kann als Vorbild dienen. Hier wird quasi der vollständige Funktionskopf wiedergegeben, dies ermöglicht dem Compiler die Überprüfung der Anzahl der Parameter und der Typen und gegebenenfalls eine Typanpassung.

- `int test ( int, int );`

  Alternativ können die Bezeichner auch weglassen werden.

- `void test ( int, ... );`

  Bei variablen Parameterlisten, wie z. B. bei der Funktion printf, kann man durch drei Punkte (...) anzeigen, daß die Parameterliste nicht festgelegt ist. Dies bedeutet aber auch, daß der Compiler beim Übersetzen nur die eindeutig festgelegten Parameter kontrollieren und anpassen kann. Es müssen also zumindest die eindeutig festgelegten Parameter richtig angegeben werden. Nun dürfte auch klar sein, warum man bei den Funktionen printf und scanf selber auf die Übereinstimmung der Typen achten muß, und warum im Formatstring die Angabe der Typen durch die Formatelemente erforderlich ist. Der Zugriff auf die aktuellen Parameter eines Funktionaufrufs kann mit Hilfe der in der Header-Datei stdarg.h definierten Makros abgewickelt werden.

- `void test ( void );`

  Auch diese Deklaration ist vorbildlich, wenn man keine Funktionsparameter zulassen möchte, so sollte man dies durch void auch eindeutig anzeigen. Denn sonst hat der Compiler keine Möglichkeit die Parameterliste zu kontrollieren.

- `int test ( );`

  Bei diesem Prototyp wurde nur der Ergebnistyp eindeutig festgelegt nicht aber die Parameterliste. Der Compiler hat keine Möglichkeit zu erkennen, ob bei einem Funktionsaufruf, zuviel oder zuwenig Parameter angegeben wurden.

Aufgabe 22

Schreiben Sie ein Filterprogramm, das alle Buchstaben des deutschen Alphabetes groß ausgibt. Falls es Ihnen angebracht erscheint, schreiben Sie für die Umwandlung der Buchstaben eine Funktion. Das ß soll nicht berücksichtigt werden.

Aufgabe 23

Schreiben Sie eine Funktion:

```
float scanfloat ( void )
```

Diese Funktion soll eine Floatgröße über die Konsole einlesen. Für den Fall, daß ein falsches Zeichen eingegeben wird, soll ein Piepton ausgegeben werden und erneut eine Eingabe angefordert werden. Die Eingabe soll grundsätzlich im Format:

```
Ziffern[.Ziffern]
```

vorgenommen werden. Schreiben Sie außerdem einen Testrahmen für die Funktion.

8.4 Rekursion

Wir haben bereits gelernt, daß jede Funktion jede Funktion mit Ausnahme der Funktion main aufrufen kann. Speziell heißt das auch, daß eine Funktion sich selbst aufrufen kann. Wenn ein Funktion sich selbst aufruft, so spricht man von einem rekursiven Aufruf. Wir wollen uns zunächst ein Beispiel für eine rekursive Definition aus der Mathematik anschauen. Die Fakultätsfunktion, wir haben dazu bereits ein Übungsaufgabe gelöst, ist das klassische Beispiel für eine rekursive Definition.

Beispiel

Um den Wert von 6! zu bestimmen, müssen wir alle ganzen Zahlen von 1 bis 6 miteinander multiplizieren. Um den Wert 5! zu bestimmen müssen wir alle ganzen Zahlen von 1 bis 5 miteinander multiplizieren usw.. Wir könnten aber auch sagen:

$6! = 6 \cdot 5!$

$5! = 5 \cdot 4!$

Allgemein läßt sich das wie folgt definieren:

$n! = n \cdot (n-1)!$

Wichtig bei einer rekursiven Definition ist die Bestimmung eines Kriteriums mit dem die Rekursion endet. Denn sonst wäre die Definition nicht vollständig, da man nie ein Ergebnis bekommen würde. Deshalb lautet die vollständige rekursive Definition für die Fakultätsfunktion wie folgt:

$n! = n \cdot (n-1)!$ mit $0! = 1$

Damit kann nun für jede beliebige ganze Zahl der Wert der Fakultätsfunktion bestimmt werden. Eine rekursive Definiton läßt sich in 'C' auch direkt umsetzen, nämlich durch den Einsatz der Rekursion bei Funktionsaufrufen.

Beispiel

```
int fakul ( int n )
{
  if ( n == 0 )
    return 1;
  else
    return n * fakul( n - 1 );
}
```

Durch den Einsatz der Rekursion können sehr viele Programmieraufgaben wesentlich vereinfacht werden. Speziell bei der Berechnung der Fakultät ist der iterative Lösungsweg vorzuziehen, da er mindestens genauso einfach und wesentlicher schneller ist. Informatiker haben nachgewiesen, daß man jede rekursive Formel oder Aufgabenstellung iterativ programmieren kann. Wenn dadurch allerdings die Lösung wesentlich komplizierter wird, sollten Sie den rekursiven Weg vorziehen.

9 Vektoren (Felder)

Die Vektoren sind uns von anderen Programmiersprachen her als Felder oder Arrays bekannt. Ein Vektor ist ein strukturierter Datentyp. Er besteht aus einer bestimmten Anzahl von Elementen des gleichen Types, die in einem zusammenhängenden Speicherbereich abgelegt werden. Auf die Elemente eines Vektors kann mittels Indizierung zugegriffen werden. Ein Index hat in 'C' immer den Typ int und die Indizierung beginnt in 'C' grundsätzlich mit 0.

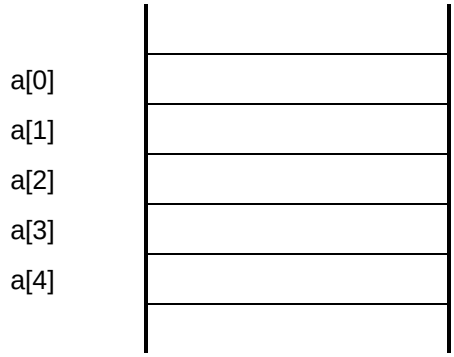
Die allgemeine Form der Vereinbarung eines Vektors ist:

```
Typ Name[Anzahl1][Anzahl2]...
```

Beispiel

```
char s[80];     /* Feld    s[0] .. s[79]   */
double a[5];    /* Feld    a[0] .. a[4]    */
int z[10][10];  /* Feld z[0][0] .. z[9][9] */
```

Die Elemente eines eindimensionalen Vektors werden nacheinander im Speicher abgelegt. Die folgende Abbildung zeigt dies am Beispiel des Vektors a.



Jeder Ausdruck vom Typ int ist als Index zulässig. Die von anderen Programmiersprachen her bekannte Art der Indizierung von mehrdimensionalen Feldern mit Hilfe von Kommas ist in 'C' nicht vorgesehen.

Beispiel

```
z[1,2]
```

Trotzdem liefert der Ausdruck im Beispiel beim Compilieren keine Fehlermeldung, da 1,2 ein zulässiger int-Ausdruck ist und den Wert 2 hat. Vielmehr müssen bei mehrdimensionalen Feldern für jeden Index eckige Klammern eingesetzt werden.

Beispiel

```
z[1][2]
```

Vektoren können für alle zuvor vereinbarten Datentypen und für die Standarddatentypen vereinbart werden.

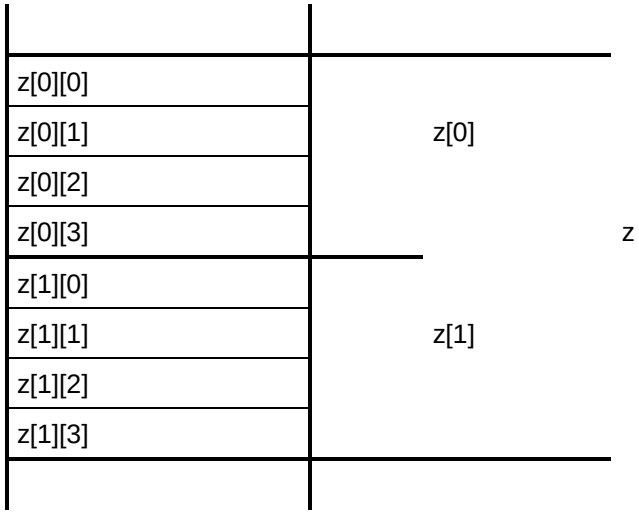
Interessant dürfte auch die Art der Abspeicherung bei den mehrdimensionalen Vektoren sein. Wir können uns einen mehrdimensionalen Vektor als Vektor von Vektoren vorstellen. Schauen wir uns die Abspeicherung eines mehrdimensionalen Vektors einmal an einem Beispiel an.

Beispiel

Wir gehen von der folgenden Deklaration aus:

```
int z[2][4];
```

Dieser Vektor wird wie folgt im Speicher abgelegt:



Mit z kann der gesamte Vektor angesprochen werden, mit z[0] bzw. z[1] können jeweils vier zusammenhängende Elemente (Teilvektoren) angesprochen werden (siehe Skizze). Ein einzelnes Datenobjekt kann z. B. mit z[1][2] angesprochen werden. Die Startadresse des gesamten Vektors ist gleich der Adresse des ersten Elementes, man kann sie also mit dem Ausdruck &z[0][0] ermitteln.

9.1 Initialisierung

Vektoren können bei der Deklaration auch direkt initialisiert werden. Allerdings ist die Initialisierung von Vektoren nur dann zulässig, wenn Sie statisch sind, d. h. entweder handelt es sich um global vereinbarte Vektoren oder um solche, die mit der Speicherklasse static vereinbart wurden.

Beispiel

```
/* Definition außerhalb von Funktionen */
double a[5] = { 1.2, 3.4, 4.5, 6.7, 8.9 };
int r[2][4] = {{ 1, 2, 3, 4 }, { 5, 6, 7, 8 }};
/* in Funktionen mit static */
static int z[4] = { 1, 2, 3, 4 };
```

Außerdem sind folgende Varianten bei der Initialisierung zulässig:

Beispiel

```
int a[20] = { 1, 2, 3 };
```

In diesem Fall werden nur die ersten drei Vektorelemente mit den angegebenen Werten initialisiert, die anderen Elemente werden zu Null gesetzt.

Beispiel

```
int a[] = { 1, 2, 3 };
```

In diesem Fall wird die Dimension des Vektors anhand der Anzahl der angegebenen Werte ermittelt also a[3].

Eine besondere Art von Vektoren sind die Zeichenketten. Auch sie können bei der Vereinbarung initialisiert werden.

Beispiel

```
char a[] = { 's', 'u', 's', 'i' };
char b[] = "susi";
```

Die Zeichenkette a hat vier Elemente. Die Zeichenkette b hat jedoch 5 Elemente, denn für b[4] wird automatisch '\0' eingesetzt, da bei der Vereinbarung die doppelten Hochkommas (Zeichenkettenkonstante) angegeben wurden.

Hinweis

In MS-C ist im Gegensatz zu K&R auch eine Initialisierung von nicht statischen Vektoren innerhalb von Funktionen möglich.

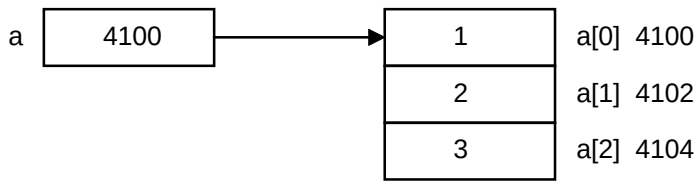
9.2 Adressierung

Ein Variable, die als Vektor vereinbart wurde, ist in 'C' immer ein Zeiger.

Beispiel

```
int a[3] = { 1, 2, 3 };
```

Für diesen Vektor werden im Speicher 4 Speicherbereiche reserviert. Zum einen der Speicherbereich für die drei Elemente des Vektors zum anderen ein Speicherplatz für die Anfangsadresse dieses Speicherbereiches:



Dabei ist die Variable a eigentlich ein Zeiger, der auf die Speicherstelle a[0] zeigt.

Intern wird vom 'C'-Compiler folgende Berechnung zur Bildung der eigentlichen Speicheradresse eines Elementes durchgeführt:

```
a[i] = *(a+i*sizeof(Typ))
```
(kein C-Code!)

Der Programmierer kann diese Notation ebenfalls benutzen, wobei er die Typgröße nicht angeben muß, da diese intern vom Compiler bestimmt wird.

Folgende Ausdrücke sind in 'C' gleichwertig:

```
a[i]    und    *(a+i)
&a[i]   und     (a+i)
```

Dies gilt auch für mehrdimensionale Vektoren.

Beispiel Vorausgesetzt wird folgende Definition:

```
typ a[M][N];
```

dann gilt:

```
a[i][k]   entspricht  *(a+i*N+k)
```

9.3 Vektoren als Funktionsparameter

Im Gegensatz zu z. B. Pascal dürfen in 'C' bei der Funktionsdeklaration bei den Vektoren, die als Parameter angegeben werden, die Feldgrenzen offen bleiben.

Beispiel

```
int name ( int a[], int anzahl )
{
  ..
}
```

Bei einer so deklarierten Funktion kann die aktuelle Anzahl der Feldelemente mit Hilfe des Parameters anzahl festgelegt werden. Die entsprechende Funktion muß natürlich so programmiert werden, daß die Feldgrenzen nicht überschritten werden. Mit der Variablen a wird nur die Anfangsadresse des Feldes übergeben.

Beispiel

```
int sum ( int a[], int anzahl )
{
  int i, sum=0;
  for ( i=0; i<anzahl; i++ )
    sum += a[i];
  return ( sum );
}
```

Betrachtung mehrdimensionaler Felder:

```
int a[M][N];
```

Ein Problem dürfte erkennbar sein, bei mehrdimensionalen Feldern kann der Compiler nicht die Dimension des zweiten Index kennen, denken Sie an die Formel:

```
a[i][k] ==  *( a + i * N + k )
```

Es muß also die Funktion so deklariert werden, daß die Dimension des zweiten Index (und weiterer Indizes) erkennbar ist.

Beispiel

```
int name ( int a[][N], int anzahl );
{
  ..
}
```

Wir wissen aber auch, daß die entsprechenden Feldelemente hintereinander im Speicher abgelegt werden. Folgender Aufruf der Funktion sum führt also auch bei mehrdimensionalen Felder zum Ziel:

```
sum ( a, M*N );
```

Hinweis

Der Compiler wird bei diesem Aufruf sicherlich eine Warnung ausgeben, da die formalen und die aktuellen Parameter nicht zusammenpassen.

Aufgabe 24

Zeigen Sie mit einem kurzen 'C'-Programm, wie ein Vektor vom Typ:

```
int r[3][3]
```

im Speicher abgelegt ist!

Aufgabe 25

Folgende Funktionen benötigen Sie für die Lösung dieser Aufgabe:

- void srand (unsigned startwert) <stdlib.h>

  Initialisiert den Zufallszahlengenerator

- int rand (void) <stdlib.h>

  Liefert als Funktionswert eine Zufallszahl

- long time (long *zeitzgr) <time.h>

  Liefert die Systemzeit. Diese Funktion soll benutzt werden, um den Zufallszahlengenerator zu initialisieren.

Beispiel

```
srand( (unsigned) time( NULL ) );
```

Schreiben Sie ein Programm, das ein eindimensionales Feld mit Zufallszahlen füllt und mit einer ebenfalls zu erstellenden Funktion minimum die kleinste Zahl aus einem Feld heraussucht!

Testen Sie die Funktion minimum auch für ein zweidimensionales Feld!

Aufgabe 26

Schreiben Sie eine Funktion:

```
int strupcase ( char s[] )
```

Diese Funktion soll alle Buchstaben des deutschen Alphabetes in der übergebenen Zeichenkette s in Großbuchstaben umwandeln. Das ß soll nicht berücksichtigt werden. Als Funktionswert soll die Länge der Zeichenkette zurückgegeben werden.

Schreiben Sie außerdem einen entsprechenden Testrahmen!

10 Zeiger

Zeiger sind Variablen, die Speicheradressen darstellen. Mit Hilfe der Zeiger kann auf andere Datenobjekte zugegriffen werden. Sie werden immer mit dem Typ des Datenobjektes in Verbindung gebracht, auf das sie zeigen. Datenobjekte eines anderen Datentyps können mit einem einmal vereinbarten Zeiger nicht direkt erreicht werden. Hierfür ist eine Cast-Operation notwendig. Die Vereinbarung eines Zeigers erfolgt mit dem Sternoperator.

Beispiel

```
int        *p;   → p ist ein Zeiger auf ein int-Objekt
char       *c;   → c ist ein Zeiger auf ein char-Objekt
double  *a[7];   → Vektor von 7 Zeigern auf double-Objekte
char      **z;   → Zeiger auf Zeiger auf ein char-Objekt
int       *f();  → Funktion, die einen Zeiger auf int ergibt
int     (*f)();  → Zeiger auf eine Funktion, die int ergibt
```

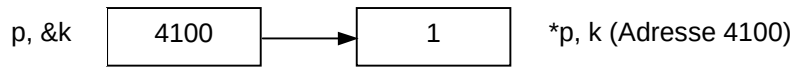
An diesen Beispielen erkennen wir bereits die vielfältigen Einsatzmöglichkeiten der Zeiger.

Im Programm dient der Sternoperator zum Erreichen des Inhalts einer Zelle auf die der Zeiger zeigt (Verweis). Der zum Sternoperator inverse Operator ist das Und-Zeichen (&). Er liefert uns die Adresse einer Speicherstelle.

Beispiel

```
int *p,k;

p = &k;
*p = 1;
printf("%d %d",*p,k);
```

Durch die Funktion printf wird zweimal die Zahl 1 ausgegeben. Wir wollen uns die Verhältnisse im Speicher anhand einer Grafik verdeutlichen.



Die vorgestellte Symbolik mit den Kästchen und den Pfeilen ist ein bewährtes Hilfsmittel bei der Programmierung mit Zeigern. Wenn Sie Aufgaben programmieren müssen, bei denen Zeiger verwendet werden, so sollten sie sich die Speicherstellen und die dazu gehörigen Zeiger mit dieser Symbolik aufzeichnen, um den Überblick zu behalten.

Interessant ist auch die Verwendung von Zeigern in Verbindung mit anderen strukturierten Datentypen, wie z.B. Vektoren. Da die Zeiger immer mit dem jeweiligen Datentyp in Verbindung gebracht werden, ist folgendes zulässig:

Beispiel

```
int *p;
int a[10];

p = &a[0];
p++;
```

Es wird hier jedoch keine echte Addition ausgeführt. Der Zeiger wird vielmehr auf das nächste Datenobjekt gesetzt. Es gilt:

```
 p ==  &a[1]  ==  (a+1)
```

und

```
*p ==   a[1]  == *(a+1)
```

Wir wollen nun weitere Operationen mit Zeigern betrachten:

```
int a[10],k=0,*p;
p=&a[0];
```

Danach gilt:

```
y = *(p++);   entspricht   y = a[k++];  ⎫ Zeiger bzw. Index
y = *(++p);   entspricht   y = a[++k];  ⎭ wird verändert
y = (*p)++;   entspricht   y = a[k]++;  ⎫ Wert der Speicherstelle
y = ++(*p);   entspricht   y = ++a[k];  ⎭ wird verändert
```

Vektoren

Mit Hilfe der Zeiger kann man nun auch Vektoren als Funktionsparameter dimensionslos übergeben. Die entsprechende Variante der Funktion sum sähe so aus:

Beispiel

```
int sum ( int *a, int anzahl )
{
  int i, sum=0;
  for (i=0; i<anzahl; i++)
    sum += *(a+i);
  return(sum);
}
```

Der Aufruf kann dann in der folgenden Form erfolgen:

```
int b[20], c;
c = sum( b, 20 )
```

Oder bei mehrdimensionalen Feldern:

Beispiel

```
int allesnull ( int *a, int anz1, int anz2 )
{
  int i,k;
  for (i=0; i<anz1; i++)
    for (k=0; k<anz2; k++)
      *(a+i*anz2+k)=0;
}
```

Rechenoperationen

Als Rechenoperationen mit Zeigern sind nicht nur die Addition sondern auch die Subtraktion und die Vergleichsoperatoren definiert. Die Subtraktion liefert bei p-q unter der Voraussetzung, daß p größer q ist, die Anzahl der zwischen p und q liegenden Datenobjekte. Alle Vergleichsoperatoren können für Zeiger auf denselben Datentyp eingesetzt werden. Eine etwas andere Lösung für die Funktion allesnull soll das zeigen:

Beispiel

```
int allesnull2 ( int *a, int anz1, int anz2 )
{
  int *p,*q;
  p=a;
  q=a+anz1*anz2;
  while ( p<q )
    *p++=0;
}
```

10.1 Zeiger auf Funktionen

Eine besondere Möglichkeit der Verwendung von Zeigern bietet 'C' mit Zeigern auf Funktionen. Ein Beispiel dafür wäre eine allgemeine Sortierfunktion sort, die einen eindimesionalen Vektor sortieren kann. Das dabei verwendete Sortierverfahren spielt bei der folgenden Überlegung keine Rolle. Klar dürfte sein, daß für eine allgemeine Sortierfunktion gelten muß, daß sie die entsprechenden Feldelemente richtig interpretiert. Da man jedoch den Aufbau der Feldelemte nicht kennt und dieser für den Sortieralgorithmus auch nicht maßgebend ist, kann man der Funktion sort mitteilen, welche Funktion für die speziellen Feldelemente den Vergleich machen kann. Ferner muß der Sortierfuktion für die notwendigen Verschiebe- und Kopieroperationen die Größe der Feldelemente bekannt sein. Wir wollen uns eine mögliche Realisierung einer solchen Funktion an einem Beispiel anschauen:

Beispiel

```
int vergleich ( int *a, int *b )
{
  return ( *a > *b );
}

int sort ( char *a,
           int anz,
           int esize,
           int (*comp)() )
{
  ..
  if ( comp ( &elema, &elemb ) )
    ..
  else
    ..
}

int a[20];

main()
{
  sort ( a, 20, sizeof(int), vergleich );
}
```

Für einen anderen Datentyp z.B. double a[30] wäre demnach folgender Aufruf notwendig:

```
sort ( a, 30, sizeof(double), vergleich2 );
```

wobei vergleich2 eine Funktion sein müßte, die für Werte vom Typ double die entsprechenden Vergleichsergebnisse liefert.

Der hier beschriebene Mechanismus wird übrigens auch bei den in den Standard-C-Entwicklungssystemen implementierten Funktionen qsort, lfind, lsearch und bsearch eingesetzt.

10.2 Cast-Operator für Zeigertypen

Wir nehmen an, es existiert eine Funktion f, die Zeiger auf Variablen vom Typ char liefert. Dies ist zum Beispiel denkbar für eine Funktion, die für dynamische Variablen Speicherplatz reserviert. Nun soll dieser Zeiger aber auf einen Speicherbereich zeigen, der als double-Größe interpretiert werden soll. Es muß also der Zeiger des Funktionsergebnisses (Zeiger auf char) übertragen werden, zu einem Zeiger auf den Typ double. Dazu ist allerdings eine Typkonvertierung notwendig.

Beispiel

```
char *f();
double *d;

...

{
  d = (double *)f(...);
}
```

10.3 Zusammenfassung

Auf Zeiger können int-Werte addiert und subtrahiert werden. Dabei werden diese int-Werte automatisch der Ausdehnung des vereinbarten Datenobjektes angepaßt.

Zeiger können voneinander subtrahiert werden, wenn sie auf Datenobjekte desselben Typs zeigen.

Zeigern können die Werte anderer Zeiger mit demselben Typ zugewiesen werden, insbesondere auch die konstanten Anfangsadressen von Vektoren. Soll einem Zeiger ein Zeiger auf einen anderen Typ zugewiesen werden, besteht die Möglichkeit, den cast-Operator einzusetzen.

Auf Zeiger des gleichen Typs können die Vergleichsoperatoren angewandt werden.

Um einen Zeiger als undefiniert zu kennzeichnen, wird ihm die Konstante NULL zugewiesen.

Sie können Zeiger mit der printf-Funktion und dem Formatelement %p ausgeben lassen.

Aufgabe 27 **Matrizenrechnung**

Schreiben Sie je eine Funktion die zweidimensionale Matrizen unabhängig von ihrer jeweiligen Größe bearbeiten kann.

1. Funktion zum Einlesen zweier Matrizen A und B
2. Funktion zur Ausgabe einer Matrix C
3. Funktion zum Addieren zweier Matrizen: C = A + B
4. Funktion zum Festlegen der Dimensionen für die Matrizen

Als Testrahmen schreiben Sie ein dialogorientiertes Programm zum Aufrufen der entsprechenden Operationen.

Für die Operationen sollen die global definierten Matrizen a,b, und c definiert werden mit der maximalen Dimension MAX x MAX, mit MAX = 10.

Aufgabe 28 **Zeiger auf Funktionen**

Testen Sie mit einem eindimensionalen Vektor vom Typ int mit 20 Elementen die Funktion qsort Ihres C-Entwicklungssystems.

Füllen Sie einen Vektor mit Zufallszahlen und lassen Sie ihn sich in absteigend und aufsteigend sortierter Reihenfolge auf dem Bildschirm ausgeben.

11 Strukturen

Strukturen sind Datenobjekte, die aus mehreren Teilobjekten bestehen. Diese dürfen im Gegensatz zu den Vektoren unterschiedliche Datentypen haben. Sie können die Strukturen mit dem von Pascal her bekannten Verbunddatentyp RECORD vergleichen. Ich möchte hier die 'C'-Deklaration einer Struktur der entsprechenden Pascal-Deklaration gegenüberstellen.

Beispiel

```
struct pkw
{
  char  marke[13]
  int   baujahr;
  float preis;
}

struct pkw auto;
struct pkw autos[20];
```

```
TYPE
pkw = RECORD
  marke   : string[12];
  baujahr : integer;
  preis   : real;
END;

VAR
auto  : pkw;
autos : ARRAY[1..19] OF pkw;
```

Ein etwas abweichende Möglichkeit der Deklaration wäre:

```
struct
{
  char  marke[12];
  int   baujahr
  float preis;
} auto;
```

```
VAR
auto : RECORD
       marke   : string[12];
       baujahr : integer;
       preis   : real;
       END;
```

In beiden Sprachen hat man mit den vorgestellten Deklarationen exakt dieselbe Wirkung erzielt. Die zweite Möglichkeit unterscheidet sich von der ersten dadurch, daß zwar ein Variable als Struktur (Verbund) definiert wurde, der Datentyp aber für weitere Operationen und Deklarationen nicht zur Verfügung steht. Man wird also in der Regel die erste Variante einsetzen. In 'C' kann mit der Typdeklaration auch direkt die Variablendeklaration verbunden werden.

Beispiel

```
struct pkw
{
  wie oben
} auto, autos[20];
```

Diese Variante ist in Pascal so nicht möglich.

Sie können nun auch weitere Variablen oder gar Funktionen mit dem deklarierten Typ versehen.

Beispiel

```
struct pkw *autoeingabe ( ... );
```

Punktoperator

Auf die Komponenten einer Struktur kann man mit dem, vielleicht schon von Pascal her bekannten, Punktoperator zugreifen. Die folgenden Beispiele sollen dies verdeutlichen.

Beispiel

```
struct pkw a, b[10], *p;

a.preis        = 9500.00;
b[9].baujahr   = 1980;
p              = &b[0];
(*p).baujahr   = 1981;
(*(p+1)).preis = 1000;
b[3].marke[0]  = getchar();
```

Pfeiloperator

Man erkennt, daß der Punktoperator eine höhere Priorität besitzt als der Sternoperator. Daher müssen im Falle der Verwendung von Zeigern die Klammern gesetzt werden. Zu dieser Notation gibt es mit dem Pfeiloperator eine anschauliche Alternative:

Beispiel

Wenn vereinbart wurde:

```
struct pkw a, *p;

p = &a;
```

Dann gilt:

```
a.preis == (*p).preis == p->preis == (&a)->preis
```

Strukturen können auch Elemente vom Strukturtyp enthalten.

Beispiel

```
struct pkwdaten
{
  float       einkauf;
  struct pkw  kfz;
} a, *p;

a.einkauf      = 1000;
a.kfz.preis    = 2000;
p              = &a;
p->kfz.baujahr = 1970;
```

11.1 Dynamische Datenstrukturen

Zeiger

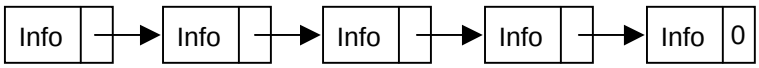
Die Komponenten einer Struktur können praktisch jeden Typ besitzen, jedoch nicht den Typ der Struktur selbst. Sehrwohl ist es aber möglich innerhalb einer Struktur Zeiger auf eine Struktur zu definieren, die den Typ der Struktur selbst hat. Mit Hilfe solcher Datenobjekte lassen sich Datenstrukturen wie linear verkettete Listen und Bäume erzeugen.

Solche Datenstrukturen setzt man typischerweise dann ein, wenn zum Zeitpunkt der Programmerstellung die Anzahl der zu erwartenden Datenobjekte nicht feststeht. Man nennt solche Datenstrukturen dynamische Datenstrukturen.

Lineare Listen

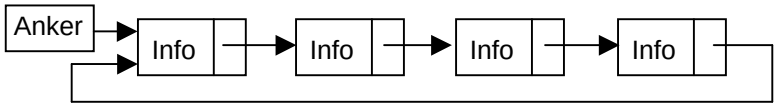
Die einzelnen Datenobjekte einer linearen Liste sind Strukturen, die aus einem Informationsteil und einem Verweis auf das nächste Element bestehen.

Eine einfach verkettete Liste kann wie folgt dargestellt werden:



Jedes Datenobjekt enthält eine Komponente die auf das jeweils folgende Datenobjekt zeigt. Das letzte Datenobjekt enthält als Kennzeichnung des Endes der linearen Liste die Konstante NULL.

Würde statt der NULL in der Verweiskomponente des letzten Elementes ein Verweis auf das erste Elemente der Liste stehen, spräche man von einer Ringliste.

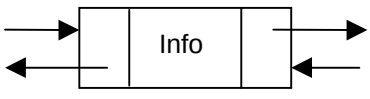


Das zusätzliche Element ohne Info-Teil ist lediglich ein Zeiger auf den Anfang der Liste, der sogenannte Anker. Man kann ihn benutzen, um den Anfang und das Ende der Liste erkennen zu können.

Um die Listenelemente ansprechen zu können, benutzt man eine weitere Zeigervariable mit der man durch die Liste wandern kann. Dieser Zeiger zeigt dann immer auf das gerade ausgewählte Listenelement. Ein Nachteil dieser Vorgehensweise dürfte bereits zu erkennen sein. Es ist zwar sehr einfach das jeweils nächste Listenelement zu erreichen, aber um das vorhergehende Element zu erreichen, muß man den Arbeitszeiger einmal rund um die Liste wandern lassen. Bei bestimmten Aufgabenstellungen setzt man daher oftmals 2 Arbeitszeiger ein.

Eine weitere Möglichkeit ist der Einsatz sogenannter doppelt verketteter Listen. Hierbei wird zusätzlich in jedem Listenelement eine Komponente mit einem Zeiger auf den jeweiligen Vorgänger gespeichert. Auch bei dieser Variante lassen sich Ringlisten einrichten.

Ein einzelnes Listenelement sähe dann so aus:



Die für die Listen benötigten Datenobjekte lassen sich in 'C' sehr leicht als Strukturen abbilden. Häufig werden die Listen im Speicher direkt sortiert angeordnet. Es lassen sich nämlich durch Umkettung Listenelemente einfügen bzw. ausketten.

Speicherplatz

Bei den dynamischen Datenstrukturen ist die dynamische Zuordnung von Speicherplatz eine sehr wichtige Operation. Mit Hilfe der folgenden Bibliotheksfunktionen läßt sich während der Laufzeit Speicherplatz vom System anfordern.

```
char *malloc( unsigned size );
```

bzw.

```
char *calloc( unsigned nelem, unsigned elsize );
```

malloc

Die Funktion malloc liefert einen Zeiger auf einen Speicherbereich mit einer Größe von size Bytes. Der bereitgestellte Speicherplatz wird nicht initialisiert.

calloc

Die Funktion calloc liefert einen Zeiger auf einen Speicherbereich mit dem Umfang von nelem Elementen, wobei jedes die Größe von elsize Bytes besitzt. Der Bereich wird mit Nullen initialisiert.

Fehlerfall

Falls kein Speicherplatz mehr bereit gestellt werden kann, liefern die Funktionen als Funktionswert den NULL-Zeiger.

Die Funktionen gehören zu der Bibliothek <stdlib.h>.

free

Der mit den Funktionen malloc und calloc bereitgestellte Speicherplatz kann mit der Funktion free wieder freigegeben. Als Aufrufparameter muß der Funktion free ein Zeiger auf den freizugebenden Speicherbereich übergeben werden.

```
void free( char *zeiger );
```

Nach auf Aufruf der Funktion ist der durch den übergebenen Zeiger referenzierte Speicherbereich wieder freigegeben. Der Inhalt des Speicherbereiches ist nicht mehr definiert.

Hinweis

In C++ stehen Ihnen zwei Operatoren zum Anfordern und Freigeben von dynamisch benötigten Speicherplatz zur Verfügung. Dies sind die Operatoren new und delete.

Beispiel

```
struct pkw *p;

p = new struct pkw;
char *lpchar = char [20];
```

Mit dem letzten Beispiel wird dynamisch der Speicherplatz für eine Zeichenkette reserviert und gleichzeitig der Zeiger auf diesen Speicherbereich deklariert. Im Zusammenhang mit den Klassen ist auch das Overloading, das Umdefinieren von Operatoren möglich, so daß Sie die Möglichkeit haben mit dem Operator new gleichzeitig Ihre Datenstrukturen zu initialisieren die Parameter für die Initialisierung werden dann ggf. in Klammern angegeben. So wäre auch folgende Konstruktion denkbar.

Beispiel

```
mytype *p = new( "Test", 200, 20) mytype;
```

Zum Freigeben von Speicherplatz, der mit dem Operator new reserviert wurde, benutzen Sie den Operator delete.

Beispiel

```
delete p;
```

Wir wollen uns die Wirkung der Funktionen malloc und calloc am Beispiel einer einfach verketteten Liste verdeutlichen:

Beispiel

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

struct liste
{
 int          info;  /* Info-Teil des Datenobjektes */
 struct liste *next; /* Verweis auf das nächste Datenobjekt */
};

main ()
{
  int eingabe  = 1; flag = 1;
  struct liste *p, *q, *anker;
  /* Eingabeschleife */
  scanf( "%d", eingabe );
  while ( eingabe )
  {
    /* Speicherplatz reservieren */
    p = (struct liste*) calloc( 1, sizeof(struct liste));
    /* neues Datenobjekt füllen */
    p->info = eingabe;
    p->next = NULL;
    /* Anker initilisieren bzw. neues Element anhängen */
    if ( flag )
      anker = p, flag = 0;
    else
      q->next = p;
    /* Arbeitszeiger weitersetzen */
    q = p;
    /* nächste Information einlesen */
    scanf ( "%d", eingabe );
  }

  /* Ausgabeschleife */
  while ( anker )
  {
    /* Information ausgeben */
    printf ( "%8d", anker->info );
    /* Arbeitszeiger weitersetzen */
    p = anker;
    anker = anker->next;
    /* Ausgegebenes Listenelement löschen */
    free( p );
  }
}
```

Dieses einfache Beispiel soll die Vorgehensweise bei der Behandlung linearer Listen verdeutlichen. Sicherlich können noch vielfältige Verbesserungen angebracht werden. Insbesondere eine Überprüfung des Ergebnisses der calloc-Funktion wäre angebracht.

Die wesentlichen Aufgaben bei der Behandlung der linearen Listen sind:

- Hinzufügen eines Listenelementes
- Einfügen eines Listenelementes
- Löschen eines Listenelementes
- Suchen eines Listenelementes

Beim Hinzufügen und Einfügen eines neuen Elementes in die Liste muß der Speicherplatz mit Hilfe der Funktion malloc oder calloc reserviert werden. Beim Löschen eines Listenelementes muß der belegte Speicherplatz mit Hilfe der Funktion free wieder freigegeben werden. Wichtiger ist aber das korrekte Ändern der in der Liste enthaltenen Verweise, sodaß z. B. beim Löschen nicht der gesamte Rest der Liste verloren geht.

Die oben beschriebenen Aufgaben können als Funktionen realisiert werden. Wobei die Algorithmen nach dem Muster von qsort allgemein beschrieben werden können und die variablen Teile über die Parameterliste übergeben werden. Häufig werden solche Algorithmen auch rekursiv programmiert.

Aufgabe 29

Es soll ein Programm erstellt werden, das anhand einer einfach verketteten Liste die elementaren Listenoperationen demonstriert.

Das Programm soll menuegesteuert ablaufen. Die Liste soll als Infoteil Namen enthalten (char info[20], vergleichen mit Hilfe der Funktion strcmp(..) aus der Bibliothek <string.h>). Sie soll alphabetisch aufsteigend sortiert aufgebaut werden.

Das Programm soll dem Benutzer folgende Operationen zur Verfügung stellen:

1. Einfügen eines Datensatzes
2. Löschen eines Datensatzes
3. Ausgabe der gesamten Liste auf den Bildschirm

Bäume

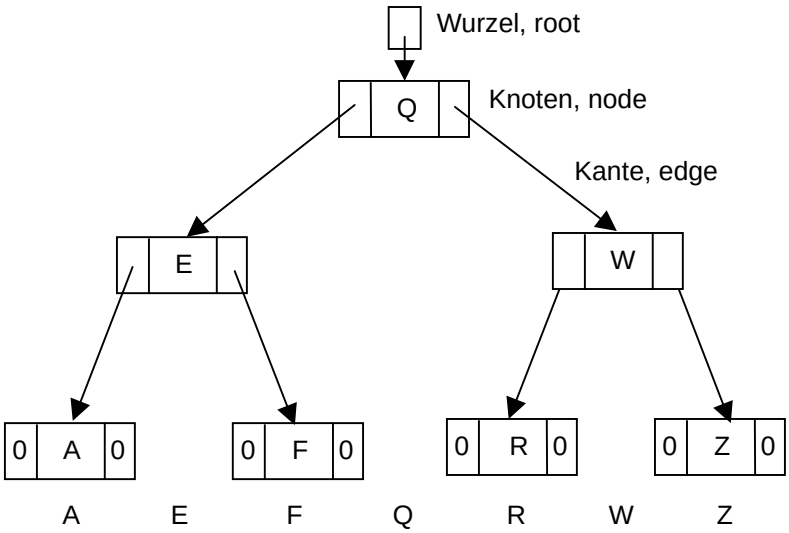
Eine weitere Form der dynamischen Datenstrukturen sind Bäume, insbesondere werden hier die sogenannten binären Bäume eingesetzt.

Die Struktur eines Elementes dieses Types könnte wie folgt aussehen:

Beispiel

```
struct knoten
{
  struct knoten *ln;
  struct knoten *rn;
  int           info;
}
```

Ein Baum könnte dann wie folgt aussehen. Die Verweise sind durch die Kanten abgebildet:



Vorteil der binären Bäume ist, daß das Suchen einzelner Elemente wesentlich schneller ist, als bei Listen. Man nennt sie deshalb auch Suchbäume.

Aufgabe 30

Es soll ein Programm erstellt werden, das die elementaren Baumoperationen anhand eines binären Suchbaumes demonstriert. Ein Knoten soll als Infoteil Namen enthalten. Der Baum soll alphabetisch sortiert sein.

Das Programm soll folgende Operationen dem Benutzer zur Verfügung stellen:

1. Einfügen eines Datensatzes
2. Löschen eines Datensatzes
3. Anzeigen des Baumes
4. Suchen eines Elementes im Baum

12 Bitfelder

Bitfelder sind eine spezielle Art von Strukturen. Bitfelder bieten die Möglichkeit die Bits eines Speicherwortes gezielt zu setzen oder abzufragen. Diese Möglichkeiten können bei der Ansteuerung der Rechnerhardware oder der Peripheriebausteine (z.B. beim Einsatz als Steuerungsrechner) genutzt werden.

Wichtig ist dabei, daß ein Speicherwort mehrere Bitfelder enthalten kann. Gibt es mehr Bitfelder als ein Speicherwort aufnehmen kann, so werden zusätzliche Speicherworte eingesetzt. Ein Bitfeld jedoch kann nie auf mehrere Speicherworte aufgeteilt sein. Durch diese Vereinbarungen bedingt, können zwischen Bitfeldern Lücken entstehen.

Die Festlegung der Bitfelder hängt sehr stark von der verwendeten Maschine ab, da die Wortbreiten auf den unterschiedlichen Maschinen immer anders sind. Dies führt natürlich zu Portabilitätsproblemen. Dies ist aber in nicht so tragisch, da die Bitfelder in der Regel in hardwareabhängigen Routinen eingesetzt werden, die ohnehin nicht portierbar sind.

Speziell in MS-'C' gelten folgende Regeln:

Als Typen für Bitfelder sind zugelassen:

- (unsigned) char
- (unsigned) int
- (unsigned) long

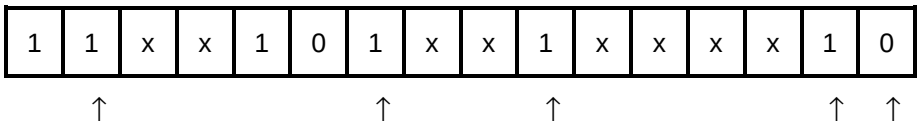
Hinweis

Die Bitfelder werden immer als unsigned interpretiert. Die Deklaration als signed (char, int oder long) ist zwar syntaktisch zulässig, wird jedoch ignoriert.

Mit Hilfe der Typdeklaration läßt sich bei den PC's und MS-'C' bestimmen, welche Wortbreite zur Ausrichtung der Bitfelder benutzt wird. Die Ausrichtung der Bits beginnt bei MS-'C' von rechts. Mit Hilfe eines Bitfeldes der Breite 0 läßt sich eine Ausrichtung des nächsten Feldes an der nächsten Wortgrenze erzwingen.

Beispiel

Annahme: 16-Bit Speicherwort



In diesem Speicherwort gibt es von rechts beginnend 5 Bitfelder unterschiedlicher Breite. Der Beginn der Bitfelder ist jeweils durch einen Pfeil gekennzeichnet.

Um die dargestellte Bitkonstellation herbeizuführen wäre die folgende Deklaration mit Intialisierung notwendig.

Beispiel

```
struct
{
  unsigned  f1 : 1;
  unsigned  f2 : 1;
               : 4;  /* 4 unbenutzte Bits */
  unsigned  f3 : 1;
               : 2;  /* 2 unbenutzte Bits */
  unsigned  f4 : 3;
               : 2;  /* 2 unbenutzte Bits */
  unsigned  f5 : 2;
} status = { 0, 1, 1, 5, 3 };
```

Mit der Zuweisung:

```
status.f4 = 3;
```

kann erreicht werden, daß im Bitfeld f4 der Inhalt 101 auf 011 geändert wird.

Hätte man die Zuweisung:

```
status.f4 = 38;
```

benutzt, dann wäre f4 auf 110 gesetzt worden, da wegen der Breite des Bitfeldes von 3 Bit der zugewiesene Wert modulo 8 interpretiert wird.

Aufgabe 31

Definieren Sie sich eine Struktur, die den Bildschirmspeicher repräsentiert. Anschließend vereinbaren Sie einen Zeiger auf diese Struktur und initialisieren diesen mit der Startadresse Ihres Bildschirmspeichers.

Schreiben Sie nun ein Programm, das es Ihnen ermöglicht verschiedene Werte für die Bildschirmattribute einzugeben und anschließend den Bildschirm entsprechend ausfüllt.

13 Unions

Unions sind ebenfalls spezielle Strukturen. Sie enthalten nur eine Komponente für deren Typ es allerdings eine beliebig große Auswahl gibt. Die Ausdehnung eines Unions richtet sich nach dem Typ mit der größten Ausdehnung. Die Unions entsprechen den varianten Records in Pascal. Die Syntax der Union-Deklaration entspricht der Syntax bei den Strukturen.

Beispiel

```
union
{
  struct
  {
    double  re;
    double  im;
  }         comp;
  double    real;
} r,u,i;

main ()
{
  r.real = 50;
  r.comp.im = 20;
  printf( "\n%f\n%f\n", r.comp.re, r.comp.im );
}
```

Die Ausgabe dieses Programmes würde wie folgt aussehen:

```
50.000000
20.000000
```

In diesem Beispiel wurde der Spezialfall vorgestellt, daß sich die verschiedenen Typen teilweise decken. Dies muß aber nicht der Fall sein. Vielmehr kann der für das Union-Datenobjekt belegte Speicherbereich ganz unterschiedlich belegt und interpretiert werden.

Beispiel

```
union test
{
  char  s[20];
  int   i;
  float f;
} a;
```

Zulässig sind dann folgende Anweisungen:

```
a.f = 3.14;
a.i = 45;
strcpy ( a.s, "Werner" );
```

13.1 Typdeklaration mit typedef

Die typedef-Deklaration definiert eine Typangabe für einen Typ. Man kann sie verwenden, um kürzere und aussagefähigere Namen für einen Typ zu erhalten. Gerade beim Einsatz von Strukturen empfiehlt sich die Anwendung der typedef Deklaration, da hierdurch Programme leichter lesbar werden und obendrein der Schreibaufwand verringert wird.

Allgemeine Form:

```
typedef datentyp typname;
```

Wenn im Programm die Angabe des Datentypes erforderlich ist, kann hierfür in Zukunft der Typname verwendet werden.

Beispiel

```
typedef double SEHRGENAU;

typedef struct knoten
{
  struct knoten *ln;
  struct knoten *rn;
  char          key[20];
} KNOTEN;

SEHRGENAU addiere( SEHR_GENAU a, SEHR_GENAU b )
{
  ...
}

KNOTEN tinsert ( KNOTEN **root, KNOTEN *new )
{
  KNOTEN hilf;
  ...
}
```

Einer 'C'-Konvention gemäß werden neben den selbstdefinierten Konstanten auch die selbstdefinierten Datentypen durchgehend groß geschrieben (siehe hierzu auch K&R, 1. Ausgabe, Seite 151)

14 Dateien

Dateien sind in 'C' immer Bytefolgen. Die Bearbeitung ist grundsätzlich auf zwei Arten möglich. Diese beiden Arten werden in den folgenden Abschnitten vorgestellt.

14.1 Mit Hilfe der Systemaufrufe

Zu den wichtigsten Systemaufrufen gehören die folgenden Funktionen:

- `fd = creat  ( name, attrib );`
- `fd = open   ( name, mode );`
- `n  = read   ( fd, buffer, n_bytes );`
- `n  = write  ( fd, buffer, n_bytes );`
- `n  = close  ( fd );`
- `n  = lseek  ( fd, offset, origin );`
- `n  = unlink ( name );`

name Der Parameter name ist eine Zeichenkette, die den Dateinamen inklusive der Pfadbezeichnung angibt. Beachten Sie, daß Sie bei der Angabe von Pfaden z. B. c:\windows\test.dat, den Backslash jeweils verdoppeln müssen, da sonst das nachfolgende Zeichen als Steuerzeichen interpretiert wird. Für das Beispiel ist also die Zeichenkette `"c:\\windows\\test.dat"` anzugeben.

fd Der Parameter fd ist der sogenannte File-Descriptor (handle). fd hat den Typ unsigned. fd ist quasi der Index in einer Tabelle, die die geöffneten Dateien beschreibt. Der maximale Wert von fd hängt bei MS-DOS-Systemen davon ab, welche Anzahl für gleichzeitig offene Dateien in der Konfigurationsdatei des DOS (CONFIG.SYS) angegeben wurde. Ist dort z.B. FILES = 20 angegeben, so ist der Wertebereich für fd: $0 \leq fd < 20$.

attrib Mit diesem Parameter kann die Zugriffsberechtigung beeinflußt werden. Bei MS-DOS-Systemen kann hier z. B. bestimmt werden, ob das Read-Only-Attribut gesetzt werden soll:

Übliche Angaben für attrib sind:

- `S_IWRITE` ⇒ Schreiben erlaubt
- `s_IREAD` ⇒ Lesen erlaubt
- `S_IWRITE | S_IREAD` ⇒ Lesen und Schreiben erlaubt

Diese Konstanten sind in der Bibliotheksdatei <sys\stat.h> definiert. Zum Einbinden dieser Datei ist ein vorheriges Einbinden der Datei <sys\types.h> notwendig.

mode Der Parameter mode bestimmt für welche Operationen, die Datei geöffnet werden soll. Dabei bedeuten:

- 0 = Lesen Konstante `O_RDONLY`
- 1 = Schreiben Konstante `O_WRONLY`
- 2 = Lesen und Schreiben Konstante `O_RDWR`

Diese Angaben schließen sich gegenseitig aus. Weitere mögliche Einstellungen wie etwa O_APPEND, O_BINARY und O_CREAT, ermöglichen das Anhängen von Daten, das Behandeln der Datei im Binärmodus und das Erzeugen neuer Dateien. Die vorgestellten Konstanten sind in der Datei <fcntl.h> definiert.

Beschreibung der Systemaufrufe

creat

Die Funktion creat erzeugt eine neue Datei bzw. öffnet eine bestehende Datei zum Schreiben. Mißlingt das Erzeugen oder Öffnen, so wird als Funktionswert EOF (-1) zurückgegeben. Ansonsten wird der Filedescriptor (handle) zurückgegeben. Mit diesem Handle wird den nachfolgenden Lese- und Schreiboperationen angegeben, welche Datei für die Ausgabe bzw. Eingabe benutzt werden soll. Für den Fehlerfall wird die globale Variable errno auf einen der folgenden Werte gesetzt:

- `EACCES` Verzeichnis oder nur zum Lesen freigegebene Datei
- `EMFILE` kein weiteres Handle frei (zu viele offene Dateien)
- `ENOENT` Pfad nicht gefunden

open

Die Funktion open öffnet die durch name angegebene Datei, je nach Angabe von mode zum Lesen und/oder Schreiben. Gelingt das Öffnen der Datei wird der File-Deskriptor zurückgeliefert, ansonsten wird der Funktionswert EOF (-1) zurückgegeben und die globale Variable errno entsprechend dem Fehler gesetzt.

write

Die Funktion write dient zum Schreiben von n-bytes in die Datei. Die Daten müssen in einem Puffer bereitgestellt werden.

Beispiel

```
write ( 1, "TEST\n", 5 );
```

Bei Programmstart sind die Handles fd = 0, 1, 2 bereits vergeben.

- 0 ⇒ Standardeingabedatei
- 1 ⇒ Standardausgabedatei
- 2 ⇒ Fehlerausgabedatei

Diese drei Dateien sind bei MS-DOS-Systemen standardmäßig der Datei CON also der Tastatur bzw. dem Bildschirm zugeordnet.

Daher werden durch den im Beispiel angegebenen Aufruf fünf Byte des durch die Zeichenkettenkonstante "Test\n" referenzierten Speicherbereiches auf den Bildschirm ausgegeben.

Als Funktionswert liefert die Funktion write die Anzahl der geschriebenen Bytes zurück. Der Funktionswert -1 zeigt einen Fehler an, die Variable errno zeigt an, welcher Fehler aufgetreten ist:

- `EBADF` ⇒ ungültiges handle, Datei nicht geöffnet
- `ENOSPC` ⇒ nicht genügend freier Speicher auf dem Ausgabegerät

read

Die Funktion read dient zum Übertragen von n bytes aus der Datei in einen Puffer. Wichtig ist, daß der Puffer ausreichend dimensioniert ist. Durch die Lesefunktion wird kein Speicherplatz bereitgestellt. Der Funktionswert zeigt an, wieviele Bytes tatsächlich gelesen wurden. Der Funktionswert -1 zeigt den Fehlerfall an. Hier nur ungültiges Handle möglich (EBADF).

close

Die Funktion close schließt die mit dem File-Descriptor angegebene Datei. Der Funktionswert ist gleich 0, wenn dies erfolgreich abgeschlossen werden konnte. Der Funktionswert -1 zeigt an, daß ein ungültiger File-Deskriptor angegebenen wurde (errno=EBADF).

Sie können sich eine Datei als eine Reihe von Bytes vorstellen. In der Datei zeigt der sogenannte Schreib-/Lesezeiger immer auf das nächste zu lesende Byte oder die nächste zu beschreibende Position. Durch die Funktionen read und write wird dieser Zeiger immer um die Anzahl der geschriebenen bzw. gelesenen Bytes weiter gesetzt.

lseek

Mit Hilfe der Funktion lseek (seek) haben Sie die Möglichkeit, den Schreib-/Lesezeiger wahlfrei (random) in einer Datei zu positionieren. Mit origin wird der Ausgangspunkt angegeben und mit offset die Anzahl der Bytes um die der Zeiger bezüglich des Ausgangspunktes versetzt werden soll. Dabei kann für origin eine der folgenden Konstanten benutzt werden:

- `SEEK_SET` ⇒ Dateianfang
- `SEEK_CUR` ⇒ Aktuelle Position des Dateizeigers
- `SEEK_END` ⇒ Dateiende

Der Dateizeiger läßt sich auch hinter das Dateiende positionieren, dies ist z. B. erforderlich, wenn Daten der Datei zugefügt werden sollen. (Übrigens wurde eine Datei mit dem mode O_APPEND geöffnet, so wird der Dateizeiger automatisch vor jedem Schreibzugriff hinter das Dateiende gesetzt.) Ein Versuch den Dateizeiger vor den Dateianfang zu positionieren, führt zu einem Fehler.

Hinweis

Wichtig ist, daß offset den Typ long hat. Mit

```
lseek ( fd, 0L, SEEK_SET );
```

wird der Dateizeiger beispielsweise auf den Dateianfang gesetzt.
Mit dem Aufruf:

```
pos = lseek ( fd, 0L, SEEK_CUR );
```

läßt sich die aktuelle Position des Dateizeigers bestimmen. Diese könnte dann beispielsweise beim Erstellen einer Datei für jeden Datensatz abgespeichert werden und später für den gezielten Zugriff auf die einzelnen Datensätze eingesetzt werden. Die Funktion liefert als Funktionswert also die jeweilige Position des Dateizeigers bezogen auf den Dateianfang. Wird als Funktionswert -1L zurückgegeben, dann ist ein Fehler aufgetreten. errno kann dann einen der beiden folgende Werte annehmen:

- `EBADF` ⇒ ungültiges Handle (Datei nicht offen)
- `EINVAL` ⇒ ungültiger Wert für Ursprung, oder neue Position vor Dateianfang

unlink

Die Funktion unlink dient zum Löschen der mit name angegebenen Datei. Als Funktionswert wird 0 zurückgegeben, wenn das Löschen erfolgreich war, ansonsten wird der Funktionswert -1 zurückgegeben und errno zeigt an, welcher Fehler vorliegt.

- `EACCES` ⇒ name bezeichnet ein Verzeichnis oder Datei, die nur zum Lesen freigegeben wurde.
- `ENOENT` ⇒ Datei oder Pfad nicht gefunden

Nachfolgend soll Ihnen ein Beispiel für die Benutzung der Systemaufrufe gegeben werden. Das Programm schreibt MAX Werte vom Datentyp char in eine Datei mit dem Namen DATEN.DAT. Anschließend werden die Werte wieder aus der Datei ausgelesen und auf dem Bildschirm ausgegeben. Zuletzt wird die Datei wieder gelöscht.

Beispiel

```
#include <stdio.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#define MAX 10
#define LENGTH 20
char a[MAX][LENGTH], b[MAX][LENGTH];

main () {
  unsigned fd, i;
  /* Erzeugen einer Datei und */
  /* Freigabe zum Lesen und Schreiben */
  fd = creat ( "DATEN.DAT", S_IWRITE | S_IREAD );
  for ( i=0; i<MAX; i++ ) {
    /* Einlesen der Daten */
    printf ( "%2d. Name: ", i+1 );
    gets ( a[i] );
    /* und Abspeichern in der geöffneten Datei */
    write ( fd, a[i], LENGTH );
  }
  /* Schließen der Datei */
  close ( fd );
  /* Öffnen der Datei zum Lesen und Schreiben */
  fd = open ( "DATEN.DAT", 2 );
  for ( i=0; i<MAX; i++ ) {
    /* Lesen aus der geöffneten Datei */
    read ( fd, b[i], LENGTH );
    /* Ausgabe des gelesenen Wertes */
    printf ( "%2d. %s", b[i] );
  }
  /* Schließen der Datei */
  close ( fd );
  /* Löschen der Datei */
  unlink ( "DATEN.DAT" );
}
```

Aufgabe 32

Das Programm aus Aufgabe 29 (Listenverwaltung) soll so erweitert werden, daß die Möglichkeit besteht, eine lineare Liste aus einer Datei einzulesen und lineare Listen in einer beliebigen Datei zu speichern.

Dazu sollen folgende Kommandos dem Programm hinzugefügt werden:

'R' Read — Lesen einer linearen Liste aus der anzugebenden Datei. Es davon ausgegangen, daß dies Datei zuvor mit dem W-Kommando erstellt wurde.

'W' Write — Speichern einer linearen Liste in die angegebene Datei

14.2 Mit Hilfe der Standardbibliothek

Diese Möglichkeit hat den Vorteil, daß sie wesentlich komfortabler ist, als der Zugriff auf die Dateien mit Hilfe der Systemfunktionen. So können wir z. B. bei Textdateien die Funktionen zur formatierten Ein- und Ausgabe benutzen.

Einzige Bibliotheksdatei, die für die Benutzung dieser Möglichkeiten eingebunden werden muß, ist die Datei <stdio.h>. Sie sollten bei der Programmierung einer Dateibehandlung nach Möglichkeit den Funktionen der Standardbibliothek, also den im folgenden vorgestellten Funktionen, gegenüber den Systemaufrufen den Vorzug geben.

Neben den bereits in Kapitel 7 vorgestellten Funktionen, die alle durch ein vorangestelltes f auch für die Ein- und Ausgabe in Dateien eingesetzt werden können, gibt es eine Reihe weiterer Funktionen die zur speziellen Dateibehandlung eingesetzt werden können. Zum besseren Verständnis sollen aber vorweg einige zentrale Deklarationen aus der Datei stdio.h betrachtet werden.

```
#define BUFSIZ  512   /* Puffergroesse                */
#define _NFILE   20   /* max. Anzahl offener Dateien  */

typedef struct _iobuf {
 char   *_ptr;   /* Position des naechsten Zeichens   */
 int     _cnt;   /* Anzahl der restlichen Zeichen     */
 char   *_base;  /* Anfangsadresse des Puffers        */
 int     _flag;  /* Art des Zugriffes                 */
 int     _file;  /* File-Deskriptor                   */
} FILE;

extern FILE _iob[_NFILE]; /* Bereich für Strukturen   */
                          /* FCB = File Control Blocks*/

#define stdin  (&_iob[0]) /* Standardeingabe          */
#define stdout (&_iob[1]) /* Standardausgabe          */
#define stderr (&_iob[2]) /* Fehlerausgabe            */
#define stdaux (&_iob[3]) /* serielle Schnittstelle   */
#define stdprn (&_iob[4]) /* paralelle Schnittstelle  */

#define NULL 0
#define EOF  (-1)

#define getc(p) (--(p)->_cnt>=0 \
                 ? *(p)->ptr++ : _filbuf(p) )

#define getchar() getc(stdin)
```

(Siehe hierzu auch K&R, 1. Auflage, Seite 180 bzw. Listing der Datei <stdio.h>)

Beachten Sie bitte, daß alle definierten Bezeichner mit einem Unterstrich beginnen. Dies ist gemacht, um Verwechselungen mit benutzerdefinierten Bezeichnern zu verhindern. In der Struktur FILE werden alle für den Zugriff auf Dateien notwendigen Daten abgelegt.

Mit Hilfe der Makros getc bzw. getchar ist der Zugriff auf den Puffer möglich, der einen Teil der Datei enthält. Jede Anwendung von getc(p) vermindert die Anzahl der verfügbaren Zeichen im Puffer (--(p)->_cnt) um das entnommene Zeichen (*(p)->_ptr++) und rückt den Zeiger auf das nächste Zeichen. Falls der Puffer leer ist, wird er mit Hilfe der Funktion _filbuf wieder gefüllt.

Wir erkennen hier die Möglichkeit Makros zu definieren. Wichtig ist dabei, daß die Parameter des Makros in der Definition immer in Klammern gesetzt werden. Damit wird sichergestellt, daß ein eventuell als Aktualparameter angegebener Ausdruck zuerst ausgewertet wird. Analog dazu existiert auch ein Makro putc bzw. putchar.

Hinweis Weitere Informationen zur Definition von Makros finden Sie im Kapitel 16.

Folgende Funktionen stehen Ihnen für die Dateibehandlung zur Verfügung:

- `fgetc( fp )` wie getc, jedoch als Funktion
- `fscanf( fp, ... )` scanf entspricht fscanf (stdin, ...)
- `fgets( fp, ... )` wie gets
- `ungetc( fp )` setzt ein Zeichen in die Datei zurück
- `fputc( fp )` wie putc, jedoch als Funktion
- `fprintf( fp, ... )` printf entspricht fprintf (stdout, ...)

Wenn Sie mit einer externen Datei arbeiten, so muß diese zuvor geöffnet und nach Abschluß aller Operationen wieder geschlossen werden. Dazu dienen die Funktionen:

- `FILE *fopen( char *name, char *mode )`
- `int fclose( FILE *stream )`

Für name und mode müssen Zeiger auf Zeichenketten oder Zeichenkettenkonstanten übergeben werden. Dabei ist name der Dateiname ggf. mit Pfadangabe.

Für mode sind folgende Angaben möglich:

- `"r"` ⇒ Öffnen zum Lesen
- `"w"` ⇒ Öffnen zum Schreiben
- `"a"` ⇒ Öffnen zum Schreiben am Dateiende (anhängen)

Durch "r+", "w+" und "a+" können die Dateien sowohl zum Schreiben, als auch zum Lesen geöffnet werden. Wird als mode "a" gewählt wird der Dateizeiger automatisch vor jedem Schreibvorgang auf das Dateiende gesetzt. Mit den Funktionen rewind und fseek läßt sich der Dateizeiger auch an andere Stellen in der Datei positionieren.

fopen Die Funktion fopen liefert als Funktionswert den NULL-Zeiger zurück, falls aus irgendwelchen Gründen das Öffnen der Datei nicht möglich war.

Hinweis Wenn eine Datei zum Schreiben ("w") geöffnet wird, die bereits existiert, so wird die bestehende Datei überschrieben. Wird eine Datei zum Anhängen ("a") geöffnet, die noch nicht existiert, so wird sie neu erstellt.

In der Praxis sollten Sie die Funktion immer in dem folgenden Rahmen aufrufen:

```
if ( ( fp = fopen ( datname, mode ) ) != NULL )
{
  /* Dateibehandlung */
}
else
{
  /* Fehlerbehandlung */
}
```

fclose Die Funktion fclose dient zum Schließen von Dateien, sie liefert den Wert 0, falls das Schließen der Datei erfolgreich war. Als Alternative für fclose bietet sich noch fcloseall(void) an. Diese Funktion ermöglicht das Schließen aller zur Zeit offenen Dateien z. B. am Programmende.

Weitere häufig benutzte Funktionen zur Dateibehandlung sind:

fflush `int fflush( FILE *stream )`

Wenn die Datei zum Schreiben geöffnet ist, wird durch fflush der Pufferinhalt in die Datei geschrieben. Ist die Datei zum Lesen geöffnet wird der Puffer geleert. Der Funktionswert EOF zeigt einen Fehler an.

getpos `int fgetpos( FILE *stream, unsigned *pos )`

setpos `int fsetpos( FILE *stream, unsigned *pos )`

Diese Funktionen dienen zum Ermitteln der Position des Dateizeigers, bzw. zum Setzen des Dateizeigers. Mit fsetpos können nur zuvor mit fgetpos ermittelte Positionszeiger eingesetzt werden.

fileno `int fileno( FILE *stream )`

Gibt das zu diesem Eingabestrom gehörende Handle zurück. Sie können das Handle dann als File-Deskriptor beim Einsatz der Systemaufrufe benutzen.

fread `int fread( void *buf, int size, int nr, FILE *stream )`

fwrite `int fwrite( void *buf, int size, int nr, FILE *stream )`

Diese beiden Funktionen sind die Gegenstücke zu read und write. Sie ermöglichen z. B. auch das Abspeichern von Strukturen. Beachten Sie bitte, daß bei MS-'C' zwischen Textmodus und Binärmodus unterschieden wird. Im Textmodus werden bestimmte Bytes umgewandelt in andere Zeichen, sie sollten daher in der Regel den Binärmodus verwenden und nur wenn feststeht, daß es sich um Textdateien handelt den Textmodus. ANSI-'C' unterscheidet die beiden Modi nicht.

fseek `int fseek( FILE *stream, long *pos, int *origin )`

Diese Funktion ermöglicht das Positionieren des Dateizeigers, z. B. relativ zum Dateianfang (origin=SEEK_SET). Damit wird der wahlfreie Zugriff auf bestimmte Datensätze der Datei ermöglicht.

ftell `long ftell( FILE *stream )`

Diese Funktion liefert den Offset relativ zum Dateianfang. Dieser kann dann mit fseek dazu benutzt werden, um den Dateizeiger auf einem bestimmten Datensatz zu positionieren.

fstat `int fstat( unsigned fd, struct stat *buf )`

Diese Funktion liefert in der Struktur stat, die Daten über die Datei, die vom jeweiligen Betriebssystem im Verzeichnis festgehalten werden, z. B. Größe der Datei, Datum und Uhrzeit der letzten Änderung und Dateiattribute.

Übrigens unter den meisten Betriebssystemen lassen sich auch die angeschlossenen Geräte als Dateien behandeln. So sind bei der MS-DOS-Version von 'C' standardmäßig bereits die Dateien stdprn für die paralle Schnitt- stelle (Drucker) und stdaux für die serielle Schnittstelle (Plotter oder Modem) definiert.

Aufgabe 33 Das Programm aus Aufgabe 30 (Baumverwaltung) soll so erweitert werden, daß man folgende weiteren Menuepunkte zur Verfügung hat:

1. Ausgabe in eine beliebige Textdatei (also auch Drucker) die Ausgabe soll alphabetisch sortiert erfolgen (Option 'D', wie Drucken).
2. Ausgabe in eine Datei zum späteren Wiedereinlesen des Baumes (Option 'W', wie Write).
3. Einlesen eines Baumes, der mit der unter 2. beschriebenen Option abgespeichert wurde. Ein ggf. im Speicher existierender Baum soll zuvor gelöscht werden (Option 'R', wie Read).

15 Kommandozeilenparameter

Von den DOS-Kommandos her ist Ihnen sicherlich die Möglichkeit bekannt, in der Kommandozeile Parameter an die Kommandos zu übergeben, wie z. B. den Namen der zu bearbeitenden Datei.

Diese Möglichkeit besteht auch bei selbstgeschriebenen 'C'-Programmen. Dies können Sie dadurch realisieren, daß Sie für die Funktion main eine Parameterschnittstelle programmieren. Dabei gibt der erste Parameter immer die Anzahl der Zeichenketten in der Kommandozeile an. Der zweite Parameter stellt einen Zeiger auf einen Vektor von Zeichenketten bereit. Diese Zeichenketten enthalten dann die in der Kommandozeile angegebenen Kommadozeilenparameter. Üblicherweise werden die beiden Parameter der Funktion main mit argc und argv bezeichnet.

Beispiel

```
#include <stdio.h>

main ( int argc, char *argv[] )
{
  while ( --argc )
    printf ( "%s", *++argv );
}
```

Dieses Programm wiederholt lediglich seine Eingabeparameter auf der Standardausgabe. Wir nehmen an, daß das Programm als ausführbares Programm unter MS-DOS mit dem Namen REPEAT.EXE vorliegt.

Dann ergibt Aufruf:

```
C:\>repeat C-Programmieren ist schön.
```

die Ausgabe:

```
C-Programmieren ist schön.
```

Die int-Variable argc hat in diesem Beispiel nach dem Aufruf des Programmes den Wert 4, da 4 Zeichenketten in der Kommandozeile angegeben sind. Der Zeiger argv zeigt auf einen Vektor von Zeichenketten (übrigens wäre die Deklaration char **argv auch richtig), der wie folgt aussieht:

argv	→	argv[0]	→	"repeat"
		argv[1]	→	"C-Programmieren"
		argv[2]	→	"ist"
		argv[3]	→	"schön."
		argv[4]	→	== NULL

Das Vorbesetzen der Variablen argc und argv ist Aufgabe einer Startprozedur, die durch die Laufzeitbibliothek automatisch ausgeführt wird. Die Auswertung der Kommandozeilenparameter ist dagegen Aufgabe der Funktion main. Häufig finden Sie in diesem Zusammenhang eine der folgenden oder ähnliche Formulierungen:

```
while ( --argc && .... )
```

oder

```
while ( *++argv && .... )
```

In DOS- und UNIX-Systemen ist es üblich, daß man die Ausführung der Kommandos bzw. Programme mit Hilfe sogenannter Schalter beeinflussen kann. Als Erkennungszeichen für einen Schalter wird bei DOS das Zeichen "/" und bei UNIX das Minus-Zeichen eingesetzt.

Beispiel

Die allgemeine Form eines fiktiven UNIX-Kommando könnte wie folgt aussehen:

```
cxy [-n] [-v]
```

Damit sind folgende Angaben für die Schalter zulässig;

- -n
- -v
- -nv
- -vn
- kein Schalter

Beispiel

Die Auswertung der Schalter kann nun z. B. mit folgendem Programmausschnitt vorgenommen werden:

```
...
n_vorhanden = 0;
v_vorhanden = 0;
errorflag   = 0;
while ( --argc && (*++argv)[0] == '-' )
  for ( ptr=argv[0]+1; *ptr != '\0'; ++ptr )
    switch ( *ptr )
    {
      case 'n' : n_vorhanden = 1;
                 break;
      case 'v' : v_vorhanden = 1;
                 break;
      default  : errorflag = 1;
    }
...
```

Durch die Auswertung der in dieser Routine gesetzten Flags kann das nachfolgende Programm in seiner Ausführung nun beeinflußt werden. Durch --argc in der while-Bedingung ist gewährleistet, daß wenn die Kommandozeile nur den Programmnamen enthält, die while-Schleife nicht zur Ausführung gelangt.

Die Möglichkeit des Einsatzes von Kommandozeilenparametern ist insbesondere bei vielen kleineren Programmen eine sehr nützliche Einrichtung. Sie ermöglichen es den "Benutzerdialog" in der Kommandozeile zu führen und eignen sich deshalb besonders für den Einsatz in Stapeldateien oder in Shellskripten (UNIX).

Aufgabe 34

Schreiben Sie ein 'C'-Programm, das es ermöglicht 'C'-Listings von MS-Quick-C (Turbo-C) auszudrucken.

Die allgemeine Form dieses Kommandos soll wie folgt aussehen:

```
clist source [dest] [-n] [-k] [-rnn] [-tnn] [-h|?]
```

Dabei bedeuten:

- `source` Name der Eingabedatei, falls kein Name angegeben wird, soll eine Fehlermeldung ausgegeben werden.
- `dest` Name der Ausgabedatei, falls kein Name angegeben wird, soll die Standardausgabedatei benutzt werden.
- `-n` Mit diesem Parameter wird angezeigt, daß die standardmäßige Zeilennummerierung ausgeschaltet wird.
- `-k` Mit diesem Parameter soll angezeigt werden, daß die standardmäßige Ausgabe eines Seitenkopfes entfallen kann.
- `-rnn` Mit diesem Parameter soll angezeigt werden, wieviele Leerzeichen jeder Zeile als linker Rand hinzugefügt werden sollen. nn hat standardmäßig den Wert 10.
- `-tnn` Mit diesem Parameter soll angezeigt werden, durch wieviele Leerzeichen die Tabulatorzeichen ersetzt werden sollen. nn hat standardmäßig den Wert 08.
- `-h, -?` Mit diesem Parameter soll der Benutzer eine Hilfe anfordern können. Bedenken Sie, daß für diesen Fall keine Sourceangabe notwendig ist.

Es soll auch möglich sein, die Parameter gruppenweise anzugeben.

Beispiel

```
-nkr00
```

oder

```
-t00kr08
```

16 Speicherklassen

Variablen und Funktionen können in 'C' bei ihrer Vereinbarung mit zusätzlichen Attributen versehen werden. Diese Attribute bestimmen, wo diese Größen bzw. Funktionen gespeichert werden, dadurch werden indirekt auch die Lebensdauer und die Sichtbarkeit der Variablen und Funktionen mitbestimmt.

Grundsätzlich unterscheidet man Speicherklassen für Variablen:

- register
- auto
- static
- extern

und für Funktionen:

- static
- extern

Die Angabe der Speicherklasse erfolgt bei der Deklaration der Variablen oder der Funktion vor der Typangabe. Dazu einige Beispiele:

Beispiel

```
static char buffer[1000];
extern double a;
auto unsigned k;
register u;

extern char *strupper( char *s );
static int count( int a, double b );
```

Die Angabe der Speicherklasse ist genau wie die Typangabe nicht unbedingt vorgeschrieben. Unterbleibt die Angabe der Speicherklasse werden Voreinstellungen angenommen. Während bei einer fehlenden Typangabe jedoch grundsätzlich int als Datentyp angenommen wird, hängt die Voreinstellung für die Speicherklasse vom Kontext (innerhalb oder außerhalb einer Funktion) und von der Art des deklarierten Objektes (Funktion oder Variable) ab.

16.1 Variablen

Im Prinzip gibt es bei den Variablen drei Möglichkeiten für die Speicherung, nämlich:

- im Datenbereich des Programmes,
- im Stackbereich des Programmes oder
- in einem Prozessorregister.

Wo die Variablen letztlich gespeichert werden, das können Sie mit Hilfe der Speicherklassen beeinflussen. Außerdem können Sie mit der Speicherklasse anzeigen, welchen Gültigkeitsbereich und welche Lebensdauer die Variablen haben sollen.

Wir müssen unterscheiden zwischen globalen und lokalen Definitionen:

Global

Global sind Definitionen außerhalb von Funktionen. Hier dürfen die Speicherklassen auto und register nicht verwendet werden. Wird keine Speicherklasse angegeben, so gilt automatisch für global definierte Variablen die Speicherklasse extern. Global deklarierte Variablen werden im Datenbereich des Programmes angelegt und leben während der gesamten Laufzeit des Programmes.

Die Speicherklassen haben dabei folgende Bedeutung:

- **extern**
  Die Variable ist in allen Modulen des Programmes erreichbar, sofern sie in dem jeweiligen Modul ebenfalls mit der Speicherklasse extern und dem gleichen Namen deklariert wurde. Die Variable darf in genau einem Modul initialisiert werden. Wenn sie nicht initialisiert wird, wird sie beim Linken automatisch mit 0 initialisiert. Stimmen bei mehreren externen Deklarationen von Variablen mit dem gleichen Namen die Typen nicht überein, so wird soviel Speicherplatz reserviert, wie für die Speicherung des Types mit dem größten Speicherbedarf notwendig ist. Mit extern hat man auch die Möglichkeit einen Vorwärtsverweis auf eine Variablen-Deklaration in derselben Datei zu schaffen, wenn diese erst nach der ersten Verwendung der Variablen erfolgt. In einem Modul kann eine Variable also mehrfach deklariert werden. Alle Deklarationen beziehen aber auf denselben Speicherbereich.
- **static**
  Mit dieser Speicherklasse hat man die Möglichkeit eine Variable nur für das Modul gültig zu machen, in dem sie deklariert wurde. Eine so definierte Variable gilt dann ausschließlich in dem Modul in dem sie deklariert wurde. Wird in einem anderen Modul eine Variable mit dem gleichen Namen als static deklariert, so wird für diese Variable ein neuer Speicherbereich angefordert. In verschiedenen Modulen können Variablen mit dem gleichen Namen aber unterschiedlichen Speicherklassen deklariert werden. Für alle als static definierten Variablen, wird jeweils ein neuer Speicherbereich zur Verfügung gestellt, alle extern deklarierten Variablen beziehen sich auf dasselbe Datenobjekt. Die Verwendung eines Namens mit unterschiedlichen Speicherklassen in einem Modul ist jedoch nicht erlaubt.

Lokal

Lokal sind Definitionen innerhalb von Funktionen. In den lokalen Deklarationen dürfen alle Speicherklassen verwendet werden. Der Ort der Speicherung der Variablen hängt von der jeweiligen Speicherklasse ab. Wird keine Speicherklasse angegeben, so gilt automatisch die Speicherklasse auto. Die Lebensdauer der Variablen hängt ebenfalls von der verwendeten Speicherklasse ab. Während der Gültigkeitsbereich immer der Block und seine inneren Blöcke ist, sofern nicht in einem inneren Block eine Variable mit dem gleichen Namen deklariert wird.

Die Speicherklassen haben dabei folgende Bedeutung:

- **register**
  Diese Speicherklasse zeigt an, das der Programmierer den Wunsch hat, diese Variable in einem Register abzulegen. Dies kann sinnvoll sein, wenn diese Variable sehr häufig benutzt wird bzw. wenn es auf Geschwindigkeit ankommt. Die auf dem Markt erhältlichen Rechner unterscheiden sich jedoch erheblich in der Anzahl und Breite der Prozessorregister, sodaß der Programmierer zu prüfen hätte, ob die deklarierte Größe sich in einem Register abspeichern läßt. Außerdem geht dann unter Umständen die Portierbarkeit der Programme verloren. Deshalb wird die Entscheidung, ob diese Variable in einem Register oder im Stackbereich des Programmes abgespeichert wird, dem Compiler überlassen. Man kann also mit der Speicherklasse register nur einen "Wunsch" äußern. Hat man diesen Wunsch geäußert, dann ergibt sich folgende Konsequenz. Wenn eine Größe in einem Register gespeichert ist, kann man nicht mehr mit Hilfe eines Zeigers auf diese Größe zugreifen. Nach den Deklarationen register int i; int *p; wäre folgender Ausdruck nicht mehr erlaubt: p = &i; Eine mit register deklarierte Variable lebt nur während der Laufzeit des Blockes in dem sie deklariert wurde.
- **auto**
  Diese Speicherklasse bewirkt, daß die entsprechende Variable im Stackbereich des Programmes angelegt wird. Sie lebt nur während der Laufzeit des Blockes in dem sie deklariert wurde. Insbesondere bei Variablen mit hohem Speicherbedarf (z. B. Vektoren) und rekursiven Funktionen ist bei der Verwendung der Speicherklasse auto Vorsicht geboten, da der Stackbereich eines Programmes in der Regel begrenzt ist, und bei jedem erneuten Aufruf der Funktion erneut Speicherbereich im Stack reserviert wird (Der Stackbereich läßt sich bei Notwendigkeit ggf. über eine Compileroption erweitern).
- **static**
  Diese Speicherklasse bewirkt, daß die entsprechende Variable im Datenbereich des Programmes angelegt wird. Sie lebt auch noch nach Ablauf der Funktion. In einer so deklarierten Variablen kann man Werte speichern, die bei einem erneuten Aufrufen der Funktion wieder zur Verfügung stehen. Es ist also quasi eine globale Variable, die aber nur innerhalb des Blockes gilt, in dem sie deklariert wurde.
- **extern**
  Diese Speicherklasse ermöglicht den blocklokalen Zugriff auf externe Datenobjekte. Wird diese Speicherklasse verwendet, so braucht in einem Modul die entsprechende Variable nicht global als extern deklariert zu werden.

16.2 Funktionen

Funktionen sind immer modulglobal, sie können immer von allen anderen Funktionen innerhalb eines Moduls aufgerufen werden. Ausnahme ist die Funktion main, die nur vom Betriebssystem aus aufgerufen werden kann. In allen zu einem Programm gehörenden Modulen darf die Funktion main exakt nur einmal auftreten, jedoch muß sie mindestens einmal auftreten.

Für die Funktionen gibt es nur die Speicherklassen extern und static. Wird keine Speicherklasse angegeben, so gilt eine Funktion automatisch als extern. Die Funktionen können somit in andere Module exportiert werden.

Wir kennen die Deklaration externer Funktionen (nämlich die der Bibliotheksfunktionen) bereits als Funktionsprototypen. Durch die Funktionsprototypen, werden in unseren Modulen, die entsprechenden Funktionen bekannt gemacht. Die '.h'-Dateien enthalten zum größten Teil nichts anderes, als die Deklaration externer Funktionen und Variablen.

Der Import externer Funktion ist sowohl global als auch lokal möglich. Durch das lokale Importieren einer Funktion wird erreicht, das diese Funktion nur in dem entsprechenden Block bekannt ist, während die global importierten Funktionen im gesamten Modul benutzt werden können.

Mit der Speicherklasse static können wir erreichen, daß eine Funktion nur in dem Modul gilt, in dem sie deklariert wurde. Wir können also in einem Modul 'innere' Funktionen nach außen verstecken.

16.3 Modularisieren eines Programmes

Bei einem größeren Programmprojekt besteht die Möglichkeit ein Programm in mehrere Module aufzuteilen.

Diese bietet verschiedene Vorteile:

- logisch zusammenhängende Programmteile werden zusammengefaßt, die Problemstellung wird in Teilaufgaben zerlegt. Es ergibt sich insgesamt ein besserer Überblick.
- Es werden größere Programme möglich. Bei den MS-DOS-Rechner ist die maximale Größe des Codebereiches auf 64 KB beschränkt. Durch Aufteilung eines Programmes in Module können die Programme größer werden, da nun für jedes Modul 64 KB zur Verfügung stehen. Außerdem können mit einem entsprechenden Linker modulweise Programmteile entstehen, die sich nur dann im Speicher befinden, wenn sie benötigt (Overlays) werden. Man kann im Prinzip Programme erzeugen, die größer als der maximal verfügbare Speicher sind.
- Bei der Erstellung der Programme brauchen einmal ausgetestete und als korrekt erkannte Module nicht jedesmal neu übersetzt zu werden. Sie werden nur noch durch den Linker angebunden (siehe auch die Bibliotheken). Hierzu stellt eine 'C'-Compiler mit dem Hilfsprogramm make ein wichtiges Werkzeug zur Verfügung. Eine entsprechende Option ist auch in den modernen Entwicklungsumgebungen vorhanden (Stichwort: Projekte). Mit Hilfe der make-Option kann man festlegen, welche Module zu einem Programm gehören.

Die Möglichkeiten der Modularisierung sollen anhand eines Beispieles gezeigt werden:

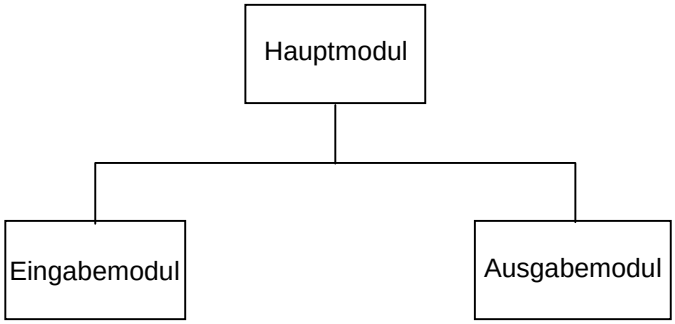
```
/* Modul A    Dateiname: ma.c ************************/

#include <stdio.h>

int va1;    /* Dies ist eine externe globale Variable */
static va2 = 1;  /* Dies Variable gilt nur im Modul A */

int fb2 ();/* Funktionsprototyp, um eine externe Funk-*/
           /* tion, die in diesem Modul bekannt wird  */

/* Hauptprogramm */
main ()
{
  extern int vb1;                /* Externe Variable, */
                                /*die nur in main gilt*/
  printf ( "%d", fb2() + vb1 +va2 );
                         /* Was wird wohl ausgegeben ? */
}
/* Ende des Hauptprogrammes */

int va1 = 3;       /* Initialisierung der globale und */
                   /* externen Variablen va1          */

int fa1 ()
{
  extern int fb1 ();    /* Ext. Fkt., gilt nur in fa1 */
  return ( fb1 () );
}


/* Modul B    Dateiname: mb.c ************************/

int va1;          /* Externe und globale Variable va1 */
extern int vb1 = 2;  /* Externe und globale Variable, */
                     /* die direkt initialisiert wird */
static int fb3 ();   /* Global, gilt nur in Modul B   */

int fb1 ()
{
  int vb2 = 4;    /* lokale Variable, gilt nur in fb1 */
  return ( fb3 () + vb2 );
}

int fb2 ()
{
  extern int fa1 ();    /* Ext. Fkt., gilt nur in fb2 */
  return ( fa1 () );
}

static int fb3 ()
{
  return ( va1 );   /* benutzt ext., globale Variable */
}
```

Bei der Erstellung der Module ist folgendes zu beachten:

- Genau eine von allen beteiligten Funktionen muß die Funktion main() sein.
- Jedes Modul kann für sich editiert und compiliert werden. Jedes Modul wird in eine eigene Datei abgelegt.
- Falls Funktionen oder Variablen in allen Modulen benutzt werden sollen, so empfiehlt sich das Anlegen einer eigenen Deklarationsdatei, nennen wir sie "def.h", die dann mit Hilfe der Include-Anweisung (#include "def.h") in jedes Modul eingebunden werden kann.

Aufgabe 35

Es soll ein Programm bestehend aus drei Modulen und einer Definitionsdatei entwickelt werden.



Im Hauptmodul soll sich nur das Hauptprogramm befinden. Im Hauptprogramm soll zuerst die Eingabefunktion aufgerufen werden und danach soll es möglich sein, die Ausgabefunktionen wahlweise, mehrfach aufzurufen.

Die Struktur eines Elementes soll wie folgt aussehen:

```
struct { char info[20]; int zaehler } FELD;
```

Das Eingabemodul soll eine Eingaberoutine enthalten, in der der Benutzer gefragt wird, wieviele Elemente eingegeben werden sollen. Danach wird der benötigte Speicherplatz mit Hilfe eines internen Unterprogrammes dynamisch bereitgestellt.

Anschließend werden die Inhalte für alle Elemente der Reihe nach abgefragt, dabei soll der Elementteil zaehler automatisch gefüllt werden.

Das Ausgabemodul soll zwei Funktionen bereitstellen. Eine zur Ausgabe eines einzelnen Listenelementes und eine zur Ausgabe der gesamten Liste

Die Aufgabe soll Sie mit der Handhabung der Entwicklungsumgebung bei mehreren Modulen vertraut machen.

17 Der Preprozessor

Der Preprozessor ist eine Art Vorübersetzer. Er untersucht den Quelltext auf das Vorkommen der mit dem Zeichen '#' beginnenden Preprozessor-Anweisungen und nimmt Auswechselungen und Einfügungen innerhalb des Quelltextes vor.

Wesentliche Möglichkeiten, die sich durch den Einsatz des Preprozessors bieten, sind:

- Definition von Konstanten und Makros
  Mit den Preprozessor-Anweisungen #define und #undef können Sie Konstanten und Makros definieren bzw. die Definiton rückgängig machen. Bei Auftreten einer so definierten Konstante oder eines Makros wird eine Textersetzung innerhalb des Quelltextes vorgenommen.
- Einfügen von Dateien in den Quelltext
  Mit der Preprozessor-Anweisung #include können Sie den Inhalt anderer Quelltextdateien in einen Quelltext einfügen. Die Preprozessor-Anweisung wird durch den Inhalt der Include-Datei ersetzt.
- Steuerung der Compilierung (Stichwort: Bedingte Compilierung)
  Mit den Preprozessor-Anweisungen #if, #if defined (), #ifdef, #ifndef, #else, #elif und #endif können Sie die Übersetzung bestimmter Stellen des Quelltextes von Bedingungen abhängig machen. Die bedingte Compilierung kann über mit der Anweisung #define definierte Konstanten gesteuert werden.
- Zeilennummerierung und Dateinamenangabe bei Fehlermeldungen
  Mit der Preprozessor-Anweisung #line können die bei Compiler-Meldungen angezeigten Dateinamen und Zeilennummern beeinflußt werden. Die Anwendung dieser Anweisung ist beim Einsatz von Programmgeneratoren sinnvoll.
- Einbringen compilerabhängiger Besonderheiten
  Mit der Preprozessor-Anweisung #pragma können Compiler-abhängige Anweisungen in den Quelltext eingebracht werden. Compiler, die diese Anweisungen nicht kennen, ignorieren die mit dem Schlüsselwort #pragma gekennzeichneten Zeilen.

17.1 Die Anweisungen #define und #undef

Die wohl am häufigsten benutzte Preprozessor-Anweisung ist die Anweisung #define. Sie ermöglicht das Definieren von Konstanten und Makros. Durch die Anweisung #undef wird eine auf diese Art gemachte Definition wieder rückgängig gemacht.

Beispiel

```
#define MAX 1000
#define MIN 1
...
printf ( "%d %d", MIN, MAX );
...
#undef MIN
...
int MIN;
MIN = MAX - ( MAX - 1 );
...
```

Aus den in diesem Beispiel vorkommenden Quelltextzeilen wird nach der Bearbeitung durch den Preprozessor folgender Quelltext entstehen:

```
printf ( "%d %d", 1, 1000 );
...
int MIN;
MIN = 1000 - ( 1000 - 1 );
...
```

Der Preprozessor nimmt also bei Konstanten eine rein textuelle Ersetzung der definierten Konstante durch den definierenden Text vor. Die Preprozessor-Anweisungen selbst werden aus dem Quelltext entfernt. Mit den Definitionen von Konstanten können also reine Textersetzungen vorgenommen werden:

Beispiel

```
#define BEGIN {
#define END }
#define PROGRAM
#define WRITE printf
#define VERSION "V 1.0 (c) System-Crash-Jonny 1990 \n"
```

Es wird immer genau das für die Konstante eingesetzt, was nach der Deklaration der Konstanten in der Zeile folgt. Reicht die Zeile für den Ersatztext nicht aus, so kann die Zeile durch das Zeichen "\" um eine oder mehrere Zeilen erweitert werden.

```
#define GLOCKE "Festgemauert in der Erden\n\
steht die Form aus Lehm gebrannt\n\
...
```

Nach diesen Definitonen von Konstanten wäre nun folgendes Programmstück in 'C' denkbar:

```
PROGRAM main ()
BEGIN
  WRITE ( VERSION );
  WRITE ( GLOCKE );
END
```

Üblicherweise werden die selbstdefinierten Konstanten dadurch gekennzeichnet, daß sie durchgehend groß geschrieben werden.

Aber nicht nur Konstanten können mit der define-Anweisung definiert werden, sondern auch Makros. Einige vordefinierte Makros, wie putc oder getc haben Sie bereits kennengelernt.

Ich möchte trotzdem nochmals kurz die allgemeine Form darstellen und auf einige Probleme bei der Verwendung von Makros hinweisen:

Allgemeine Form

```
#define Name(Parameterliste) Ersatztext
```

Innerhalb der Parameterliste können Bezeichner aufgeführt werden. Diese Bezeichner werden im Ersatztext durch die aktuellen Parameter ersetzt.

Beispiel

```
#define writeln(x) printf ( #x "\n" )
```

Ein Aufruf der folgenden Form:

```
writeln ( Dieser Text wird für x eingesetzt );
```

ergibt folgenden 'C'-Quelltext nach dem Preprozessorlauf:

```
printf ( "Dieser Text wird für x eingesetzt" "\n" );
```

Durch das #-Zeichen vor dem Parameter wird erreicht, daß bei der Ersetzung durch die aktuellen Aufrufparameter, diese in Anführungszeichen gesetzt werden.

Natürlich sind als Parameter nicht nur Zeichenketten zugelassen, sondern auch Konstanten, Variablen und Ausdrücke. Gerade im Bezug auf die Ausdrücke soll auf zwei Probleme hingewiesen werden:

Beispiel

Die Preprozessor-Anweisung:

```
#define sqr(x) x * x
```

und der Aufruf:

```
sqr ( a + b );
```

führen zu der folgenden Ersetzung:

```
a + b * a + b;
```

Offentsichtlich führt diese Ersetzung nicht zu dem gewünschten Ergebnis, nämlich dem Quadrat des Ausdruckes. Der durch die Ersetzung entstandene Ausdruck wird infolge der Rangordnung der Operatoren nicht in der gewünschten Reihenfolge ausgewertet. Dieses Problem läßt sich dadurch vermeiden, daß man im Ersatztext um die Parameter Klammern setzt:

Beispiel

```
#define sqr(x) (x) * (x)
```

führt bei dem Aufruf

```
sqr ( a + b );
```

zur folgenden Ersetzung:

```
(a + b) * (a + b)
```

Dieser Aufruf hat nun auch den gewünschten Effekt.

Ein anderes Problem stellt der folgende Aufruf dar:

```
sqr ( x++ )
```

Dieser Aufruf führt zur folgenden Ersetzung:

```
(x++) * (x++)
```

Hier liegt offensichtlich ebenfalls ein unerwünschter Nebeneffekt vor, denn die Variable x ist nach der Ausführung des Makros nicht um 1, sondern um 2 erhöht. Sie sollten also Wertzuweisungen in der aktuellen Parameterliste der Makros vermeiden, wenn Sie nicht genau wissen, daß diese nur einmal den aktuellen Parameter einsetzen.

Wann empfiehlt sich der Einsatz von Makros ?

Die Makros bieten den Vorteil, daß sie gegenüber dem Einsatz von Funktionen ein besseres Laufzeitverhalten zeigen, da ein Anlegen neuer Datenbereiche im Stack usw. nicht notwendig ist. Sie werden vielmehr zu einem Codestück im eigentlichen Programm. Hier wird nun auch der Nachteil der Makros klar, denn sie vergrößern in der Regel den Codeumfang des Programmes. Welche der beiden Möglichkeiten, Funktion oder Makro, letzlich eingesetzt wird, muß im Einzelfall entschieden werden.

17.2 Die Anweisung #include

Die Anweisung #include ermöglicht, das Einfügen immer wieder benötigter Quelltextstücke aus anderen Dateien in die aktuelle Quelltextdatei. Die Include-Dateien werden in den Quelltext regelrecht eingefügt. Eine Include-Datei kann ihrerseits wieder include-Anweisungen enthalten (Schachtelung).

Für das Einfügen der Quelltextdateien gibt es nur die eine Anweisung #include, allerdings gibt es bei den Parametern zwei Varianten:

- Angabe des Dateinamens in spitzen Klammern

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

  Durch die Benutzung der spitzen Klammern, wird der Compiler angewiesen die Datei ausgehend von einem in der Systemumgebung festgelegten Unterverzeichnis für die Include-Dateien zu suchen. Innerhalb der spitzen Klammern sind Pfadangaben erlaubt.

- Angabe des Dateinames in Anführungszeichen

  ```
  #include "def.h"
  ```

  Durch die Benutzung der Anführungszeichen, wird der Compiler angewiesen die Datei ausgehend von dem aktuell gesetzten Verzeichnis zu suchen. Innerhalb der Anführungszeichen sind ebenfalls Pfadangaben erlaubt.

Beispiel

```
#include "\bin\userlibs\trig.h"
```

17.3 Bedingte Compilierung

Zur Steuerung der Compilierung gibt es mehrere Möglichkeiten. Die beiden letzten Möglichkeiten entsprechen einem veralteten Standard, werden von den meisten Compilern aber aus Kompatibilitätsgründen noch unterstützt.

#if k_ausdruck	#if defined(name)	#ifdef name	#ifndef name
Block 1	Block 1	Block 1	Block 2
#elif k_ausdruck	#elif defined(name)	#else	#else
Block 2	Block 2	Block 2	Block 1
#elif ...	#elif ...	#endif	#endif
...	...		
#else	#else		
Block n	Block n		
#endif	#endif		

Die Ausdrücke hinter den if-Anweisungen werden genauso logisch interpretiert wie bei einer gewöhnlichen if-Anweisung in 'C'. Die einzelnen Blöcke werden nur dann übersetzt, wenn die zugehörige Bedingung in der if-Anweisung erfüllt ist. Die beiden Varianten in der dritten und der vierten Spalte haben exakt die gleiche Wirkung. Mit der bedingten Compilierung können Sie z. B. Maschinen- oder Compiler-abhängige Besonderheiten im Quelltext berücksichtigen und bei der Compilierung die wirklich zu übersetzenden Textstellen auswählen.

Beispiel

```
#define f8087
...
#ifdef f8087
  ...
#else
  ...
#endif
```

In diesem Beispiel wird durch die einleitende define-Anweisung erreicht, daß der im if-Teil stehende Quelltext compiliert wird. Würde man die define-Anweisung weglassen, würde der im Else-Teil stehende Quelltext übersetzt. Die Else- bzw. Elif-Anweisungen sind optional. Durch die elif-Anweisung wird quasi eine Mehrfachauswahl ermöglicht. Der Bezeichner k_ausdruck soll andeuten, daß hier ein konstanter Ausdruck mit den bekannten Vergleichsoperatoren erlaubt ist.

Beispiel

```
#if STEUER == 3
  ...
#elif STEUER == 2
  ...
#elif STEUER == 1
  ...
#else
  ...
#endif
```

Mit Hilfe der Definition einer Konstanten STEUER läßt sich die Compilierung steuern, die Bedingungsausdrücke sind erfüllt, wenn sie ungleich 0 sind. Eine Konstante, die nur definiert wird aber keinen Wert zugewiesen bekommt, z. B.

```
#define STEUER
```

erhält automatisch den Wert 1.

17.4 Die Anweisungen #line und #pragma

Diese beiden Anweisungen sind bei K&R noch nicht definiert, wohl aber in der ANSI-Norm.

Mit der Anweisung:

```
#line 150  "name.c"
```

können die bei der Ausgabe von Fehlermeldungen gemachten Angaben über Datei und Zeilennummer beeinflußt werden. Diese Möglichkeit wird hauptsächlich beim Einsatz von Programmgeneratoren genutzt.

Mit der #pragma-Anweisung können Compiler-spezifische Fähigkeiten ausgenutzt werden. Compiler, die diese Fähigkeiten nicht haben, ignorieren die Zeilen mit der Anweisung #pragma.

Beispiele für unterstützte Pragmas bei Borland-C

- inline

  kündigt Assembler-Anweisungen im Quelltext an

- warn

  unterdrückt die nachfolgend angegebene Warnung

- saveregs

  verhindert Veränderungen der Segmentregister

Unterstützte Pragmas bei Microsoft-C

- check_stack

  schaltet die Stackprüfung ein oder aus

- check_pointer

  schaltet die Zeigerüberprüfung ein oder aus

- message

  gibt die entsprechende Meldung auf stdout aus

- pack

  Beeinflußt das Packen von Strukturen

18 Die Standard-Bibliotheken

Welche Funktionen durch die Standard-Bibliotheken bereitgestellt werden, hängt von der verwendeten Entwicklungsumgebung ab. Wie wir eingangs gelernt haben, sind mehrere 'C'-Standards zu unterscheiden. Der Standard mit der zur Zeit größten Bedeutung ist der ANSI-Standard. Deshalb werden in der nachfolgenden Übersicht auch nur die Bibliotheken ausführlich vorgestellt, die dem ANSI-Standard entsprechen. In den verschiedenen Entwicklungsumgebungen stehen darüber hinaus in der Regel noch weitere Funktionen bereit. Zum einen sind dies Funktionen und Bibliotheken, die aufgenommen wurden, um die Kompatibilität zu älteren Standards zu wahren. Zum anderen handelt es sich um Funktionen, die aufgenommen, um bestimmte Eigenschaften der Rechner auszunutzen. So gibt es bei fast allen Entwicklungssystemen für MS-DOS-Rechner z. B. eine Bibliothek mit Funktionen zur Ansteuerung des Grafikadapters im Grafikmodus. Wenn Sie solche Funktionen in Ihren Programmen verwenden, so müssen Sie sich dessen bewußt sein, daß dies bei einer Portierung Ihrer Programme auf andere Systeme zu Problemen führen kann. Deswegen sollten Sie die vom Entwicklungssystem abhängigen Programmteile in eigenen Modulen codieren, so daß bei einer Portierung nur diese Module angepaßt werden müssen.

Zur Nutzung der in der Laufzeit-Bibliothek von ANSI-C verfügbaren Funktionen stehen Ihnen folgende Include-Dateien zur Verfügung:

- assert.h
- ctype.h
- float.h
- limits.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stdio.h
- stdlib.h
- string.h
- time.h

Die einzelnen in den jeweiligen Include-Dateien definierten Funktionen, Variablen und Konstanten werden in den nachfolgenden Abschnitten beschrieben.

Ferner stehen Ihnen je nach System weitere Bibliotheken zur Verfügung. Die nachfolgende Aufstellung soll Ihnen als Anhalt zur Orientierung dienen und erhebt keinen Anspruch auf Vollständigkeit.

Include-Dateien, die aus Kompatibilitätsgründen zu älteren 'C'-Standards enthalten sind:

- errno.h (Fehlernummern)
- fcntl.h (Dateikontrolle)
- io.h (Ein- und Ausgabe)
- malloc.h (Dynamische Speicherverwaltung)
- memory.h (Puffermanipulation)
- process.h (Prozeßverwaltung)
- search.h (Such- und Sortierfunktionen)
- stddef.h (Standard-Definitionen)
- sys\stat.h (Strukturtypen für den Dateistatus)
- sys\timeb.h (Strukturtypen für Zeitfunktionen)
- sys\types.h (Standardtypen)
- sys\utime.h (Zeitfunktionen)

Include-Dateien, die z. B. beim Microsoft Quick-C Compiler enthalten sind, um die speziellen Funktionen für MS-DOS-Systeme bereitzustellen:

- bios.h (Zugriff auf BIOS-Routinen)
- conio.h (Direkter Zugriff auf die Systemkonsole)
- direct.h (Handhabung von Verzeichnissen)
- dos.h (Zugriff auf DOS-Funktionen)
- graph.h (Behandlung eines Grafikbildschirms)
- share.h (Dateizugriff in Netzen)
- sys\locking.h (Dateizugriff in Netzen)

In den nachfolgenden Abschnitten werden die Include-Dateien des ANSI-Standards vorgestellt.

18.1 Die Include-Datei assert.h

In dieser Include-Datei ist nur ein einziges Makro definiert. Dieses Makro dient dazu, Fehler bei der Programmentwicklung aufzudecken.

Die allgemeine Form für den Aufruf des Makros ist:

```
assert( ausdruck );
```

Wenn der als Parameter angegebene Ausdruck falsch ist, wird beim Aufruf des Makros der Programmlauf abgebrochen und eine entsprechende Meldung mit Angaben zur Aufrufstelle des Makros (Dateiname und Zeilennummer) ausgegeben. Auf diese Weise lassen sich logische Programmierfehler leichter finden.

Wenn nach dem Austesten der Programme die assert-Anweisungen entfernt werden sollen, so können Sie dies automatisch über die Definition einer Konstanten NDEBUG erreichen. Wenn diese Konstante definiert wurde, so werden beim Preprozessorlauf die assert-Anweisungen durch leere Anweisungen ersetzt.

18.2 Die Include-Datei ctype.h

In dieser Include-Datei sind Makros und Konstanten definiert, die zur Zeichenklassifizierung verwendet werden können.

Für den Benutzer stehen die in der folgenden Tabelle definierte Makros zur Verfügung:

isalnum	`int isalnum( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein alphanumerisches Zeichen (A bis Z oder a bis z oder 0 bis 9) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isalpha	`int isalpha( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein Buchstabe (A bis Z oder a bis z) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
iscntrl	`int iscntrl( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein Steuerzeichen (ASCII-Codes 0 bis 31 und 127) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isdigit	`int isdigit( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein numerisches Zeichen (0 bis 9) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.

isgraph	`int isgraph( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein druckbares Zeichen aber kein Leerzeichen (ASCII-Codes 33 bis 126) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
islower	`int islower( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein kleiner Buchstabe (a bis z) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isprint	`int isprint( int Zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein druckbares Zeichen einschließlich dem Leerzeichen (ASCII-Codes 32 bis 126) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
ispunct	`int ispunct( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen das Interpunktionszeichen ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isspace	`int isspace( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein unsichtbares Zeichen (white spaces; ASCII-Codes 9 bis 14 und 32) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isupper	`int isupper( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen ein großer Buchstabe (A bis Z) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
isxdigit	`int isxdigit( int zeichen )` Dieses Makro testet, ob das als Parameter angegebene Zeichen eine gültige Hex-Ziffer (a bis f oder A bis F oder 0 bis 9) ist. Wenn das Zeichen die Bedingung erfüllt, dann ist das Ergebnis ungleich 0. Wenn die Bedingung nicht erfüllt ist, dann ist das Ergebnis gleich 0.
tolower	`int tolower( int zeichen )` Dieses Makro wandelt einen Großbuchstaben (A bis Z) in den entsprechenden Kleinbuchstaben um.
toupper	`int toupper( int zeichen )` Dieses Makro wandelt einen Kleinbuchstaben (a bis z) in den entsprechenden Großbuchstaben um.

Die in der Tabelle aufgeführten Makros setzen voraus, daß die bearbeiteten Zeichen ASCII-Zeichen sind.

18.3 Die Include-Datei float.h

In der Include-Datei float.h sind Konstanten definiert, die für den jeweiligen Compiler die Datentypen float und double beschreiben. Darüber hinaus können in dieser Include-Datei Funktionen und Makros definiert sein, die den für das jeweilige Entwicklungssystem verfügbaren numerischen Coprozessor unterstützen.

Die nachfolgende Tabelle zeigt eine Auswahl der definierten Konstanten:

Name	Wert für Microsoft-C	Bedeutung
DBL_DIG	15	Präzision als Anzahl der Nachkommastellen
DBL_EPSILON	2.2204460492503131e-016	Die kleinste darstellbare Zahl vom Datentyp double, für die gilt: 1.0 + DBL_EPSILON != 1.0
DBL_MANT_DIG	53	Anzahl der Bits in der Mantisse
DBL_MAX	1.7976931348623158e+308	Größte darstellbare Zahl vom Datentyp double
DBL_MAX_10_EXP	308	Höchster Dezimalexponent
DBL_MIN	2.2250738585072014e-308	Kleinste darstellbare Zahl vom Datentyp double.
DBL_MIN_10_EXP	-307	Kleinster Dezimalexponent
FLT_DIG	6	Präzision als Anzahl der Nachkommastellen
FLT_EPSILON	1.192092896e-07	Kleinste darstellbare Zahl vom Datentyp float, für die gilt: 1.0 + FLT_EPSILON != 1.0
FLT_MANT_DIG	24	Anzahl der Bits in der Mantisse
FLT_MAX	3.402823466e+38	Größte darstellbare Zahl vom Datentyp float
FLT_MAX_10_EXP	38	Höchster Dezimalexponent
FLT_MIN	1.175494351e-38	Kleinste darstellbare Zahl vom Datentyp float
FLT_MIN_10_EXP	-37	Kleinster Dezimalexponent

18.4 Die Include-Datei limits.h

In der Include-Datei limits.h sind Konstanten definiert, die für den jeweiligen Compiler die ganzzahligen Datentypen beschreiben.

Die nachfolgende Tabelle zeigt eine Auswahl der definierten Konstanten:

Name	Wert	Bedeutung
CHAR_MAX	127	Höchstwert für char
CHAR_MIN	-127	Mindestwert für char
SCHAR_MAX	127	Höchstwert für signed char
SCHAR_MIN	-127	Mindestwert für signed char
UCHAR_MAX	255	Höchstwert für unsigned char
CHAR_BIT	8	Anzahl der Bits beim Datentyp char
USHRT_MAX	0xffff	Höchstwert für unsigned short
SHRT_MAX	32767	Höchstwert für (signed) short
SHRT_MIN	-32767	Mindestwert für (signed) short
UINT_MAX	0xffff	Höchstwert für unsigned int
ULONG_MAX	0xffffffff	Höchstwert für unsigned long
INT_MAX	32767	Höchstwert für (signed) int
INT_MIN	-32767	Mindestwert für (signed) int
LONG_MAX	2147483647	Höchstwert für (signed) long
LONG_MIN	-2147483647	Mindestwert für (signed) long

18.5 Die include-Datei math.h

Die Include-Datei math.h enthält die Funktionsprototypen für spezielle mathematische Funktionen. Darüber hinaus sind einige Konstanten definiert, die im Fehlerfall von den verwendeten Funktionen zurückgegeben werden. Als Beispiel hierfür sei die Konstante HUGE_VAL genannt, die bei einem Überlauf als Funktionswert zurückgegeben wird.

Die nachfolgende Tabelle gibt eine Übersicht über die definierte Funktionen:

acos	`double acos( double x )` Diese Funktion liefert als Funktionswert den Arcuscosinus des Parameters x.
asin	`double asin( double x )` Diese Funktion liefert als Funktionswert den Arcussinus des Parameters x.
atan	`double atan( double x )` Diese Funktion liefert als Funktionswert den Arcustangens des Parameters x.
atan2	`double atan2( double x, double y )` Diese Funktion liefert als Funktionswert den Arcustangens von x/y.
ceil	`double ceil( double x )` Diese Funktion liefert als Funktionswert den auf die nächste ganze Zahl aufgerundeten Wert des Parameters x.
cos	`double cos( double x )` Diese Funktion liefert als Funktionswert den Cosinus des Parameters x.
cosh	`double cosh( double x )` Diese Funktion liefert als Funktionswert den Cosinus Hyperbolicus des Parameters x.
exp	`double exp( double x )` Diese Funktion liefert als Funktionswert das Ergebnis der Exponential-Funktion (e^x).
fabs	`double fabs( double x )` Diese Funktion liefert als Funktionswert den Absolutwert des Parameters x.
floor	`double floor( double x )` Diese Funktion liefert als Funktionswert den auf die nächste ganze Zahl abgerundeten Wert des Parameters x.
fmod	`double fmod( double x, double y )` Diese Funktion liefert als Funktionswert den Rest der Division von x/y.
frexp	`double frexp( double x, int *n )` Diese Funktion liefert als Funktionswert die Mantisse des Parameters x. Über den Parameter n, wird der Exponent des Parameters x zurückgegeben. Es gilt die Formel $x = m * 2^n$ und $0.5 < m < 1.0$.

ldexp	`double ldexp( double m, int n )` Diese Funktion liefert als Funktionswert das Ergebnis der Funktion $m * 2^n$.
log	`double log( double x )` Diese Funktion liefert als Funktionswert den natürlichen Logarithmus des Parameters x.
log10	`double log10( double x )` Diese Funktion liefert als Funktionswert den Logarithmus des Parameters x zur Basis 10.
modf	`double modf( double x, double y )` Diese Funktion teilt den Parameter x in Vor- und Nachkommastellen auf. Als Funktionswert werden die Nachkommastellen des Parameters x zurückgegeben. Über den Parameter y werden die Vorkommastellen zurückgegeben.
pow	`double pow( double m, double n )` Diese Funktion liefert als Funktionswert das Ergebnis der Funktion m^n.
sin	`double sin( double x )` Diese Funktion liefert als Funktionswert den Sinus des Parameters x.
sinh	`double sinh( double x )` Diese Funktion liefert als Funktionswert den Sinus Hyperbolicus des Parameters x.
sqrt	`double sqrt( double x )` Diese Funktion liefert als Funktionswert die Quadratwurzel des Parameters x.
tan	`double tan( double x )` Diese Funktion liefert als Funktionswert den Tangens des Parameters x.
tanh	`double tanh( double x )` Diese Funktion liefert als Funktionswert den Tangens Hyperbolicus des Parameters x.

Alle trigonometrischen Funktionen arbeiten mit dem Bogenmaß.

18.6 Die Include-Datei setjmp.h

In der Include-Datei setjmp.h sind zwei Funktionen definiert, mit denen Sie die normale Folge von Funktionsaufruf und Funktionsende umgehen können. Dadurch können Sie quasi globale goto's realisieren.

Durch die Funktion setjmp kann die aktuelle Stackumgebung gesichert werden und durch die Funktion longjmp kann sie wieder hergestellt werden.

18.7 Die Include-Datei signal.h

In der Include-Datei signal.h sind zwei Funktionen definiert, mit denen Sie auf Signale reagieren bzw. Signale erzeugen können.

Mit der Funktion signal, können Sie eine Behandlungsroutine für ein Signal setzen bzw. sperren. Mit der Funktion raise können Sie selbst Signale senden. Die für das entsprechende Signal gesetzte Behandlungsroutine wird aufgerufen, sobald das entsprechende Signal innerhalb des Systems erzeugt wird. Wenn die Behandlungsroutine beendet ist, wird die Ausführung des Programms an der Stelle fortgesetzt an der das Programm zum Zeitpunkt des Eintreffens des Signals war.

Diese Funktionen werden in der Regel für die Ausnahmebehandlung eingesetzt, wie z. B. Interrupt-Signale einer externen Quelle oder Fehler bei der Programmausführung. In der Include-Datei signal.h sind bereits einige Signale vordefiniert. Diese führen bei Ihrem Auftreten zum Abbruch des Programms. Durch Neudefinition der Behandlungsroutinen können Sie z. B. Fehler abfangen und korrigieren und somit eine unbeabsichtigte Beendigung des Programms verhindern.

18.8 Die Include-Datei stdarg.h

Die in dieser Include-Datei definierten Makros ermöglichen den Zugriff auf die Parameter von Funktionen mit variablen Argumentlisten. Dadurch wird der Zugriff auf im Prinzip beliebig lange Argumentlisten möglich.

Drei Makros sind für den Zugriff auf die Argumentlisten definiert:

- `void va_start( va_list ap, last_arg )`

  Mit diesem Makro muß die Bearbeitung der variablen Parameterlisten initialisiert werden. last_arg ist der letzte benannte Parameter in dem deklarierten Funktionskopf. Durch den Aufruf des Makros wird der Zeiger auf die Argumentliste (ap) initialisiert.

- `Typ va_arg( va_list ap, Typ )`

  Durch den wiederholten Aufruf des Makro va_arg können die einzelnen Argumente aus der Argumentliste geholt werden. Die Typen der Elemente müssen bekannt sein. Dies wird in der Regel dadurch erreicht, daß für eine Funktion nur ein Datentyp für die variablen Argumente erlaubt wird oder die Typen der einzelnen Argumente der auswertenden Funktion über einen Formatstring angegeben werden. Die Anzahl der Argumente kann ebenfalls über einen Formatstring festgelegt werden. Sie können aber auch einen speziellen Wert als Ende der Argumentliste interpretieren (z. B. -1 oder NULL).

- `void va_end( va_list )`

  Dieses Makro muß nach der Abarbeitung der Argumentliste einmal aufgerufen werden, um das Ende der Bearbeitung der Argumentliste anzuzeigen und die intern benutzten Größen zurückzusetzen.

Beachten Sie bitte, daß es auch im UNIX-Standard einen Mechanismus zur Abarbeitung variabler Argumentlisten gibt. Dieser Mechanismus ist aber nicht identisch mit dem hier vorgestellten Mechanismus zur Abarbeitung der variablen Argumentlisten.

Darüber hinaus stellt ANSI-C Funktionen bereit, die die Auswertung von variablen Argumentlisten automatisch vornehmen können, wenn der variablen Argumentliste ein von den Funktionen der printf-Familie her bekannter Formatstring vorangestellt ist. Dies sind die Funktionen:

- vprintf
- vfprintf
- vsprintf

Diese Funktionen können dann anstelle des Makros va_arg verwendet werden.

18.9 Die Include-Datei stdio.h

Die Include-Datei stdio.h enthält neben Konstanten, Makros und Typen für die Ein- und Ausgabe auch die Funktionsprototypen für die Ein- und Ausgabefunktionen.

Der wohl wichtigste Datentyp, der in dieser Datei definiert ist, ist der Datentyp FILE. Dies ist der Datentyp für einen sogenannten Ein- und Ausgabestrom. Sie können sich eine solchen Strom am besten als eine beliebig lange Folge von Bytes vorstellen.

Zwei wichtige Konstanten, die in dieser Include-Datei definiert sind, sind die Konstanten NULL und EOF. Die Konstante NULL definiert den Wert des sogenannten Nullzeigers und die Konstante EOF definiert den Wert des Dateiendezeichens.

Die nachfolgende Tabelle gibt einen Überblick über die definierten Funktionen und Makros:

clearerr	`void clearerr( FILE *f )` Diese Funktion setzt den Fehlerstatus der Datei f zurück.
fclose	`int fclose( FILE *f )` Diese Funktion schließt die Datei f. Wenn die Funktion erfolgreich war, hat sie den Funktionswert 0, sonst den Funktionswert EOF.
feof	`int feof( FILE *f )` Diese Funktion liefert als Funktionswert einen Wert ungleich 0, wenn das Dateiende erreicht ist, sonst liefert sie den Funktionswert 0.
ferror	`int ferror( FILE *f )` Der Funktionswert dieser Funktion zeigt einen Fehler bei einem Zugriff auf die Datei f an. Wenn kein Fehler aufgetreten ist, liefert die Funktion als Funktionswert 0. Nach dem Aufruf von ferror bleibt der Fehlerstatus für die Datei f solange gesetzt bis die Datei geschlossen oder die Funktion clearerr aufgerufen wurde.
fflush	`int fflush( FILE *f )` Schreibt den Inhalt des Puffers der Datei f in die Datei f. Der Aufruf fflush(NULL) bezieht sich auf alle Dateien. Der Funktionswert ist 0, wenn die Funktion erfolgreich war und sonst EOF.
fgetc	`int fgetc( FILE *f )` Diese Funktion liefert als Funktionswert das nächste Zeichen aus einer zum Lesen geöffneten Datei. Die Funktion liefert als Funktionswert EOF, wenn das Dateiende erreicht ist.

fgetpos	`int fgetpos( FILE *f, fpos_t *pos )` Diese Funktion gibt über den Parameter pos die aktuelle Position des Dateizeigers der Datei f zurück. Dieser Wert kann mit der Funktion fsetpos benutzt werden, um den Dateizeiger wieder zu positionieren. Wenn die Funktion erfolgreich war, liefert sie als Funktionswert den Wert 0, sonst einen Wert ungleich 0.
fgets	`char *fgets( char *str, int n, FILE *f )` Diese Funktion liest eine Zeichenkette aus der Datei f. Die Zeichenkette wird in den durch den Zeiger *str referenzierten Speicherbereich übertragen. Durch die Funktion werden alle Zeichen bis zum nächsten Zeichen '\n' gelesen, jedoch höchstens n Zeichen, um einen Überlauf des zur durch str zur Verfügung gestellten Speicherbereiches zu verhindern. Als Funktionswert gibt die Funktion einen Zeiger auf die eingelesene Zeichenkette zurück, wenn sie erfolgreich war und sonst NULL.
fopen	`FILE *fopen( char *name, char *mode )` Diese Funktion öffnet die durch den Dateinamen name angegebene Datei. Der Parameter mode bestimmt, ob die Datei zum Anhängen, Schreiben und/oder Lesen geöffnet werden soll. Als Funktionswert wird ein Zeiger auf eine FILE-Objekt zurückgegeben, wenn die Funktion erfolgreich war und sonst der Nullzeiger.
fprintf	`int fprintf( FILE *f, char *format, ... )` Diese Funktion dient zum Schreiben von Werten in eine Textdatei. Als Funktionswert wird die Anzahl der ausgegebenen Zeichen zurückgegeben, wenn die Funktion erfolgreich war und sonst EOF. Eine genauere Beschreibung der Funktion finden Sie im Kapitel zur Ein- und Ausgabe.
fputc	`int fputc( char c, FILE *f )` Diese Funktion gibt das Zeichen c in die angegebene Datei aus. Die Funktion liefert als Funktionswert das ausgegebene Zeichen, wenn sie erfolgreich war und sonst EOF.
fputs	`int fputs( char *str, FILE *f )` Diese Funktion gibt die durch str referenzierte Zeichenkette in die Datei f aus. Die Funktion liefert als Funktionswert 0, wenn sie erfolgreich war und sonst EOF.
fread	`int fread( void *buf, unsigned s, unsigned n, FILE *f )` Diese Funktion liest n Datensätze mit einer Größe von je s Bytes aus der Datei f und überträgt sie in den durch den Zeiger buf referenzierten Speicherbereich. Als Funktionswert gibt sie die Anzahl der tatsächlich gelesenen Datensätze zurück. Diese Zahl kann von n abweichen, wenn z. B. das Dateiende während des Lesevorgangs erreicht wurde. Mit den Funktionen feof und ferror kann beim Funktionswert 0 geprüft werden, ob ein Fehler vorliegt oder das Dateiende erreicht wurde.

freopen	`FILE *freopen( char *name, char *mode, FILE *f )` Diese Funktion schließt die Datei f und öffnet gleichzeitig die mit dem Parameter name angegebene Datei. Durch den Parameter mode wird bestimmt, ob die Datei zum Anhängen, Schreiben und/oder Lesen geöffnet werden soll. Als Funktionswert wird ein Zeiger auf eine FILE-Objekt zurückgegegeben, wenn die Funktion erfolgreich war und sonst NULL.
fscanf	`int fscanf( FILE *f, char *format, ... )` Diese Funktion dient zum Lesen von Werten aus der Datei f. Die Funktion liefert als Funktionswert die Anzahl der eingelesenen Werte, wenn sie erfolgreich war und sonst EOF. Eine genauere Beschreibung der Funktion finden sie im Kapitel über die Ein- und Ausgabe.
fseek	`int fseek( FILE *f, long offset, int pos )` Diese Funktion setzt den Dateizeiger an eine Position, die durch den Parameter offset relativ zu einer Ausgangsposition pos beschrieben wird. Als Funktionswert gibt die Funktion 0 zurück, wenn sie erfolgreich war und sonst EOF. Eine genauere Beschreibung der Funktion finden Sie im Kapitel zur Dateibehandlung.
fsetpos	`int fsetpos( FILE *f, fpos_t pos )` Diese Funktion setzt den Dateizeiger auf die durch den Parameter pos beschriebene Position. Der als Parameter anzugebende Positionszeiger muß zuvor mit der Funktion fgetpos ermittelt worden sein. Die Funktion liefert als Funktionswert 0, wenn sie erfolgreich war und sonst einen Wert ungleich 0.
ftell	`long ftell( FILE *f )` Diese Funktion liefert als Funktionswert die aktuelle Position des Dateizeiger für die Datei f relativ zum Dateianfang. Wenn die Funktion nicht erfolgreich war, liefert sie als Funktionswert EOF.
fwrite	`int fwrite( void *b, unsigned s, unsigned n, FILE *f )` Diese Funktion schreibt n Datensätze der Größe s Bytes in die Datei f. Die Datensätze werden aus dem durch den Zeiger b referenzierten Speicherbereich übernommen. Die Funktion liefert als Funktionswert die Anzahl der tatsächlich geschriebenen Datensätze.
getc	`int getc( FILE *f )` Dies ist die Makroimplementierung der Funktion fgetc.
getchar	`int getchar( void )` Dieses Makro entspricht dem Makroaufruf getc(stdin).

gets	`char *gets( char *str )` Diese Funktion liest eine Zeichenkette von der Standardeingabedatei stdin. Die eingelesene Zeichenkette wird in den durch Zeiger str referenzierten Speicherbereich übertragen. Die Funktion liefert, wenn sie erfolgreich war, als Funktionswert einen Zeiger auf die eingelesene Zeichenkette und sonst den Nullzeiger.
perror	`void perror( char *msg )` Diese Funktion gibt die durch Parameter msg angegebene Zeichenkette gefolgt von einer Systemfehlermeldung auf stderr aus. Die Funktion sollte unmittelbar nach Auftreten des Fehlers aufgerufen werden. Die ausgegebene Systemfehlermeldung wird durch die globale Variable errno bestimmt.
printf	`int printf( char * format, ... )` Diese Funktion entspricht dem Funktionsaufruf fprintf(stdout, format, ...). Weitere Informationen finden Sie im Kapitel zur Ein- und Ausgabe.
putc	`int putc( char c, FILE *f )` Dies ist die Makroimplementierung der Funktion fputc.
putchar	`int putchar( char c )` Dieses Makro entspricht dem Makroaufruf putc(c, stdout).
puts	`int puts( char *str )` Diese Funktion entspricht dem Funktionsaufruf fputs(str, stdout).
remove	`int remove( char *name )` Diese Funktion löscht die Datei mit dem durch den Parameter name angegebenen Dateinamen. Die Funktion liefert als Funktionswert 0, wenn sie erfolgreich war und sonst EOF.
rename	`int rename( char *oldname, char *newname )` Diese Funktion gibt einer Datei einen neuen Namen. Die Datei wird ausgewählt durch den Parameter oldname. Der neue Dateiname wird durch den Parameter newname angegeben. Die Funktion liefert als Funktionswert 0, wenn sie erfolgreich war und sonst EOF.
rewind	`void rewind( FILE *f )` Der Dateizeiger der Datei f wird auf den Dateianfang gesetzt. Gleichzeitig wird der Fehlerstatus der angegebenen Datei zurückgesetzt.
scanf	`int scanf( char *format, ... )` Diese Funktion entspricht dem Funktionsaufruf fscanf(stdin, format, ...).

setbuf	`void setbuf( FILE *f, char *buf )` Durch einen Aufruf dieser Funktion kann der Anwender einen eigenen Puffer für die durch den Parameter f angegebene Datei einrichten. Dieser Puffer wird von den Ein- und Ausgabe-Operationen dann anstelle des vom System eingerichten Puffers verwendet.
setvbuf	`int setvbuf( FILE *f, char *b, int m, unsigned s )` Wie die Funktion setbuf, jedoch können hierbei auch die Größe des Puffers (s) und der Puffermodus (m) bestimmt werden.
sprintf	`int sprintf( char *str, char *format, ... )` Wie die Funktion fprintf, jedoch Ausgabe in die durch den Parameter str referenzierte Zeichenkette.
sscanf	`int sscanf( char *str, char *format, ... )` Wie die Funktion fscanf, jedoch Eingabe von der durch den Parameter str referenzierten Zeichenkette.
tempnam	`char *tempnam( char *dir, char *nam )` Diese Funktion erzeugt einen eindeutigen Dateinamen für eine temporäre Datei in dem angegebenen Verzeichnis. Dieser Dateiname setzt sich zusammen aus dem Namensvorsatz nam und einer numerischen Zeichenkette. Der Namensvorsatz sollte nur drei Zeichen haben.
tmpfile	`FILE *tmpfile( void )` Diese Funktion erzeugt eine temporäre datei und gibt als Funktionswert einen Zeiger auf diese Datei zurück. Wenn die Funktion nicht erfolgreich war, gibt sie als Funktionswert den Nullzeiger zurück. Die Datei wird automatisch wieder gelöscht, wenn sie geschlossen oder das Programm beendet wird.
ungetc	`int ungetc( char c, FILE *f )` Diese Funktion stellt das Zeichen c in den Eingabepuffer der durch den Parameter f angegebenen Datei zurück.
vfprintf	`int vfprintf( char *format, va_list ap )` Wie die Funktion fprintf, jedoch mit variabler Parameterliste. Nähere Informationen zu den variablen Parameterlisten finden sie Abschnitt zur Include-Datei stdarg.h.
vprintf	`int vprintf( char *str, char *format, val_list ap )` Wie die Funktion printf, jedoch mit variabler Parameterliste.
vsprintf	`int vsprintf( char *str, char *format, val_list ap )` Wie die Funktion sprintf, jedoch mit variabler Parameterliste.

18.10 Die Include-Datei stdlib.h

Die Include-Datei enthält neben einigen globalen Variablen die mehr allgemeinen Funktionen der Standard-Bibliothek. Als globale Variablen sind zu nennen:

- environ
- errno
- sys_errlist
- sys_nerr

Die Variable environ ist ein Vektor von Zeiger auf die Zeichenketten, die die Prozeßumgebung bilden. Bei MS-DOS-Systemen sind dies die Variablen des DOS-Umgebungsbereiches. Die einzelnen Zeichenketten haben die Form Name=Zeichenkette (z. B. PATH=C:\;C:\DOS;). Durch die Auswertung dieses Vektors können Sie Informationen über die gesetzten Umgebungsvariablen erhalten. Zum Auswerten und Setzen der Umgebungsvariablen können Sie die Funktionen getenv bzw. putenv verwenden.

Die Variablen errno, sys_errlist und sys_nerr werden von der Funktion perror verwendet. Wenn auf Systemebene ein Fehler auftritt, so wird die Variable errno automatisch auf einen ganzzahligen Wert gesetzt. Die Funktion perror benutzt diesen Wert, um aus der Tabelle sys_errlist die entsprechende Fehlermeldung auszuwählen. Die Variable sys_nerr ist als Anzahl der Einträge in der sys_errlist definiert.

Die folgende Tabelle gibt Ihnen einen Überblick über die Funktionen der Standard-Bibliothek

abort	`void abort( void )` Das aktuell laufende Programm wird abgebrochen und eine entsprechende Fehlermeldung wird auf stderr ausgegeben. Der Aufruf der Funktion erzeugt ein entsprechendes Signal, das über einen eigene Behandlungsroutine abgefangen werden kann.
abs	`int abs( int x )` Diese Funktion liefert als Funktionswert den Absolutwert des Parameters x.
atexit	`int atexit( void *f( void ))` Diese Funktion ermöglicht es Funktionen zu bestimmen, die automatisch beim Programmende aufgerufen werden. Bis zu 32 Funktionen können auf diese Weise gesetzt werden. Die gesetzten Funktionen werden nach dem LIFO-Prinzip abgearbeitet. Der Funktionswert der Funktion ist 0, wenn die Funktion erfolgreich war, sonst ungleich 0.
atof	`double atof( char *str )` Diese Funktion wandelt, wenn möglich, die über den Parameter str referenzierte Zeichenkette in eine Gleitkommazahl um. Der Funktionswert entspricht der umgewandelten Zahl, wenn die Funktion erfolgreich war. Der Funktionswert ist 0, wenn die Zeichenkette nicht in eine Zahl umgewandelt werden konnte.

atoi	`int atoi( char *str )` Wie die Funktion atof, jedoch für ganzzahlige Zahlen vom Typ int.
atol	`long atol( char *str )` Wie die Funktion atof, jedoch für ganzzahlige Zaheln vom Typ long.

bsearch	`void *bsearch( void *s, void *lst,`
	`int n, int size,`
	`int (*comp)( void *e1, void *e2 ) )`
	
	Diese Funktion sucht ein dem durch den Parameter s referenzierten Element entsprechendes Element in dem durch den Parameter lst referenzierten Vektor nach binären Suchverfahren. Der Parameter n bestimmt die Anzahl der Listenelemente und der Parameter size die Größe der Listenelemente in Bytes. Der letzte Parameter referenziert die zu benutzende Vergleichsfunktion. Wenn die Funktion erfolgreich war, liefert sie als Funktionswert einen Zeiger auf das gefundene Listenelement sonst den Nullzeiger.

calloc	`void *calloc( unsigned n, unsigned size )` Die Funktion calloc reserviert dynamisch Speicherplatz für einen Vektor mit n Elementen und einer Größe der Elemente von je size Bytes. Die Funktion liefert als Funktionswert einen Zeiger auf den Anfang des reservierten Speicherbereiches, wenn sie erfolgreich war, und sonst den Nullzeiger.
div	`struct div_t { int quot; int rem; } div( int x, int y )` Die Funktion bestimmt den Quotienten und den Divisionsrest von x/y. Das Ergebnis wird als Funktionswert in einer Struktur vom Typ div_t zurückgegeben.
exit	`void exit( int exitcode )` Ein Aufruf dieser Funktion führt zu einem normalen Programmende. Über den Parameter exitcode kann dem aufrufenden Prozeß ein Funktionswert übergeben werden.
free	`void free( void *ptr )` Diese Funktion gibt den durch den Parameter ptr referenzierten Speicherplatz wieder frei. Der Speicherplatz muß zuvor mit einer der Funktionen calloc, malloc oder realloc dynamisch angefordert worden sein.
getenv	`char *getenv( char *name )` Diese Funktion durchsucht die Liste der Umgebungsvariablen nach der durch den Parameter name angegebenen Variablen. Wenn die Funktion die Variable findet, gibt sie als Funktionswert einen Zeiger auf den entsprechenden Eintrag im Umgebungsbereich des Programmes zurück sonst liefert sie den Nullzeiger.
labs	`long labs( long x )` Diese Funktion liefert als Funktionswert den Absolutwert des Parameters x.

ldiv	`struct ldiv_t {long q; long r;} ldiv( long x, long y )` Wie die Funktion div, jedoch für den Datentyp long.
malloc	`void *malloc( int size )` Diese Funktion reserviert einen Speicherbereich mit einer Größe von size Bytes. Wenn die Funktion erfolgreich war, liefert sie als Funktionswert einen Zeiger auf den Anfang des reservierten Speicherbereichs sonst liefert sie den Nullzeiger.

qsort	`void qsort( void *lst, int n, int size,`
	`int (*comp)( void *e1, void *e2) )`
	
	Diese Funktion sortiert einen durch Parameter lst referenzierten Vektor mit n Elementen der Größe size nach dem Quicksort-Verfahren. Der vierte Parameter referenziert die zu verwendende Vergleichsfunktion.

rand	`int rand( void )` Diese Funktion liefert als Funktionswert eine ganzzahlige Zufallszahl im Bereich zwischen 0 und 32767. Um den Zufallszahlengenerator zu initialisieren, können Sie die Funktion srand verwenden. Wenn Sie die Zufallszahlengenerator nicht initialisieren, liefert Ihnen die Funktion rand nach jedem Programmstart reproduzierbare Zahlenfolgen.
realloc	`void *realloc( void *buf, unsigned size )` Diese Funktion ermöglicht es einen zuvor mit der Funktion calloc oder malloc dynamisch angeforderten Speicherbereich nachträglich zu vergrößern oder zu verkleinern. Der zuvor vorhandene Inhalt des Speicherbereiches bleibt soweit er noch zur Verfügung steht dabei erhalten. Die Funktion liefert einen Zeiger auf den neu zugeordneten Speicherbereich. Dieser kann von dem ursprünglichen Zeiger verschieden sein, da es notwendig sein kann, daß der Speicherblock innerhalb des Speichers verschoben wird. Wenn die Funktion nicht erfolgreich war, liefert sie als Funktionswert den Nullzeiger zurück.
srand	`void srand( unsigned start )` Diese Funktion initialisiert den Zufallszahlengenerator. Wenn Sie reproduzierbare Folgen von Zufallszahlen erzeugen möchten, dann können Sie den Zufallszahlengenerator mit 1 initialisieren. Ansonsten empfiehlt sich z. B. eine Initialisierung mit der Uhrzeit.
system	`int system( char *command )` Diese Funktion übergibt die über den Parameter command referenzierte Zeichenkette an den Befehlsinterpreter des Systems. Sie können dadurch Systemkommandos und andere Programme von einem Programm aus ausführen lassen. Als Funktionswert gibt die Funktion den Wert 0 zurück, wenn das Kommando erfolgreich ausgeführt werden konnte. Wenn die Funktion nicht erfolgreich war, liefert sie den Wert EOF und setzt die globale Variable errno.

18.11 Die Include-Datei string.h

Die Include-Datei string.h definiert Funktionen zur Manipulation von Zeichenketten. Die nachfolgende Tabelle gibt Ihnen einen Überblick über die zur Verfügung stehenden Funktionen.

memchr	`void *memchr( void *buf, char c, int count )` Diese Funktion sucht innerhalb der ersten count Bytes nach dem ersten Vorkommen des Zeichens c in dem durch den Parameter buf referenzierten Speicherbereich. Die Funktion gibt als Funktionswert einen Zeiger auf das erste gefundene Zeichen in dem Speicherbereich zurück. Wenn das Zeichen nicht gefunden werden konnte, liefert die Funktion den Nullzeiger.
memcmp	`int memcmp( void *buf1, void *buf2, int count )` Diese Funktion vergleicht die ersten count bytes der durch die Parameter buf1 und buf2 referenzierten Speicherbereiche. Sie liefert einen Funktionswert kleiner 0, wenn der Inhalt von buf1 kleiner ist als der Inhalt von buf2. Sie liefert den Funktionswert 0, wenn die verglichenen Bereiche gleich sind und sie liefert einen Funktionswert größer 0, wenn der Inhalt von buf1 größer als der Inhalt von buf2 ist.
memcpy	`void memcpy( void *dst, void *src, unsigned n )` Diese Funktion kopiert n Bytes aus dem durch den Parameter src referenzierten Speicherbreich in den durch Parameter dst referenzierten Speicherbereich. Dabei ist nicht sichergestellt, daß bei überlappenden Speicherbereichen die Bytes des Quellbereiches vor dem Überschreiben kopiert werden. Verwenden Sie zum Kopieren überlappender Speicherbereiche die Funktion memmove. Die Funktion liefert als Funktionswert einen Zeiger auf den Anfang des Zielbereiches.
memmove	`void memmove( void *dst, void *src, unsigned n )` Wie die Funktion memcpy, jedoch ist bei dieser Funktion sichergestellt, daß bei überlappenden Speicherbereiche die Bytes des Quellbereiches vor dem Überschreiben kopiert werden.
memset	`void *memset( void *dst, char c, unsigned count )` Diese Funktion füllt die ersten count Bytes des durch den Parameter dst referenzierten Speicherbereiches mit dem Zeichen c. Als Funktionswert gibt die Funktion einen Zeiger auf den Anfang des gefüllten Speicherbereichs zurück.
strcat	`char *strcat( char *dst, char *src )` Diese Funktion hängt die Zeichenkette src an die Zeichenkette dst an. Als Funktionswert wird ein Zeiger auf dst zurückgegeben.

strchr	`char *strchr( char *str, int c )` Diese Funktion sucht das erste Vorkommen des Zeichens c in der Zeichenkette str. Als Funktionswert wird ein Zeiger auf das gefundene Zeichen zurückgegeben, wenn die Funktion erfolgreich war und sonst der Nullzeiger.
strcmp	`int strcmp( char *s1, char *s2 )` Diese Funktion vergleicht die beiden Zeichenketten s1 und s2. Als Funktionswert wird eine Zahl kleiner 0 zurückgegeben, wenn gilt s1 < s2. Der Funktionswert ist 0, wenn beide Zeichenketten gleich sind und größer 0, wenn gilt s1 > s2.
strcpy	`char *strcpy( char *dst, char *src )` Diese Funktion kopiert die Zeichenkette src in die Zeichenkette dst. Als Funktionswert gibt die Funktion einen Zeiger auf die Zeichenkette dst zurück.
strcspn	`unsigned strcspn( char *s1, char *s2 )` Diese Funktion gibt die Position des ersten Zeichens in s1 zurück, das zu dem in s2 angegebenen Zeichensatz gehört. Dies entspricht der Länge der Teilzeichenkette von s1, die vollständig aus Zeichen besteht, die nicht in der Zeichenkette s2 enthalten sind.
strerror	`char *strerror( int errno )` Diese Funktion liefert die Zeichenkette, die die zu der Fehlernummer errno gehörende Systemfehlermeldung enthält.
strlen	`unsigned strlen( char *s )` Diese Funktion liefert als Funktionswert die Länge der Zeichenkette s.
strncat	`char *strncat( char *dst, char *src )` Diese Funktion hängt n Zeichen der Zeichenkette src an die Zeichenkette dst an. Als Funktionswert liefert sie einen Zeiger auf die Zeichenkette dst.
strncmp	`int *strncmp( char *s1, char *s2, unsigned n )` Wie die Funktion strcmp, jedoch werden nur die ersten n Zeichen bei dem Vergleich berücksichtigt.
strncpy	`char *strncpy( char *dst, char *src, unsigned n )` Wie die Funktion strcpy, jedoch werden nur die ersten n Zeichen der Zeichenkette src kopiert.
strpbrk	`char *strpbrk( char *s1, char *s2 )` Wie die Funktion strcspn, jedoch liefert diese Funktion als Funktionswert einen Zeiger auf das gefundene Zeichen zurück. Wenn die Funktion nicht erfolgreich war, liefert sie als Funktionswert den Nullzeiger.

strrchr	`char *strrchr( char *s, int c )` Diese Funktion sucht das letzte Vorkommen des Zeichens c in der Zeichenkette s. Als Funktionswert wird ein Zeiger auf das gefundene Zeichen zurückgegeben. Wenn die Funktion nicht erfolgreich war, liefert sie als Funktionswert den Nullzeiger.
strspn	`unsigned strcspn( char *s1, char *s2 )` Diese Funktion gibt die Position des ersten Zeichens in s1 zurück, das nicht zu dem in s2 angegebenen Zeichensatz gehört. Dies entspricht der Länge der Teilzeichenkette von s1, die vollständig aus Zeichen besteht, die in der Zeichenkette s2 enthalten sind.
strstr	`char *strstr( char *s1, char *s2 )` Diese Funktion sucht das erste Vorkommen der Zeichenkette s2 in der Zeichenkette s1. Wenn die Funktion erfolgreich war, gibt sie als Funktionswert einen Zeiger auf die gefundene Zeichenkette zurück und sonst den Nullzeiger.
strtod	`double strtod( char *s, char **ptr )` Diese Funktion wandelt die über den Parameter s referenzierte Zeichenkette in eine Zahl mit Datentyp double um. Das Ergebnis der Umwandlung wird als Funktionswert zurückgegeben. Der Zeiger ptr zeigt nach der Ausführung der Funktion auf das erste Zeichen, daß nicht mehr umgewandelt werden konnte. Bei der Ausführung der Funktion wird die globale Variable errno gesetzt. Wenn ein Fehler auftritt, erhält diese Variable den Wert ERANGE.
strtok	`char *strtok( char *s1, char *s2 )` Diese Funktion interpretiert die Zeichenkette s1 als eine Menge von Grundsymbolen (Tokens) und die Zeichenkette s2 als eine Menge von Trennzeichen (Delimeters), die die in s1 enthaltenen Grundsymbole voneinander trennen. Der erste Aufruf der Funktion strtok liefert einen Zeiger auf das erste Grundsymbol in der Zeichenkette s1. Jeder weitere Aufruf der Funktion strtok mit dem Parameter s1 = NULL liefert die weiteren Grundsymbole in der beim ersten Aufruf angegebenen Zeichenkette. Wenn keine (weiteren) Grundsymbole gefunden werden konnten, liefert die Funktion als Funktionswert den Nullzeiger. Durch den Aufruf der Funktion wird auch die Zeichenkette s2 modifiziert, da strtok hinter jedem Grundsymbol in der Zeichenkette, das Zeichen '\0' einfügt. Dies erleichtert das Herauskopieren der Grundsymbole aus der Zeichenkette s2.
strtol	`long strtol( char *s, char **ptr )` Wie die Funktion strtod, jedoch für den Datentyp long.
strtoul	`unsigned long strtoul( char *s, char **ptr )` Wie die Funktion strtod, jedoch für den Datentyp unsigned long.

18.12 Die Include-Datei time.h

Die Include-Datei time.h definiert Funktionen und globale Variablen zur Behandlung von Zeitangaben bzw. zum Ermitteln der Systemzeit und zum Berechnen von Zeitunterschieden.

Folgende globale Variablen sind definiert:

- daylight
- timezone
- tzname

Die Variable daylight bestimmt, ob die Sommerzeit durch das System berücksichtigt wird. Die Variable timezone enthält den Zeitunterschied in Sekunden zwischen der Greenwich Mean Time und der aktuellen Zeitzone. Der Inhalt der Variablen tzname wird bestimmt durch den Inhalt der Umgebungsvariable TZ, mit deren Hilfe Voreinstellungen für Zeitzone und die Verwendung der Sommerzeit getroffen werden können.

Ferner wird in dieser Include-Datei die Struktur tm definiert, die von einigen Zeitfunktionen benutzt wird. Diese Struktur ist wie folgt definiert:

```
struct tm
{
  int tm_sec;   /* Sekunden          (0.. 59) */
  int tm_min;   /* Minuten           (0.. 59) */
  int tm_hour;  /* Stunden           (0.. 23) */
  int tm_mday;  /* Tag               (0.. 31) */
  int tm_mon;   /* Monat             (0.. 11) */
  int tm_year;  /* Jahre seit 1900            */
  int tm_wday;  /* Tage seit Sonntag  (0..  6) */
  int tm_yday;  /* Tage seit den 1.1. (0..365) */
  int tm_isdst; /* Merker für Sommerzeitangabe */
} tm;
```

Die Typen clock_t und time_t sind systemabhängige arithmetische Größen, die die jeweilige Zeit bzw. Datum und Uhrzeit repräsentieren.

Die nachfolgende Tabelle gibt Ihnen einen Überblick über die zur Verfügung stehenden Funktionen:

asctime	`char *asctime( struct tm *time )` Diese Funktion wandelt eine in einer durch den Parameter time referenzierten Struktur gespeicherte Zeitangabe in eine Zeichenkette um. Die Funktion liefert als Funktionswert einen Zeiger auf die erzeugte Zeichenkette.
clock	`clock_t clock( void )` Diese Funktion ermittelt die seit Beginn des laufenden Prozesses vergangene Zeit. Der Funktionswert wird in einem vom verwendeten System abhängigen Format zurückgegeben. Der Funktionswert kann durch Division mit der Konstante CLOCKS_PER_SEC in Sekunden umgerechnet werden.

ctime	`char *ctime( time_t *time )` Diese Funktion wandelt die durch den Parameter time referenzierte Zeitangabe in eine Zeichenkette um. Die Funktion liefert als Funktionswert einen Zeiger auf die erzeugte Zeichenkette.
difftime	`double difftime( time_t *t1, time_t *t2 )` Diese Funktion bestimmt den Zeitunterschied zwischen den Zeiten t1 und t2 in Sekunden. Als Funktionswert gibt die Funktion den Zeitunterschied in Sekunden zurück.
gmtime	`struct tm *gmtime( time_t *time )` Diese Funktion wandelt den Wert der systemabhängigen Darstellung des Parameters time in eine Darstellung mit der Struktur tm um. Damit können Sie die Zeitangaben auswerten. Die Zeitangabe entspricht der sogenannte Greenwich Mean Time. Als Funktionswert gibt die Funktion einen Zeiger auf die erzeugte Struktur zurück.
localtime	`struct tm *localtime( time_t *time )` Wie die Funktion gmtime, jedoch unter Berücksichtigung der für das verwendete System gesetzten Zeitzone.
mktime	`time_t mktime( struct tm *time )` Diese Funktion wandelt eine in einer durch den Parameter time referenzierten Struktur gespeicherte Zeitangabe in die systemabhängige Zeitdarstellung um. Die Funktion liefert als Funktionswert die systemabhängige Zeitdarstellung.
time	`time_t time( time_t *time )` Diese Funktion liefert Datum und Uhrzeit in der systemabhängigen Darstellung.
_strtime	`char _strtime( char *s )` Diese Funktion kopiert die aktuelle Systemzeit im Format HH:MM:SS in die Zeichenkette s. Als Funktionswert wird ein Zeiger auf die Zeichenkette s zurückgegeben.

19 Schlußwort

Ich danke dem Leser, der bis hier gekommen ist, für seine Aufmerksamkeit und wünsche ihm noch viel Vergnügen beim Programmieren in 'C'. Sicherlich werden Sie über den Kurs hinaus noch eine Menge 'C'-Programme schreiben müssen, bis Sie auf dem 'C'-Klavier virtuos spielen und alle Möglichkeiten ausnutzen können. Aber wie heißt es so schön: Übung macht den Meister.

Kevelaer, im Mai 1997

Reinhard Peters

20 Quellennachweis

- Arne Schäpers, Turbo-C-Handbuch, Borland 1988
- Kerninghan und Ritchie, Programmieren in C, Hanser 1983
- Prof. Lippert, Programmiersprachen: 'C', FH Niederrhein 1989
- Microsoft, C-Programmieren leicht gemacht, Microsoft 1989
- Microsoft, Referenzhandbuch zur Laufzeitbibliothek von MS-Quick-C, Microsoft 1989
- Kerninghan und Ritchie, Programmieren in C, Hanser 1990
- Tondo und Gimpel, Das C-Lösungsbuch, Hanser 1990
- Bjarne Stroustrup, Programmieren mit C++, Addison-Wesley 1990