Technische Informatik

Clauß · Fischer

# Programmieren mit C

VEB Verlag Technik Berlin

Clauß · Fischer

Programmieren mit C

# Technische Informatik

Herausgegeben von

Dr.-Ing. Ludwig Claßen

Dr.-Ing. Dieter Nedo

Dr. rer. nat. Gerhard Paulin

# Programmieren mit C

Dipl.-Math. Matthias Clauß
Dipl.-Math. Günther Fischer

2., durchgesehene Auflage

VEB VERLAG TECHNIK BERLIN

Clauß, Matthias
Programmieren mit C / Matthias Clauß ;
Günther Fischer. - 2., durchges. Aufl. -
Berlin : Verl. Technik, 1990
239 S. : 14 Bilder . - (Technische Informatik)
ISBN 3-341-00864-0
NE: Fischer, Günther:

ISBN 3-341-00864-0

2., durchgesehene Auflage

VT 201 · 3/5954-2 (5)
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung: (140) Druckerei
Neues Deutschland, Berlin
Lektor: Jürgen Reichenbach
Einbandgestaltung: Gabriele Schwesinger
LSV 3053
Bestellnummer: 554 277 8
02400

# Vorwort

Systemsoftware unterliegt extremen Anforderungen hinsichtlich Kodeeffizienz, Zuverlässigkeit, Wartungsfreundlichkeit sowie zunehmend auch hinsichtlich der Übertragbarkeit auf andere Rechnerarchitekturen. Moderne höhere Programmiersprachen unterstützen die Entwicklung gut strukturierter, zuverlässiger und portabler Programme. Trotzdem wird insbesondere in der Systemprogrammierung auch noch gegenwärtig vor allem auf Grund geringeren Speicherplatzbedarfes und günstigeren Laufzeitverhaltens der Programme oft auf Assembler zurückgegriffen, obwohl damit erhebliche Nachteile, wie verlängerte Programmentwicklungszeiten usw. verbunden sind.

Die Programmiersprache C wurde im Jahre 1972 von Dennis M. Ritchie entworfen und implementiert. Sie vereinigt die Vorteile höherer Sprachen mit den Vorzügen einer maschinennahen Programmierung. Wesentliche Konzepte von C entstammen der typenlosen Systemprogrammiersprache BCPL /41/.

Die Entwicklung portabler und zuverlässiger Software gewinnt mit dem breiten Einsatz leistungsfähiger Mikrorechnersysteme zunehmend an Bedeutung. Eine wichtige Rolle spielt dabei das Betriebssystem UNIX /43/, /3/, /8/, /9/, /32/, /33/, das auf unterschiedlichen Rechnerklassen eine einheitliche Nutzerschnittstelle anbietet und bei 16- und 32-Bit-Mikrorechnersystemen mittlerweile einen De-facto-Industriestandard darstellt. Im Zusammenhang mit UNIX hat C international weite Verbreitung gefunden. C-Compiler existieren auf den unterschiedlichsten Rechnerarchitekturen über die gesamte Breite von Mikrorechnersystemen bis zu Hochleistungsrechnern /28/, /29/, 40/, /11/. C unterstützt die Entwicklung gut strukturierter und portabler Software und bietet eine leistungsfähige Schnittstelle zur Hardware sowie geeignete Mittel zur Entwicklung kodeeffizienter Programme. Diese Stärken kommen insbesondere bei der Implementation von Systemsoftware (darunter sind neben Betriebssystemkomponenten auch Compiler, Dienstprogramme, Textverarbeitungssysteme u.a. zu verstehen) zum Tragen. Darüber hinaus ist die Sprache für viele andere Anwendungsgebiete einsetzbar.

Der überwiegende Teil des Betriebssystems UNIX wurde in der Sprache C implementiert. Dies stellt eine ausgezeichnete Basis für den Transport des Systems auf verschiedene Rechnerarchitekturen sowie die Entwicklung einer Reihe von UNIX-kompatiblen Systemen dar.

Dieser Band der Fachbuchreihe TECHNISCHE INFORMATIK kommt der Forderung eines breiten Anwenderkreises nach, eine umfassende und methodisch aufbereitete Darstellung der sprachlichen Konzepte und Mittel von C sowie eine geeignete Programmiermethodik bereitzustellen. Interesse daran besteht nicht nur bei System- und Anwendungsprogrammierern, sondern auch bei Ingenieuren und Studenten unterschiedlichster Fach- und Anwendungsgebiete. Das Buch ist als Lehrtext konzipiert und ermöglicht das Erlernen der Programmiersprache C im Selbststudium. Dabei wird angenommen, dass der Leser mit den grundlegenden Begriffen und Techniken der Programmierung vertraut ist. Wünschenswert, aber nicht notwendig ist es, wenn er praktische Erfahrungen bei der Anwendung einer der "konventionellen" Programmiersprachen (z.B. PASCAL oder FORTRAN) besitzt. Dem erfahrenen Programmierer dient dieser Band als Nachschlagewerk, da er hier den vollständigen Sprachumfang entsprechend des X/OPEN-Standards /50/ sowie die Beschreibung der Funktionen der Standardbibliothek vorfindet.

An der Erarbeitung des Manuskriptes haben direkt oder indirekt viele Kollegen Anteil. Unser Dank gilt insbesondere Prof. Dr. sc. nat. K.Mätzel, der uns zu diesem Vorhaben ermutigte und das Projekt grosszügig unterstützte und förderte. Den Mitarbeitern der Sektion Informatik der TU Karl-Marx-Stadt, die zur Erarbeitung des Buches durch Anregungen und kritische Hinweise beigetragen haben, sei ausdrücklich gedankt, ebenso den Kolleginnen, die die Texteingabe des Manuskriptes vornahmen.

Die Erarbeitung des Manuskriptes und die Herstellung der Druckvorlage erfolgte mit Hilfe eines Textverarbeitungssystems. Die Beispiele wurden mit verschiedenen C-Compilern und dem Prüfprogramm lint getestet. Trotzdem sind sowohl Sach- als auch Druckfehler nicht auszuschliessen. Für diesbezügliche Anmerkungen und kritische Hinweise an den VEB Verlag Technik oder direkt an die

Technische Universität Karl-Marx-Stadt
Sektion Informatik
Strasse der Nationen 62
9010 Karl-Marx-Stadt

sind die Autoren dankbar.

Es ist uns ein Bedürfnis, dem Herausgeber dieser Fachbuchreihe Herrn Dr. G. Paulin und dem Lektor des VEB Verlag Technik Herrn J. Reichenbach für die wertvollen fachlichen und methodischen Hinweise sowie für die konstruktive Zusammenarbeit zu danken.

Matthias Clauss Günther Fischer

# Inhaltsverzeichnis

# 0. Einführung

Hinsichtlich des Umfangs und der sprachlichen Möglichkeiten ist C eine relativ "kleine" Sprache. Beispielsweise sind weder E/A-Anweisungen Bestandteil von C noch existieren Mittel zur Zeichenkettenverarbeitung. In den meisten C-Programmierumgebungen stehen jedoch viele dieser Dinge in Form von Bibliotheks- bzw. Systemfunktionen bereit. Diese Beschränkung im Sprachumfang ist eine wesentliche Ursache dafür, dass C mit relativ geringem Aufwand implementiert werden kann, insbesondere auch auf Mikro- und Kleinrechnern.

In gewissem Sinne stellt C einen Kompromiss zwischen einer typischen höheren Programmiersprache (wie beispielsweise PASCAL) und der Assemblersprache der jeweiligen Maschine dar. Auf der einen Seite wird die Entwicklung gut strukturierter und portabler Software unterstützt, andererseits bietet C eine leistungsfähige Schnittstelle zur Hardware sowie vielfältige und äusserst flexibel verwendbare Mittel zur Entwicklung kompakter sowie kode- und laufzeiteffizienter Programme. Während die Lösung insbesondere systemnaher Aufgabenstellungen in anderen Programmiersprachen zum Teil erhebliche "Klimmzüge" erfordert, um durch die Sprache vorgegebene Restriktionen zu umgehen, ist C durch Freizügigkeit bei der Wahl algorithmischer Strukturen und beim Zugriff auf Datenobjekte gekennzeichnet. Diese Entwurfsstrategie bedingt aber auch, dass dem Programmierer ein hohes Mass an Verantwortung überlassen bleibt. Vor allem dem Neuling wird empfohlen, die zur Verfügung stehenden sprachlichen Mittel sehr sorgsam und gezielt einzusetzen. Ein disziplinierter Programmierstil und Erfahrungen im Umgang mit C sind notwendig, da andernfalls die Gefahr besteht, dass fehleranfällige und wenig übersichtliche Programme entstehen. Obwohl man die Sprache C relativ leicht erlernen kann, ist sie aus obengenannten Gründen nur bedingt für den Anfänger auf dem Gebiet der Programmierung geeignet.

C befindet sich seit dem Zeitpunkt des Entwurfes in einer ständigen Weiterentwicklung. Lange Zeit galt der in /34/ beschriebene Sprachumfang als De-facto-Sprachstandard. Viele der verfügbaren Compiler berücksichtigen nachträgliche Erweiterungen, wie z.B. den Aufzählungstyp (enum) und die Strukturzuweisung. Grundlage für den in diesem Buch gewählten Sprachumfang bildet die im Teil 3 des X/OPEN Portability Guide /50/ veröffentlichte C-Sprachdefinition. Sie entspricht dem C Reference Manual der Version UNIX V, Release 2.0 /2/.

Im ersten Teil des Buches wird eine Einführung in die grundlegenden Sprachelemente von C gegeben. Behandelt werden die Grunddatentypen und ihre Realisierung als variable und konstante Werte, die einfachen arithmetischen Operatoren und Ausdrücke sowie die elementaren Steuerstrukturen, soweit sie mit einer vergleichbaren Semantik auch in anderen Programmiersprachen existieren. Dieser Teil des Buches enthält auch die Erläuterung der Grundlagen zu Funktionen und zur Programmstruktur. Auf dieser Basis werden einfache, aber vollständige Beispielprogramme gezeigt und somit alle notwendigen Kenntnisse vermittelt, um möglichst schnell selbst praktische Programmierübungen vornehmen zu können.

Teil 2 des Buches behandelt die für C charakteristischen Sprachaspekte. In den ersten Abschnitten werden die vielfältigen Operatoren erklärt, die strukturierten Datentypen und die Arbeit mit Zeigern erläutert sowie

die sprachlichen Mittel zur Programmlaufsteuerung vervollständigt. Der Abschnitt über Funktionen fasst alle damit im Zusammenhang stehenden Details zusammen. Die restlichen Abschnitte sind speziellen Fragen gewidmet, z.B. den Problemen der Speicherklasse von Variablen, Funktionen und Parametern, den Möglichkeiten der Initialisierung von Variablen sowie den Aspekten der Typkonvertierung. Die Diskussion dieser Probleme wird so lange wie möglich vertagt, um alle damit verbundenen Gesichtspunkte im Zusammenhang erläutern zu können. Abschliessend werden einige etwas komplexere Beispiele zu ausgewählten Aufgabenstellungen vorgestellt und diskutiert.

Teil 3 ist den Fragen der Anwendung von C gewidmet. Zunächst werden die Konventionen vorgestellt, nach denen der Aufruf von C-Programmen erfolgt. Zu jedem C-Compilersystem gehört ein Präprozessor, der - unabhängig von der eigentlichen Sprache - u.a. Möglichkeiten zur Bildung von Makros und zur bedingten Compilierung bietet. In einem weiteren Abschnitt erfolgt die Beschreibung wesentlicher Funktionen der Standardbibliothek. Hieran schliesst sich die Diskussion ausgewählter Fragen der systemnahen Programmierung in einer konkreten Programmierumgebung, dem Betriebssystem UNIX, an. Das betrifft z.B. die Anwendung der Systemfunktionen zur Ein- und Ausgabe und zur Prozesssteuerung. Abgerundet wird Teil 3 mit einer Reihe von Programmbeispielen zu unterschiedlichen Aufgabenstellungen. Dadurch sollen verschiedene Aspekte der C-Programmierung verdeutlicht und so zur Beherrschung der sprachlichen Mittel beigetragen werden.

Für das Erlernen der Programmiersprache C wird empfohlen, die einzelnen Abschnitte bis einschliesslich der Behandlung der Standardbibliothek sequentiell zu studieren und durch praktische Übungen zu ergänzen. Mit den Problemen der systemnahen Programmierung und den abschliessenden Beispielen kann sich der Leser in beliebiger Reihenfolge und je nach Interesse auseinandersetzen.

Dem C-Programmierer steht in den meisten Programmierumgebungen ein umfangreicher Satz von Bibliotheksfunktionen zur Verfügung. Anhang B enthält eine Übersicht über die zur Standardbibliothek gehörenden Funktionen und Makros. Darüberhinaus gibt es im Betriebssystem UNIX zur Unterstützung der Softwareentwicklung eine Reihe spezieller Werkzeuge, z.B.

- das C-Prüfprogramm lint /21/
- das Programm make zur Wartung von Programmsystemen /13/
- das Source Code Control System (/45/).

Da lint eine intensivere Prüfung der C-Quelltexte auf potentielle Fehlerquellen durchführt als die meisten C-Compiler, ist dieses Werkzeug auch für den Neuling sehr hilfreich. Anhang C enthält neben Hinweisen zur Anwendung eine Zusammenstellung der wesentlichen Fähigkeiten dieses Programms.

# 1. Grundlagen

Im ersten Teil sollen grundsätzliche Aspekte der Sprache C anhand einfacher Beispiele gezeigt werden. Dabei stehen Sprachelemente im Mittelpunkt, die in analoger Form in anderen höheren Programmiersprachen enthalten sind. Sie bilden die Grundlage für die Diskussion der speziellen Möglichkeiten, die dem C-Programmierer zur Verfügung stehen.

## 1.1. Programmstruktur

```
main()                  /* Ausgabe: Mein erstes C-Programm! */
{
    printf("Mein erstes C-Programm!\n");
}
```

Ein **Programm** in C besteht aus einer Menge von Funktionen, die in einem oder in mehreren Quelltextfiles untergebracht sein können. In unserem ersten Beispiel besteht das Programm aus einer **Funktion** mit dem Namen main. In jedem C-Programm muss eine Funktion mit diesem Namen vorhanden sein, da hier die Ausführung beginnt. Nach dem Funktionsnamen folgt, in runde Klammern eingeschlossen, die Liste der **formalen Parameter**. Sie werden voneinander durch Komma getrennt. Besitzt eine Funktion keine Parameter, wie in diesem Beispiel die main-Funktion, so wird durch () eine leere Parameterliste gekennzeichnet.

Alle zu einer Funktion gehörenden Anweisungen müssen in geschweifte Klammern eingeschlossen werden. Jede einzelne **Anweisung** wird durch ein Semikolon beendet. Unser Programm enthält nur eine Anweisung, den Aufruf der Funktion printf. Als **Argument** wird printf eine **Zeichenkettenkonstante**, d.h. eine in Anführungszeichen eingeschlossene Zeichenfolge übergeben. Allgemein erfolgt ein **Funktionsaufruf** durch Angabe des Funktionsnamens, dem in runde Klammern eingeschlossen die **Argumentliste** folgt. Die runden Klammern muss man auch dann angeben, wenn an die Funktion keine Argumente übergeben werden.

Bei der Funktion printf handelt es sich um eine Bibliotheksfunktion zur formatierten Ausgabe von Daten. In diesem Zusammenhang soll darauf hingewiesen werden, dass zur Sprache selbst keinerlei Anweisungen zur Ein- und Ausgabe gehören. Dafür stehen leistungsfähige Funktionen in der Standardbibliothek zur Verfügung. Obwohl eine zusammenfassende Erläuterung der E/A-Funktionen erst im Abschn. 3. erfolgt, werden in den weiteren Beispielen bereits verschiedene dieser Funktionen verwendet.

Das obige Programm enthält einen **Kommentar**, der durch die Zeichenkombinationen /* und */ geklammert wird. Ein Kommentar darf überall dort stehen, wo auch ein Leerzeichen stehen kann. Er kann sich über mehrere Zeilen erstrecken, jedoch dürfen Kommentare nicht verschachtelt sein.

Die Aufgabe des Programms ist es, den Text

```
Mein erstes C-Programm!
```

auszugeben. Die Zeichenfolge \n steht für das Steuerzeichen Newline. Es bewirkt den Übergang auf eine neue Zeile nach der Ausgabe des Textes. Ohne

die Zeichenfolge \n würde die nächste Ausgabe in der gleichen Zeile fortgesetzt. Folgende Version des Programms hat also die gleiche Wirkung:

```
main()                  /* Ausgabe: Mein erstes C-Programm! */
{
    printf("Mein erstes ");
    printf("C-Programm!\n");
}
```

## 1.2. Grunddatentypen

Unter einem Datentyp versteht man eine Menge von Werten und eine Menge von Operationen, die auf diese Werte angewendet werden können /19/.

Das Typkonzept der Sprache C bietet Voraussetzungen für einen direkten und effektiven Zugriff auf die zugrunde liegende Maschine. Als Grunddatentypen gibt es den Zeichentyp, verschiedene Integer- und Realtypen, den Aufzählungstyp sowie den Typ für die leere Wertemenge. Weiterhin existieren Möglichkeiten, zusammengesetzte Datentypen zu bilden, z.B. Felder, Strukturen und Zeiger. Darauf wird in den Abschnitten 2.2. und 2.3. eingegangen.

### 1.2.1. Variable

In der Sprache C werden Variablen - als symbolische Repräsentation von Speicherplatz - durch ihren **Typ** und ihre **Speicherklasse** beschrieben. Eine Variable muss vor ihrer Benutzung vereinbart werden. **Vereinbarungen** bestehen aus einem Speicherklassenattribut und/oder einer Typspezifikation sowie einer Liste von **Bezeichnern**. Jeder Bezeichner legt den Namen einer Variablen fest.

```
int x;              /* x ist eine Variable für ganze Zahlen */
float y, z;         /* y und z sind Variablen für reelle Zahlen */
```

Die folgende Notation soll keine exakte Syntaxbeschreibung für Variablenvereinbarungen darstellen. Sie wird hier und an weiteren Stellen im Text gewählt, um die allgemein benutzte Form einer C-Sprachkonstruktion hervorzuheben. In spitze Klammern eingeschlossene Begriffe dienen dabei als "Platzhalter" für im konkreten Fall einzusetzende syntaktische Kategorien, wie Schlüsselwörter, Variablenbezeichner, Ausdrücke usw. Durch die Wahl der Begriffe soll deren intuitive Bedeutung ausgedrückt werden. Die Zeichenfolge "..." deutet die Möglichkeit der Wiederholung an. Alle anderen Zeichen sind sog. Terminalsymbole und müssen unverändert angegeben werden.

Die allgemeine Form einer Variablenvereinbarung lautet:

<speicherklasse> <typ> <bezeichner$_1$>, ..., <bezeichner$_n$>;

Mit dem Speicherklassenattribut kann man Festlegungen über die Sichtbarkeit und Lebensdauer der Variablen treffen. In den meisten Fällen ergibt sich die Speicherklasse einer Variablen implizit aus der Stellung der Variablenvereinbarung. Häufig kann deshalb die Speicherklassenangabe weggelassen werden.

An dieser Stelle soll auf einen wesentlichen Unterschied in der Begriffsverwendung aufmerksam gemacht werden. Wenn von einer **Vereinbarung**

die Rede ist, bleibt offen, ob mit der Spezifikation von Typ und Speicherklasse auch die Vergabe von Speicherplatz verbunden ist. Während bei der **Deklaration** nur die Attribute festgelegt werden, schliesst die **Definition** die Bereitstellung von Speicherplatz ein. In Zusammenhängen, wo der Aspekt der Speicherplatzvergabe von Bedeutung ist, werden deshalb im weiteren die Begriffe Definition bzw. Deklaration verwendet.

Bezeichner können aus Buchstaben, dem Zeichen _ (Unterstreichungszeichen) und Ziffern bestehen. Das erste Zeichen muss ein Buchstabe sein, wobei _ als Buchstabe gilt. Es wird zwischen Gross- und Kleinbuchstaben unterschieden. Allgemein ist es jedoch üblich, für Bezeichner ausschliesslich Kleinbuchstaben zu verwenden, um sie leicht von symbolischen Konstanten unterscheiden zu können. Bezeichner sollten so gebildet werden, dass sie zur Eigendokumentation des Programms beitragen, z.B.

```
konto_nummer
betr_teil
```

Zwar existiert keine Beschränkung, wie lang Bezeichner sein dürfen, es sind aber nur die ersten n Zeichen (n >= 8) signifikant.

**Schlüsselwörter** sind reserviert und dürfen nicht als Bezeichner verwendet werden (Bild 1.1).

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

Bild 1.1. Schlüsselwörter

Zu den Grunddatentypen gehören der Zeichentyp **char**, die Integertypen **int, short, long, unsigned** sowie die Realtypen **float** und **double**.

Variablen des Zeichentyps können Zeichen aus dem implementierten Zeichensatz und andere Bitkombinationen aufnehmen. Integervariable dienen zur Speicherung ganzer Zahlen. Länge und damit Wertebereich hängen von der jeweiligen Maschinenarchitektur ab (Bild 1.2), wobei hinsichtlich der Länge L gilt: L(**short**) <= L(**int**) <= L(**long**).

| Typ | 16-Bit-Rechner | 32-Bit-Rechner |
|---|---|---|
| **char** | 8 Bit | 8 Bit |
| **int** | 16 | 32 |
| **short** | 16 | 16 |
| **long** | 32 | 32 |
| **float** | 32 | 32 |
| **double** | 64 | 64 |

Bild 1.2. Speicherplatzbedarf

Eine 16-Bit-Integerzahl kann somit im Bereich von -32768 bis 32767 liegen, eine Integervariable vom Typ **long** liegt im Intervall $[-2^{31}, 2^{31}-1]$. Reelle

Zahlen werden in Variablen der Realtypen **float** (einfache Genauigkeit) bzw. **double** (doppelte Genauigkeit) gespeichert. Wertebereich und Genauigkeit der Realtypen sind maschinen- und implementationsabhängig. Der Integertyp **unsigned** ist eine Besonderheit von C. Eine Variable dieses Typs wird als vorzeichenlose ganze Zahl betrachtet. Das erweist sich in der systemnahen Programmierung oft als nützlich. Als Typspezifikationen sind in einer Vereinbarung verschiedene Kombinationen möglich:

```
short int           /* gleichbedeutend mit short */
long int            /* gleichbedeutend mit long */
unsigned int        /* gleichbedeutend mit unsigned */
long float          /* gleichbedeutend mit double */
unsigned short
unsigned short int  /* gleichbedeutend mit unsigned  short */
unsigned long
unsigned long int   /* gleichbedeutend mit unsigned long */
unsigned char
```

Es muss darauf hingewiesen werden, dass einige Compilerimplementationen nicht alle der hier aufgeführten Typspezifikationen realisieren, das trifft z.B. **unsigned char** und **long float**. Abschliessend folgen einige Beispiele für Variablenvereinbarungen:

```
short int a1;       /* a1 repräsentiert einen 16-Bit-Integerwert */
short a2;           /* ebenso die Variable a2 */
int i, j;           /* zwei "normale" Integervariable i und j */
long  k, l, m;      /* Wertebereich von k, l und m : [-2^31, 2^31-1] */
unsigned anz;       /* Wertebereich von anz : [0, 2^16-1 bzw. 2^32-1] */
char c;             /* für Zeichen des betreffenden Zeichensatzes */
float x;            /* einfach genaue Realvariable */
double y, z;        /* doppelt genaue Realvariable */
```

Im weiteren werden auch die Begriffe **integrale** und **arithmetische** Typen verwendet. Den Zeichentyp, die Integertypen und den Aufzählungstyp (s. Abschn. 2.2.5.) rechnet man zu den integralen Typen. Zu den arithmetischen Typen gehören die integralen Typen und die Realtypen.

### 1.2.2. Konstanten

C erlaubt die Verwendung von Integer-, Real-, Zeichen- und Zeichenkettenkonstanten.

**Integerkonstanten**

Integerkonstanten können als Dezimalkonstanten (Basis 10), Oktalkonstanten (Basis 8) oder Hexadezimalkonstanten (Basis 16) vorkommen. Standardmässig ist ihr Typ **int**, wenn nicht der zulässige Wertebereich über- bzw. unterschritten wird. In diesem Fall wird der Typ **long** angenommen, der auch explizit durch das nachgestellte Alphazeichen L oder l gefordert werden kann. Allgemein besteht eine Integerkonstante aus einer Ziffernfolge, wobei die zulässigen Ziffern von der Basis abhängen. Für die Dezimaldarstellung sind das die Zeichen 0 bis 9, für die Oktaldarstellung 0 bis 7 und für die Hexadezimaldarstellung 0 bis 9 und a bis f bzw. A bis F, d.h., die Werte 10 bis 15 werden durch die Buchstaben a bis f bzw. A bis F dargestellt. Steht vor der ersten signifikanten Ziffer eine 0, so wird die Ziffernfolge als Oktalkonstante interpretiert. Beginnt die Konstante mit der Zeichenfolge 0X oder 0x, so handelt es sich um eine Hexadezimalkonstante.

In den folgenden Beispielen sind als Kommentar Typ und äquivalente Darstellungen angegeben:

```
Dezimalkonstanten:        100         /*  int;  0144, 0x64  */
                          5L          /*  long;  051, 0x51  */
                          12843       /*  long;  037253, 0x1f42b  */

Oktalkonstanten:          020         /*  int;  16, 0x10  */
                          01000L      /*  long;  512, 0x200  */
                          0177        /*  int;  127, 0x7f  */

Hexadezimalkonstanten:    0X15        /*  int;  21, 025  */
                          0xFF        /*  int;  255, 0377  */
                          0x1ffff1    /*  long;  131071, 0377777  */
```

**Realkonstanten**

Realkonstanten bestehen aus einem ganzzahligen Anteil, einem Dezimalpunkt, einem gebrochenen Anteil, dem Zeichen E oder e und einem eventuell vorzeichenbehafteten Exponenten. Ganzzahliger und gebrochener Anteil sowie der Exponent stellen Folgen von Dezimalziffern dar. Verschiedene dieser Angaben können weggelassen werden. Die folgenden Beispiele verdeutlichen die möglichen Notationen:

```
17631.0e-7      /*  17631E-7  */
1E-10           /*  0.0000000001  */
1.              /*  1.0  */
-3.25           /*  -325.e-2  */
1.E+12          /*  1e12  */
.78             /*  .078E1  */
.2e3            /*  200.  */
```

Realkonstanten sind grundsätzlich vom Typ **double**.

**Zeichenkonstanten**

Einzelne Zeichen werden durch Einschluss in Apostrophe dargestellt. Der numerische Wert einer Zeichenkonstanten hängt vom verwendeten Zeichensatz ab, z.B. entspricht im ASCII-Zeichensatz die Zeichenkonstante 'a' der Integerkonstanten 0x61. Zeichenkonstanten besitzen den Typ **int**.

```
'0'             /*  Ziffer Null als Zeichenkonstante  */
'*'             /*  das Zeichen *  (Stern)  */
'"'             /*  Anführungszeichen  */
```

Nichtdarstellbare Steuerzeichen, die Zeichen Apostroph und Backslash sowie beliebige Bitkombinationen können wie folgt angegeben werden:

```
'\0'            /*  Nullzeichen (oder kurz: NUL)  */
'\n'            /*  Newline-Zeichen (newline)  */
'\t'            /*  Tabulator-Zeichen (tabulator)  */
'\b'            /*  Backspace-Zeichen (backspace)  */
'\f'            /*  Seitenvorschub-Zeichen (formfeed)  */
'\r'            /*  Wagenrücklauf-Zeichen (carriage return)  */
'\v'            /*  Vertikal-Tabulator  */
'\''            /*  Apostroph  */
'\\'            /*  Backslash-Zeichen (backslash)  */
'\nnn'          /*  Bitmuster (nnn ist Folge von 1-3 Oktalziffern)  */
```

In allen Fällen wird nur **ein** Zeichen dargestellt. Folgt dem Zeichen \ in der Zeichenkonstanten ein Zeichen, das nicht in der Aufstellung angegeben

wurde, so ist das Verhalten des Compilers laut /50/ undefiniert. Die Notationsmöglichkeiten von Zeichenkonstanten unterstützen die Lesbarkeit und Portabilität von Programmen, da die konkrete numerische Verschlüsselung eines Zeichens nicht bekannt sein muss.

**Zeichenkettenkonstanten**

Eine Zeichenkettenkonstante ist eine in Anführungszeichen eingeschlossene Folge von Null oder mehreren Zeichen. Sie besitzt den Typ "Zeichenfeld", d.h., vom Compiler wird ein eindimensionales Feld von Zeichen erzeugt und mit \Ø abgeschlossen (s. Abschn. 2.2.1.). Demzufolge ist eine Zeichenkettenkonstante immer um ein Zeichen länger als tatsächlich angegeben. Das Zeichen \Ø wird zur Endeerkennung benutzt. Nichtdarstellbare Steuerzeichen, das Anführungszeichen, das Zeichen Backslash und beliebige Bitkombinationen können wie in Zeichenkonstanten notiert werden.

```
"int"
"Mein erstes C-Programm!\n"
"A"                          /* nicht zu verwechseln mit 'A' */
"\"abc\""                    /* Zeichenfolge "abc" */
""                           /* "leere" Zeichenkette */
```

Während die Zeichenkonstante 'A' nur aus dem Buchstaben A besteht, stellt die Zeichenkettenkonstante "A" eine Folge von zwei Zeichen dar: A und das Zeichen \Ø. Das Anführungszeichen selbst ist innerhalb einer Zeichenkettenkonstanten als \" darstellbar. Die Notation "" repräsentiert eine leere Zeichenkettenkonstante. Sie besteht aus einem Byte mit dem Nullzeichen \Ø.

**Symbolische Konstanten**

Häufig werden in C-Quelltexten symbolische Konstanten verwendet, um die Lesbarkeit von Programmen zu unterstützen. Die Vereinbarung einer symbolischen Konstanten kann mittels **#define** erfolgen.

```
#define     ZINS    3.25
#define     EOF     (-1)
#define     NL      '\n'
```

Beginnt eine Quelltextzeile mit der Zeichenfolge **#define**, so wird sie von einem Präprozessor (vor der eigentlichen Compilierung) ausgewertet. Symbolische Konstanten sind genaugenommen kein Bestandteil der Sprache. Die allgemeine Form der Definition einer symbolischen Konstanten lautet:

**#define** <bezeichner> <zeichenfolge>

Der Präprozessor ersetzt die symbolische Konstante <bezeichner> bei jedem Auftreten im nachfolgenden Quelltext durch <zeichenfolge>.

Die Regeln zur Bildung von Namen für symbolische Konstanten sind analog zu denen für Variablenbezeichner. Es ist allgemeine Praxis, Konstantennamen als Folge von Grossbuchstaben darzustellen. Als Ersetzungstext kann eine beliebige Zeichenfolge stehen. Treten symbolische Konstanten innerhalb von Kommentaren, in Zeichenkonstanten oder in Zeichenkettenkonstanten auf, so wird an diesen Stellen keine Ersetzung vorgenommen. Arbeitsweise und weitere Funktionen des Präprozessors werden im Abschn. 3.2. ausführlich beschrieben.

## 1.3. Ausdrücke und Operatoren

**Ausdrücke** bestehen aus **Operanden** und **Operatoren**. Die Operanden sind Variablen, Konstanten oder wiederum Ausdrücke. Die Auswertung jedes Ausdrucks liefert einen Wert, der sich aus der Verknüpfung von Operanden durch Operatoren ergibt.

Im folgenden werden die elementaren Operatoren vorgestellt. Auf spezifische Operatoren der Sprache C, solche Aspekte wie Vorrang und Assoziativität von Operatoren sowie die Typkonvertierung in Ausdrücken gehen wir im Teil 2 ein.

### 1.3.1. Wertzuweisung

Die Wertzuweisung ist ein Ausdruck. Sie stellt die übliche Methode dar, einer Variablen einen bestimmten Wert zuzuordnen.

```
x = 0
y = x + 4
c = getchar()
```

Der rechts vom Operator = stehende Ausdruck wird berechnet und sein Wert der Variablen auf der linken Seite zugewiesen. Die allgemeine Form einer Wertzuweisung lautet:

```
<lvalue> = <ausdruck>
```

Dabei ist unter <lvalue> ein Ausdruck zu verstehen, der sich auf einen modifizierbaren Hauptspeicherbereich bezieht und durch einen Typ gekennzeichnet ist. Das einfachste Beispiel hierfür ist ein Variablenbezeichner.

Im Gegensatz zu anderen Programmiersprachen besitzt eine Wertzuweisung wie jeder andere Ausdruck stets einen Wert. Aus diesem Grunde sind **Mehrfachzuweisungen** wie z.B.

```
a = b = c = 1
```

erlaubt, die von rechts nach links berechnet werden, d.h.

```
a = (b = (c = 1))
```

Der Teilausdruck

```
c = 1
```

besitzt den Wert 1, der b zugewiesen wird usw. Allgemein gilt, dass der Wertzuweisungsoperator in einem Ausdruck wie jeder andere Operator benutzt werden kann.

In einem Ausdruck

```
(c = getchar()) == '\n'
```

liefert die Funktion getchar das nächste Zeichen. Das Zeichen wird nach c gespeichert und stellt den Wert dieses Teilausdrucks dar. Anschliessend wird geprüft, ob es sich bei diesem Wert um das Zeichen Newline ('\n') handelt. Derartige Konstruktionen tragen zu einer kompakteren Schreibweise bei. Ausserdem müssen logisch zusammenhängende Dinge nicht unnötig zerpflückt werden.

### 1.3.2. Arithmetische Operatoren

Als elementare arithmetische Operatoren gibt es den unären Operator

```
-                    /* negatives Vorzeichen */
```

und die binären Operatoren

```
+, -                 /* Addition, Subtraktion */
*, /                 /* Multiplikation, Division */
%                    /* Rest der ganzzahligen Division */
```

Bei der Division x/y von positiven Integerwerten x und y wird der gebrochene Teil abgeschnitten. Ist einer der Operanden negativ, so ist die Form des Abschneidens maschinenabhängig. Der Divisionsrest kann durch x%y ermittelt werden. Welches Vorzeichen der Rest besitzt, hängt vom Maschinentyp ab. Der Operator % ist nur für Operanden mit integralem Typ erlaubt.

### 1.3.3. Vergleichsoperatoren

Die Sprache C besitzt die Vergleichsoperatoren

```
>, >=                /* grösser, grösser gleich */
<, <=                /* kleiner, kleiner gleich */
==                   /* gleich */
!=                   /* ungleich */
```

Per Definition besitzt ein Vergleichsausdruck der Form

<ausdruck$_1$> <rel_op> <ausdruck$_2$>

den Wert 1, wenn die durch <rel_op> spezifizierte Bedingung erfüllt ist, ansonsten den Wert Ø. Es sei z.B. folgende Anweisungsfolge gegeben:

```
int x, y, schalter;
   ...
x = 4;
y = 2;
schalter = x > y;
```

Im Ergebnis der letzten Anweisung wird der Variablen schalter der Wert 1 zugewiesen.

Ungewöhnlich ist die Notation des Vergleichsoperators ==, der sorgfältig von dem Wertzuweisungsoperator = unterschieden werden muss. Eine häufige Fehlerquelle insbesondere bei Neulingen besteht darin, statt des beabsichtigten Vergleichsoperators den Zuweisungsoperator zu schreiben. Da bei einer Verwechslung syntaktisch korrekte Ausdrücke entstehen, ist die Ursache für das Fehlverhalten des Programms mitunter schwer zu entdecken. Das trifft z.B. zu, wenn man anstelle von

```
if(a == 1) ...
```

irrtümlich den syntaktisch zwar zulässigen, aber sicherlich nicht sinnvollen Text

```
if(a = 1) ...
```

notiert.

## 1.4. Ablaufsteuerung

Die Möglichkeiten zur Steuerung des Programmablaufs korrespondieren mit sprachlichen Mitteln anderer Programmiersprachen. Dieser Abschnitt geht dabei nur auf die elementaren Steueranweisungen (**if-else**, **while**, **do-while**) ein. Fortgesetzt wird diese Diskussion im Abschn. 2.4. mit den Anweisungen **switch**, **for**, **break**, **continue** und **goto**.

### 1.4.1. Einfache Anweisung

Eine **einfache Anweisung** (expression statement) besteht aus einem Ausdruck, dem ein Semikolon folgt:

```
<ausdruck>;
```

Gewöhnlich handelt es sich bei <ausdruck> um eine Wertzuweisung oder einen Funktionsaufruf:

```
bytes = Ø;
printf("Mein erstes C-Programm!\n");
schalter = x > y;
```

Der Trivialfall einer einfachen Anweisung ist die leere Anweisung. Sie besteht nur aus einem Semikolon

```
;
```

und ist (wie wir später noch in konkreten Beispielen sehen werden) in bestimmten Situationen erforderlich.

### 1.4.2. Block

Mit Hilfe geschweifter Klammern werden Vereinbarungen und Anweisungen zu einem **Block** (auch **Verbundanweisung** genannt) zusammengefasst. Zum Beispiel bilden alle zu einer Funktion gehörenden Variablenvereinbarungen und Anweisungen einen Block, den sog. Funktionsblock:

```
main()
{
    int i, n;
    ...
    i = Ø;
    n = i + 1;
    ...
}
```

An jeder Stelle, wo eine Anweisung stehen darf, kann auch ein Block stehen. Der schliessenden geschweiften Klammer folgt kein Semikolon. Blöcke können beliebig geschachtelt werden. Die Vereinbarung von Variablen ist nur am Blockanfang vor der ersten Anweisung möglich. Eine in einem Block

vereinbarte Variable ist ausserhalb des Blocks nicht sichtbar, d.h., auf sie kann nur innerhalb dieses Blocks zugegriffen werden.

### 1.4.3. if-else-Anweisung

Zur Auswertung von Entscheidungen gibt es die **if-else**-Anweisung.

```
if(<ausdruck>)
    <anweisung1>
else
    <anweisung2>
```

Der auszuwertende Vergleichsausdruck <ausdruck> muss in runde Klammern eingeschlossen werden. Besitzt <ausdruck> den Wahrheitswert TRUE, so wird <anweisung$_1$> ausgeführt. Wenn die Auswertung des Vergleichsausdrucks den Wahrheitswert FALSE liefert, so erfolgt die Ausführung von <anweisung$_2$>.

Es sei an dieser Stelle nochmals darauf hingewiesen, dass es sich bei <anweisung$_1$> bzw. <anweisung$_2$> sowohl um einzelne Anweisungen als auch um Blöcke handeln kann.

Der **else**-Zweig kann auch fehlen:

```
if(<ausdruck>)
    <anweisung>
```

Zu beachten ist, dass in C kein spezieller **boolean**-Typ zur Darstellung von Wahrheitswerten existiert. Allgemein gilt, dass der Wahrheitswert eines Ausdrucks TRUE ist, falls sein numerischer Wert verschieden von Null ist, und FALSE bei Null.

```
if(z != Ø)
    y = x/z;
else
    printf("Division durch Ø\n");
```

Da nur geprüft wird, ob <ausdruck> einen Wert gleich oder ungleich Null besitzt, sind auch Kurzschreibweisen möglich, also

```
if(z)
    y = x/z;
else
    printf("Division durch Ø\n");
```

Beliebige Schachtelungen von **if-else-if**-Konstruktionen sind erlaubt. Ein eventuell folgender **else**-Zweig wird hierbei dem letzten **else**-losen **if**-Zweig zugeordnet. Ist es anders beabsichtigt, muss die gewünschte Abarbeitungsfolge durch gezielte Klammernsetzung erzwungen werden:

```
if(z != Ø)   {
    if(x != Ø)
        y = x/z;
} else
    printf("Division durch Ø\n");
```

Mit diesen Mitteln steht auch eine Form der Mehrwegeentscheidung zur Verfügung:

```
if(<ausdruck1>)
    <anweisung1>
else if(<ausdruck2>)
    <anweisung2>
   ...
else if(<ausdruckn-1>)
    <anweisungn-1>
else
    <anweisungn>
```

Die Vergleichsausdrücke werden der Reihe nach ausgewertet, bis ein Ausdruck gefunden wird, z.B. <ausdruck$_i$>, der den Wahrheitswert TRUE, d.h. einen Wert ungleich Null besitzt. In diesem Fall erfolgt die Ausführung des zugehörigen Anweisungsteils <anweisung$_i$>. Die Abarbeitung wird hinter der Gesamtkonstruktion fortgesetzt. Ist keine der Bedingungen erfüllt, so wird <anweisung$_n$> ausgeführt. Der letzte **else**-Zweig ist wiederum wahlweise.

### 1.4.4. while-Schleifen

Man unterscheidet zwei Varianten von **while**-Schleifen: **while** und **do-while**. Betrachten wir zunächst die allgemeine Form der **while**-Schleife:

```
while(<ausdruck>)
    <anweisung>
```

Hierbei erfolgt als erstes die Auswertung von <ausdruck>. Bei einem Wahrheitswert TRUE wird <anweisung> ausgeführt und anschliessend erneut der Vergleichsausdruck berechnet. Dies geschieht zyklisch so lange, bis die Auswertung von <ausdruck> einen Wahrheitswert FALSE ergibt und damit die Abarbeitung der **while**-Anweisung beendet ist. Besitzt <ausdruck> bereits beim ersten Durchlauf den Wahrheitswert FALSE, d.h., sein numerischer Wert ist gleich Null, so wird <anweisung> überhaupt nicht ausgeführt.

```
#define EOF (-1)

main()              /* Kopieren eines Files */
{
    int c;

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

Dieses Programm liest zyklisch ein Zeichen mit getchar und gibt dieses Zeichen sofort wieder aus. Das geschieht so lange, bis die Fileendebedingung EOF erkannt wird. Der Ausdruck

```
(c = getchar()) != EOF
```

muss vor der Anweisung

```
putchar(c);
```

ausgewertet werden, da das Eingabefile leer sein kann und bei EOF kein

Zeichen mehr ausgegeben werden darf. Es könnte in diesem Zusammenhang die Frage auftauchen, warum c nicht als **char**-Variable definiert ist, sondern den Typ **int** besitzt. Das liegt daran, dass getchar in der Lage sein muss, alle möglichen Zeichenwerte als Funktionsergebnis zurückzugeben und darüber hinaus EOF, d.h. den Integerwert -1.

Im Gegensatz zur **while**-Konstruktion wird bei der **do-while**-Schleife

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

die Bedingung <ausdruck> erst nach Ausführung von <anweisung> überprüft. Der Anweisungsteil einer **do-while**-Schleife wird also mindestens einmal abgearbeitet.

In den meisten Anwendungsfällen kann anstelle einer **do-while**-Anweisung auch eine **while**-Schleife verwendet werden. So auch in dem folgenden Programm, das ein File kopiert und dabei jedes Tabulatorzeichen durch eine festgelegte Anzahl von Leerzeichen ersetzt.

```
#define EOF (-1)
#define TABANZ 6

main()              /* Filekopieren mit Tabulatorauswertung */
{
    int c, i;

    while((c = getchar()) != EOF)  {
        if(c == '\t')  {
            i = Ø;
            do  {
                putchar(' ');
                i = i + 1;
            } while(i < TABANZ);
        } else
            putchar(c);
    }
}
```

Natürlich könnte der gleiche Effekt durch eine **while**-Schleife erreicht werden:

```
...
i = Ø;
while(i < TABANZ)  {
    putchar(' ');
    i = i + 1;
}
...
```

Weitere Möglichkeiten zur Schleifenbildung werden im Abschn. 2.4. behandelt.

## 1.5. Programmbeispiele

Der bis hierher dargelegte konventionelle Kern der Programmiersprache C bietet Voraussetzungen, um einige praktische Programmierübungen vornehmen zu können. Man muss hierbei nicht nur wissen, wie man ein Programm schreibt, sondern auch in der Lage sein, das Programm zur Ausführung zu

bringen. Wir gehen davon aus, dass sich der Programmtext bereits in einem Quelltextfile mit dem Namen prog.c befindet. Der Filename kann natürlich beliebig gewählt werden. Gefordert wird nur, dass er auf .c endet. Dann kann man im UNIX durch das Kommando

```
cc prog.c
```

erreichen, dass prog.c compiliert wird. Stellt der Compiler dabei keine Fehler fest, erfolgt darüber hinaus die Erzeugung eines ausführbaren Files mit dem Standardnamen a.out. Ein anschliessender Aufruf des erzeugten Programms durch das Kommando

```
a.out
```

bewirkt die Ausführung dieses Programms.

Etwas verallgemeinert kann das cc-Kommando wie folgt benutzt werden:

```
cc [-c] [-o <a_file>] <e_file1> [<e_file2> ... ]
```

Die in eckigen Klammern eingeschlossenen Aufrufparameter sind nur bei Bedarf erforderlich. Steht das Zeichen - als erstes Zeichen eines Parameters, so wird das nachfolgende Zeichen Flag genannt.

Das c-Flag gibt an, dass kein ausführbares File erzeugt werden soll, sondern nur die Objektfiles. Der Filename eines Objektfiles ergibt sich aus dem Filenamen des Quellfiles, wobei das Suffix .c durch .o ersetzt wird. Für unser obiges Beispiel erzeugt der Compileraufruf

```
cc -c prog.c
```

ein Objektfile prog.o.

Die Angabe -o <a_file> legt fest, dass das ausführbare File den Namen <a_file> erhalten soll. Soll das Programm z.B. unter dem Namen copy abgearbeitet werden, so ist folgender Compileraufruf erforderlich:

```
cc -o copy prog.c
```

Die Parameter <e_file1>, <e_file2> usw. geben die Namen der Eingabefiles an. Das können sowohl Quellfiles (mit dem Suffix .c) als auch Objektfiles (Suffix .o) sein. Somit lassen sich Programme, die aus mehreren Files bestehen, sehr einfach separat übersetzen und verbinden.

Wir gehen an dieser Stelle nicht weiter auf den Compileraufruf mittels cc ein. Eine detailierte Beschreibung der Möglichkeiten dieses Kommandos ist z.B. in /30/ enthalten.

Die folgenden Beispiele sollen als Anregung für praktische Experimente dienen. Sie helfen, den Umgang mit dem Compiler und dem Betriebssystem kennenzulernen sowie die bisher behandelten Sprachkonstruktionen zu beherrschen. Das erste Experiment kann durchaus das triviale Programm aus Abschn. 1.1. sein.

Soweit es für das Verständnis der Beispiele notwendig ist, werden wir in diesem Abschnitt auf einige bisher noch nicht diskutierte Aspekte eingehen. Das betrifft z.B. die **return**-Anweisung zur Rückgabe des Funktionsergebnisses, einige E/A-Funktionen der Standardbibliothek sowie einige Steueranweisungen des C-Präprozessors. Im weiteren werden diese Informationen benutzt, obwohl die ausführliche Behandlung erst zu einem späteren Zeitpunkt erfolgt.

### 1.5.1. Berechnung von Quadratwurzeln

Es sollen die Quadratwurzeln für die Werte z = 1, z = 2, ... , z = 5 berechnet und zeilenweise ausgegeben werden. Wir verwenden dazu die Iterationsformel

$$x_{i+1} = 1/2 * ( x_i + z/x_i ) \qquad \text{mit} \quad x_0 = z$$

und brechen die Iteration bei $| x_i - x_{i+1} | < EPSILON$ ab.

```
#include  <stdio.h>

#define  EPSILON  Ø.ØØ1
#define  FIRST  1.Ø
#define  LAST   5.Ø
#define  STEP   1.Ø

main()              /* Wurzelberechnung für die Werte 1.Ø - 5.Ø */
{
    float xneu;
    float xalt;
    float wert;
    float abs;

    printf("\n  Wurzelberechnung\n  ================\n");
    wert = FIRST;
    while(wert <= LAST) {
        xneu = wert;
        do {
            xalt = xneu;
            xneu = (xalt + wert / xalt) / 2.;
            if((abs = xalt - xneu) < 0)
                abs = -abs;
        } while(abs >= EPSILON);
        printf("  %.5f    %.5f\n", wert, xneu);
        wert = wert + STEP;
    }
}
```

Zunächst sei nochmals darauf hingewiesen, dass die Sprache C selbst keine Ein- und Ausgabeanweisungen enthält. Aus diesem Grunde wurde eine umfangreiche Standardbibliothek definiert und bereitgestellt /30/, /50/. Die erste Zeile des Programms ist eine Steuerzeile, die durch den Präprozessor ausgewertet wird. Sie bewirkt den Einschluss des Files stdio.h.[1] Das File stdio.h ist Bestandteil des Compilersystems und enthält die für die Nutzung der Funktionen erforderlichen Deklarationen. Die Bibliotheksfunktion printf wird zur Ausgabe einer Überschrift und zur formatierten Ausgabe der Ausgangs- und Ergebniswerte benutzt. Das erste Argument von printf ist eine Zeichenkette. Sie enthält **Konvertierungsvorschriften.** Eine Konvertierungsvorschrift wird mit dem Zeichen % eingeleitet und legt fest, in welchem Format jeweils eines der weiteren printf-Argumente ausgegeben werden soll. Die Konvertierungsvorschrift %.5f gibt an, dass die Ausgabe des Arguments wert als gebrochene Dezimalzahl mit 5 Stellen nach dem Dezimalpunkt erfolgen soll. Die zweite %.5f-Spezifikation bezieht sich in analoger Weise auf den Ergebniswert xneu. Ausführlich wird die Bibliotheksfunktion printf und deren Konvertierungsmöglichkeiten im Abschn. 3.3.

---

[1] Genaugenommen ist das in diesem Beispiel nicht unbedingt erforderlich. Dem Anfänger wird aber empfohlen, das File stdio.h grundsätzlich einzuschliessen.

erläutert. Anhang B enthält eine vollständige Beschreibung von printf.

Die Genauigkeit EPSILON, die Intervallgrenzen FIRST und LAST und die Schrittweite STEP werden als symbolische Konstanten definiert. Das Problem besteht darin, dass diese Werte damit im Programm "fest verdrahtet" sind. Jede beabsichtigte Änderung setzt eine Modifikation des Quelltextes sowie eine Neuübersetzung voraus. Viel günstiger wäre es z.B., Intervallgrenzen, Schrittweite und Genauigkeit während der Programmabarbeitung im Dialog angeben zu können. Auf derartige Möglichkeiten gehen wir im Abschn. 1.5.4. ein.

Die **while**-Schleife stellt alle Werte bereit, für die die Wurzel berechnet werden soll. In der **do-while**-Schleife erfolgt die iterative Berechnung der Quadratwurzel.

Der Quelltext sei in einem File wurzel.c enthalten. Bild 1.3 zeigt den Aufruf des Compilers und die Programmausführung am Bildschirm.

```
$ cc -o wurzel wurzel.c
  wurzel

  Wurzelberechnung
  ================
  1.00000     1.00000
  2.00000     1.41421
  3.00000     1.73205
  4.00000     2.00000
  5.00000     2.23606
$
```

Bild 1.3. Übersetzen und Aufruf eines C-Programms

Eingaben des Nutzers sind im Fettdruck dargestellt. Das Zeichen $ zeigt in UNIX-Systemen die Eingabebereitschaft des Systems an. Mit dem cc-Kommando wird das Quellfile wurzel.c compiliert und das ausführbare File wurzel erzeugt. Der Compiler liefert keine Programmliste und bei syntaktisch korrekten Quellfiles auch keinerlei Mitteilungen.

### 1.5.2. Kopieren von Files

Die zweite Aufgabe besteht darin, ein Programm zu schreiben, das ein File zeichenweise liest und diese Zeichen wieder ausgibt. Ausserdem soll die Anzahl der insgesamt gelesenen Zeichen und Blöcke ermittelt und ausgegeben werden.

Das Problem ist nicht kompliziert und könnte ohne Schwierigkeit in einer einzigen Funktion gelöst werden. Aus methodischen Gründen soll die Ein- und Ausgabe des Files sowie das Zählen der Zeichen in separate Funktionen verlagert werden. Ein Grund für diese Vorgehensweise besteht in dem Prinzip der Modularisierung, d.h., komplexere Problemstellungen in über-

schaubare Teilprobleme zu zerlegen. Funktionen bieten hierfür geeignete Möglichkeiten. Um eine Funktion benutzen zu können, muss lediglich bekannt sein, was die Funktion bewirkt, welche Funktionsargumente sie erwartet und welches Funktionsergebnis geliefert wird. Wie die Funktion ihre Aufgabe löst, ist für den Benutzer uninteressant. Eine Funktion sollte immer so entworfen werden, dass sie sich auf die Lösung **einer** Aufgabe konzentriert. Um so grösser ist die Wahrscheinlichkeit, dass diese Funktion auch in einem anderen Zusammenhang als ursprünglich vorgesehen Verwendung finden kann.

Zurück zum Filekopierprogramm, dessen Text folgendermassen aussieht:

```
#include  <stdio.h>

main()              /* Block- und Byteanzahl des kopierten Files */
{
    int bytes;
    int blocks;

    bytes = copy();
    blocks = bytes / BUFSIZ;
    if((bytes % BUFSIZ) > Ø)
        blocks = blocks + 1;
    fprintf(stderr, "%d %d\n", blocks, bytes);
}

int copy()          /* copy liefert Anzahl der kopierten Zeichen */
{
    int c;
    int bytes;

    bytes = Ø;
    while((c = getchar()) != EOF)  {
        putchar(c);
        bytes = bytes + 1;
    }
    return(bytes);
}
```

Das Programm besteht aus den Funktionen main und copy. Zu einem Programm gehörende Funktionen kann man über mehrere Quellfiles verteilen, jede Funktion muss jedoch vollständig in einem File enthalten sein. Funktionsdefinitionen dürfen nicht geschachtelt werden. Im obigen Beispiel gehen wir davon aus, dass sich beide Funktionen in einem File befinden. Andernfalls müsste die **include**-Steuerzeile zu Beginn eines jeden Quellfiles angegeben werden.

Die Zeile

```
int copy()          /* copy liefert Anzahl der kopierten Zeichen */
```

leitet die Definition der Funktion copy ein und gibt an, dass diese Funktion einen Ergebniswert vom Typ **int** liefert. (Das Typattribut könnte weggelassen werden, da standardmässig **int** als Typ für den Funktionswert angenommen wird.)

Eine Funktion kann durch eine Anweisung

```
return <ausdruck>;
```

einen Ergebniswert an die aufrufende Funktion zurückgeben. Gewöhnlich wird aus Gründen besserer Lesbarkeit <ausdruck> geklammert, obwohl dazu aus syntaktischer Sicht keine Notwendigkeit besteht:

```
return(<ausdruck>);
```

Im Beispiel liefert die Funktion copy die Anzahl der gelesenen Zeichen.

Die Standardbibliothek unterstützt drei Standardfiles, die der Programmierer ohne weitere Vorbereitungen nutzen kann und die normalerweise dem Nutzerterminal zugeordnet sind:

| | |
|---|---|
| Standardeingabe: | stdin |
| Standardausgabe: | stdout |
| Standardfehlerausgabe: | stderr |

Das Programm verwendet drei E/A-Funktionen der Standardbibliothek. Mittels getchar wird jeweils ein Zeichen von der Standardeingabe gelesen, und putchar gibt seinen Argumentwert in die Standardausgabe aus.

Die Wirkungsweise von fprintf entspricht der von printf, wobei das erste Argument angibt, wohin die Ausgabe erfolgen soll, nämlich auf die Standardfehlerausgabe stderr. Es ist sinnvoll, die Ausgabe der Zeichen- und Blockanzahl nicht mit der Standardausgabe zu vermischen. Übrigens sind im File stdio.h auch die symbolischen Konstanten BUFSIZ und EOF definiert.

Man beachte, dass die int-Variable bytes der Funktion main in keinerlei Beziehung zu der Variablen bytes der Funktion copy steht. Die Sichtbarkeit dieser Variablen erstreckt sich nur über den Funktionsblock, in dem sie vereinbart werden.

### 1.5.3. Modifikation des Filekopierens

Es folgen zwei Varianten des Filekopierprogramms. Zunächst soll jede Zeile des gelesenen Files mit einer Zeilennummer versehen und wieder ausgegeben werden. Das Problem ist im Grunde genommen trivial. Erschwert wird die Aufgabe nur deshalb, weil wir zur Zeit noch auf die zeichenweise Ein- und Ausgabe beschränkt sind.

```
#include <stdio.h>

#define YES 1
#define NO Ø

main()              /* File kopieren und Numerieren aller Zeilen */
{
    short int nl_anz;
    int c, zeile_nr;

    zeile_nr = 1;
    nl_anz = YES;
    while((c = getchar()) != EOF)  {
        if(nl_anz)  {
             printf("%6d  ", zeile_nr);
             zeile_nr = zeile_nr + 1;
        }
        if(c != '\n')
            nl_anz = NO;
        else
            nl_anz = YES;
        putchar(c);
    }
}
```

Der "Schalter" nl_anz steuert die Ausgabe der Zeilennummer am Anfang jeder Zeile. Der Test

```
if(nl_anz)
```

stellt die verkürzte Form von

```
if(nl_anz == YES)
```

dar. Das ist möglich, da die symbolische Konstante YES einen Wert ungleich Null besitzt. Da sowohl printf als auch putchar auf die Standardausgabe ausgeben, können Aufrufe beider Funktionen in beliebiger Folge gemischt auftreten. Die Konvertierungsvorschrift %6d bewirkt die sechsstellige Ausgabe der Zeilennummer, eventuell mit führenden Leerzeichen.

Variante 2 wandelt alle im Eingabefile enthaltenen nichtdruckbaren Zeichen vor der Ausgabe in eine spezielle Zeichenfolge \nnn um, wobei nnn eine Folge von drei Oktalziffern bezeichnet. Diese Darstellung entspricht den im Abschn. 1.2.2. behandelten Bitmustern.

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

main()              /* Umwandlung nichtdruckbarer Zeichen in \nnn */
{
    int c;

    while((c = getchar()) != EOF)
        if(isprint(c))
            putchar(c);
        else if(isspace(c))
            putchar(c);
        else
            printf("\\%.3o", c);
}
```

Bei isprint und isspace handelt es sich nicht um Funktionen, sondern um sog. **Makros**, die man aber wie Funktionen benutzen kann. Makros sind eine spezielle Spracherweiterung, die auf Fähigkeiten des C-Präprozessors basiert. Deshalb werden Makros ausführlich im Abschn. 3.2. behandelt. Im Moment genügt es, zu wissen, dass beide Makros Bestandteil der Standardbibliothek sind und durch die Steuerzeile

```
#include <ctype.h>
```

in den Quelltext eingeschlossen werden. Dabei liefert isprint(c) einen Wert ungleich Null, wenn c ein druckbares Zeichen ist. Hierzu zählen alle Buchstaben, Ziffern und die Sonderzeichen einschliesslich Leerzeichen. Darüber hinaus filtert isspace(c) ausser dem Leerzeichen auch die Zeichen Tabulator, Vertikal-Tabulator, Carriage return, Newline und Formfeed heraus. Alle anderen Steuerzeichen und Bitkombinationen werden mittels der Konvertierungsvorschrift %.3o vor der Ausgabe in eine "sichtbare" Zeichenfolge transformiert. Dabei erfolgt die Umwandlung des Zeichenwertes in eine dreistellige Oktalzahl, ggf. mit führenden Nullen.[1] Schliesslich sei daran erinnert, dass innerhalb einer Zeichenkettenkonstanten das Zeichen **Backslash** durch \\ dargestellt werden muss.

---

[1] Ältere printf-Versionen fordern hierfür die Angabe von %Ø3o.

### 1.5.4. Berechnen von Primzahlen

Als letzte Aufgabe dieses Abschnittes sollen alle Primzahlen eines Intervalls der Kardinalzahlen berechnet und ausgegeben werden.

Unter einer Primzahl versteht man eine natürliche Zahl i grösser als 1, die keinen echten Divisor j besitzt, wobei gilt: 1 < j < i. Das heisst, eine Primzahl ist nur durch 1 und durch sich selbst ohne Rest teilbar. Die Folge der ersten zehn Primzahlen lautet:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...

Um festzustellen, ob i eine Primzahl ist, versucht man eine Zahl j zu finden, durch die i ohne Rest teilbar ist, wobei j die Werte 2, 3, 4, 5, ... annimmt. Gelingt das für irgendeinen Wert j mit $j^2$ <= i, so ist i keine Primzahl.

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

#define YES  1
#define NO   0

main()              /* Primzahlen des Intervalls [von,bis] */
{
    long int n, von, bis;

    printf("Berechnung der Primzahlen eines Zahlenintervalls\n");
    printf("\nGeben Sie die untere Intervallgrenze ein:");
    von = getlong();
    printf("\nGeben Sie die obere Intervallgrenze ein:");
    bis = getlong();
    n = von;
    while(n <= bis)  {
        if(prim(n))
            printf("   %ld\n", n);
        n = n + 1;
    }
}

int prim(i)         /* liefert YES, falls i eine Primzahl ist */
    long int i;
{
    long int j;

    if(i < 2)
        return(NO);
    j = 2;
    while(j < i)  {
        if(i % j == 0)
            return(NO);
        else if(j * j > i)
            return(YES);
        else
            j = j + 1;
    }
    return(YES);
}
```

```
long int getlong()  /* Ziffernfolge lesen und konvertieren in long */
{
    long int i;
    int c;

    i = Ø;
    c = getchar();
    while(isdigit(c)) {
        i = 1Ø * i + c - 'Ø';
        c = getchar();
    }
    return(i);
}
```

Das Primzahlprogramm enthält zwei neue Aspekte - die Deklaration des Typs von Parametern einer Funktion und die Argumentübergabe beim Aufruf der Funktion.

Bei der Definition der Funktion prim wird i als formaler Parameter deklariert, der Werte vom Typ **long int** (oder kurz **long**) annehmen kann. Typdeklarationen von Funktionsparametern besitzen die gleiche syntaktische Form wie Variablenvereinbarungen. Sie müssen jedoch vor dem Beginn des Funktionsblocks stehen und können für Parameter vom Standardtyp **int** weggelassen werden. Beim Aufruf von prim durch die main-Funktion wird der aktuelle Wert der Variablen n übergeben und i zugewiesen, d.h., i wird mit dem Wert von n initialisiert.

Der Algorithmus der Primzahlberechnung ist ohne Zweifel nicht sehr effektiv. Das liegt daran, dass wir zum gegenwärtigen Zeitpunkt noch keine geeigneten sprachlichen Mittel zur Verfügung haben, um effektivere Verfahren anwenden zu können. Im Zusammenhang mit der Behandlung von Feldern werden wir deshalb auf diese Aufgabenstellung zurückkommen.

Alle die Primzahlberechnung betreffenden Variablen und der Funktionswert von getlong werden vom Typ **long** angenommen, um von vornherein für Primzahlen bis $2^{31}-1$ gewappnet zu sein. Daraus resultiert auch die Notwendigkeit der Konvertierungsvorschrift %ld bei printf.

Die Funktion getlong liest eine Folge von Dezimalziffern und konvertiert sie in einen **long**-Wert. Zur Prüfung jedes gelesenen Zeichens wird der Bibliotheksmakro isdigit benutzt.

Betrachten wir den Teilausdruck

```
c - 'Ø'
```

etwas genauer. Wie bereits früher erwähnt, stellen Zeichenkonstanten Integerwerte dar. Der Integerwert der jeweiligen Ziffer kann durch Subtraktion des Zeichens und der Zeichenkonstanten 'Ø' ermittelt werden. Das funktioniert nur dann, wenn Ziffern im Zeichensatz lückenlos und in aufsteigender Folge 'Ø', '1', '2' usw. dargestellt sind.

Bei genauer Analyse stellt man fest, dass getlong bei nichtkorrekten Eingaben keinerlei Fehlerbehandlung durchführt, sondern beim ersten Zeichen abbricht, das keine Dezimalziffer darstellt. Für den gegenwärtigen Zweck ist diese einfache Version jedoch ausreichend.

Es soll an dieser Stelle noch einmal verdeutlicht werden, dass die Wertzuweisung ein ganz normaler Ausdruck ist. Diese Eigenschaft kann oft genutzt werden, um kompakten Quellkode (und darüber hinaus effizienten Maschinenkode) zu erzielen.

```
long int getlong()  /* Ziffernfolge lesen und konvertieren in long */
{
    long int i;
    int c;

    i = Ø;
    while(isdigit(c = getchar()))
        i = 1Ø * i + c - 'Ø';
    return(i);
}
```

Vergegenwärtigen wir uns die Auswertung des Vergleichsausdrucks

```
isdigit(c = getchar())
```

Zunächst erfolgt der Aufruf der Funktion getchar, die das nächste Zeichen der Eingabe liefert. Dieser Wert wird der Variablen c zugewiesen und stellt zugleich den Argumentwert für isdigit dar. Wie oben erwähnt, prüft isdigit, ob es sich dabei um eine Dezimalziffer handelt. Wenn ja, so liefert isdigit einen Wert ungleich Null. Folglich besitzt der Vergleichsausdruck den Wahrheitswert TRUE genau dann, wenn das gelesene Zeichen eine Dezimalziffer ist.

**Aufgabe 1.1.** Modifizieren Sie das Programm zur Wurzelberechnung so, dass eine zur Zahlentafel analoge Tabelle entsteht, aus der sich alle Quadratwurzeln von 1.Ø bis 3Ø.Ø mit der Schrittweite Ø.1 ablesen lassen. Die eigentliche Wurzelberechnung soll in einer separaten Funktion erfolgen, während sich die main-Funktion auf den Tabellenaufbau und die Ausgabe konzentriert.

**Aufgabe 1.2.** Experimentieren Sie mit dem Programm zur Berechnung der Primzahlen, indem Sie unterschiedliche Intervallgrenzen eingeben. Messen Sie die Laufzeit des Programms, und vergleichen Sie diese später mit der Laufzeit des Primzahlprogramms in Abschn. 2.2.1.

**Aufgabe 1.3.** Schreiben Sie ein Programm, das eine Tabelle erzeugt, aus der die hexadezimalen und oktalen Äquivalente der Integerzahlen von Ø bis 1ØØ ablesbar sind. (Hinweis: Die Konvertierung könnte mit den printf-Konvertierungsvorschriften %x (hexadezimal) und %o (oktal) erfolgen.)

# 2. Sprachkonzepte von C

In den folgenden Abschnitten werden die charakteristischen Operatoren diskutiert und die Methoden der Datenstrukturierung sowie das Zeigerkonzept vorgestellt. Weiterhin gehen wir auf die noch nicht behandelten Anweisungen zur Steuerung des Programmablaufs ein und fassen die Möglichkeiten der Strukturierung von C-Progammen zusammen.

Bevor Sie sich mit diesen Problemen beschäftigen, sollten Sie einige praktische Erfahrungen mit dem bisher diskutierten Sprachumfang und dem zur Verfügung stehenden Compiler gesammelt haben.

## 2.1. Spezielle Operatoren

Einige der Operatoren von C sind in den meisten anderen höheren Programmiersprachen nicht vorhanden. Hierzu gehören:

- Inkrement- und Dekrementoperatoren, die spezielle Maschinenbefehle einiger Rechnerarchitekturen widerspiegeln
- Operatoren zur direkten Bitmanipulation
- zusammengesetzte Zuweisungsoperatoren zur kompakteren Kodierung und effektiveren Auswertung häufig auftretender Zuweisungsausdrücke
- ein Entscheidungsoperator für Zweiwegeentscheidungen.

### 2.1.1. Inkrement- und Dekrementoperatoren

Neben den bereits erläuterten arithmetischen Operatoren gibt es die unären Operatoren ++ zur Inkrementierung und -- zur Dekrementierung des Operanden um den Wert 1. Beide Operatoren können nur auf einen Operanden angewendet werden, der sich auf einen modifizierbaren Speicherbereich bezieht. Ein solcher Operand wird auch Lvalue-Ausdruck (oder kurz Lvalue) genannt. Das Ergebnis der Operationen ist kein Lvalue. Es ist erlaubt, Inkrement- und Dekrementoperatoren sowohl in der Präfix- als auch in der Postfixnotation zu verwenden:

```
++<lvalue>          /* Präfixnotation */
--<lvalue>

<lvalue>++          /* Postfixnotation */
<lvalue>--
```

Bei Verwendung der Präfixnotation ergibt sich der Ausdruckswert aus dem Wert des Operanden **nach** der Inkrementierung bzw. Dekrementierung. Im Gegensatz hierzu ist der Wert des Ausdrucks bei der Postfixnotation gleich dem Wert des Operanden **vor** Ausführung der Operation.

Folgende Beispiele verdeutlichen die unterschiedliche Wirkungsweise der Präfix- gegenüber der Postfixinkrementierung:

```
int x, y;
...
x = 3;
y = ++x;            /*  Präfixinkrementierung:  x = x + 1; y = x;  */
...
x = 3;
y = x++;            /*  Postfixinkrementierung:  y = x; x = x + 1;  */
```

In beiden Fällen wird zunächst der Ausdruckswert auf der rechten Seite berechnet und anschliessend der Variablen y zugewiesen. Demzufolge erhält y im ersten Fall den Wert 4, im Fall der Postfixinkrementierung jedoch den Wert 3. Ausserdem wird der Wert von x jeweils um 1 erhöht.

Inkrement- und Dekrementoperationen sind Operationen mit **Nebeneffekt**; der Haupteffekt besteht in der Ausdrucksberechnung, während die Inkrementierung bzw. Dekrementierung des Operanden den Nebeneffekt darstellen. [1] Oftmals jedoch ist der Nebeneffekt der eigentliche Zweck einer solchen Operation. Wird der Ausdruckswert nicht benötigt, ist es gleichgültig, ob man die Präfix- oder die Postfixnotation anwendet.

```
int x;
...
x = 3;
++x;                /*  ist identisch zu:  x++;  */
```

Als erstes Beispiel wollen wir eine Funktion zeigen, die für natürliche Zahlen n (n > 1) die Fakultät n! nach der Formel

```
n! = 1*2* ... *n
```

berechnet. Per Definition gilt

```
Ø! = 1
```

Der Quelltext der Funktion fakult zeigt, wie einfach diese Aufgabe gelöst werden kann.

```
long fakult(n)      /* Berechnung von n! = 1*2* ... *n  */
    int n;
{
    long i, fak;

    fak = i = 1;
    while(++i <= n)
        fak = fak * i;
    return(fak);
}
```

Betrachten wir die Auswertung des Vergleichsausdrucks

```
++i <= n
```

Der Teilausdruck ++i besitzt den gleichen Wert wie i nach der Inkrementierung. Dieser Wert geht in den Vergleich mit n ein. Ist die Bedingung erfüllt, d.h., ist der Wahrheitswert des Vergleichsausdrucks TRUE, so wird der Schleifenkörper ausgeführt.

Als zweites Beispiel folgt eine Version des Filekopierprogramms aus Abschn. 1.4.4., wobei der Dekrementoperator verwendet wird. Das Programm ersetzt jeden Tabulator durch eine Folge von Leerzeichen.

[1] Im Abschn. 2.1.7. wird auf die Probleme bei Ausdrücken mit Nebeneffekten eingegangen.

```
#include <stdio.h>
#define TABANZ 6

main()              /* Filekopieren mit Tabulatorauswertung */
{
    int c, i;

    while((c = getchar()) != EOF) {
        if(c != '\t')
            putchar(c);
        else {
            i = TABANZ;
            while(i--)
                putchar(' ');
        }
    }
}
```

Gegenüber dem vorhergehenden Beispiel ergibt sich der Wert des Ausdrucks

```
i--
```

aus dem Wert von i vor der Dekrementierung. Demzufolge wird der Schleifenkörper nur dann ausgeführt, wenn der "alte" Wert von i grösser als Ø ist.

Die Operatoren zur Inkrementierung und Dekrementierung werden hauptsächlich beim elementweisen Verarbeiten von Feldern sowie im Zusammenhang mit Zeigeroperationen angewendet.

**Aufgabe 2.1.** Suchen Sie in den bisherigen Beispielen nach Möglichkeiten, den Inkrementoperator anzuwenden.

**Aufgabe 2.2.** Gegeben seien folgende Anweisungen:

a)
```
int i, x;
...
x = 1;
i = Ø;
if(--x)
    x = i++;
else
    x = ++i;
```

b)
```
int i, x;
...
x = 1;
i = Ø;
if(x--)
    x = i++;
else
    x = ++i;
```

Welche Werte erhalten die Variablen x und i? Überprüfen Sie die Ergebnisse durch Programmtestung.

### 2.1.2. Bitorientierte Operatoren

Die bitorientierten Operatoren ermöglichen die direkte Manipulation einzelner Bits. Auf den meisten Rechnerarchitekturen können derartige Operationen durch den Compiler effektiv auf entsprechende Maschinenbefehle abgebildet werden. Neben den binären Operatoren

```
&                   /*  UND:  Maskieren von Bits  */
|, ^                /*  inklusives bzw. exklusives ODER  */
<<, >>              /*  Links- bzw. Rechtsverschiebung  */
```

gibt es den unären Operator

```
~                   /*  Einerkomplement  */
```

Die Operanden bitorientierter logischer Operatoren werden in jeder Bitposition nach folgenden Regeln miteinander verknüpft. Gegeben sei ein Ausdruck x <op> y. Bild 2.1 zeigt die Wirkung der Operatoren &, | und ^ bezogen auf jede einzelne Bitposition.

| x | y | x & y | x \| y | x ^ y |
|---|---|---|---|---|
| Ø | Ø | Ø | Ø | Ø |
| Ø | 1 | Ø | 1 | 1 |
| 1 | Ø | Ø | 1 | 1 |
| 1 | 1 | 1 | 1 | Ø |

Bild 2.1. Wirkung bitorientierter logischer Operatoren

Bei Anwendung des Operators & ist z.B. jedes Bit des Ausdruckswertes genau dann 1, wenn die korrespondierenden Bits beider Operanden 1 sind usw.

Bei Verschiebeoperationen der Form x << n bzw. x >> n wird x um so viele Stellen nach links bzw. nach rechts verschoben, wie n angibt.

In einem Ausdruck ~x bewirkt der Einerkomplementoperator ~ das Vertauschen der Bitbelegung seines Operanden x.

Die Operanden bitorientierter Operationen unterliegen gewissen Forderungen:

- Sie müssen vom integralen Typ sein, z.B. **char**, **short**, **int**, **unsigned**, **long** oder eine der möglichen Kombinationen.
- Bei Verschiebeoperationen muss der Verschiebewert positiv und, gemessen in Bits, kleiner als die Länge des zu verschiebenden Operanden sein. Andernfalls ist das Ergebnis undefiniert.
- Bei der Linksverschiebung erfolgt ein Auffüllen der frei werdenden Bitpositionen mit Ø, d.h., es handelt sich um eine **logische** Verschiebung. Bei der Rechtsverschiebung kann bei einem negativen Operandenwert eine arithmetische Verschiebung erfolgen, d.h., frei werdende Bitstellen werden mit 1 aufgefüllt. Für Werte vom Typ **unsigned** wird die logische Verschiebung garantiert.

Verdeutlichen wir die Anwendung einiger der bitorientierten Operatoren. Zu diesem Zweck nehmen wir an, dass die Bitpositionen einer **int**-Variablen wie folgt durchnumeriert werden: Das Bit mit dem Stellenwert $2^{Ø}$ belegt die Position Ø, das Bit mit dem Stellenwert $2^1$ belegt die Position 1 usw. Die Bitpositionen Ø, 1 und 2 einer Variablen flag sollen als Anzeigebits benutzt werden.

```
#define  ANZEIGE   Ø7
#define  BITPOS_Ø  Ø1
#define  BITPOS_1  Ø2
#define  BITPOS_2  Ø4
  ...
int flag;
```

Die erste Anweisung zeigt, wie man alle drei Anzeigebits setzen kann.

```
flag = flag | ANZEIGE;      /* Setzen der drei Anzeigebits auf 1 */
```

Nun werden alle Anzeigebits auf Ø gesetzt.

```
flag = flag &~ANZEIGE;   /* Löschen aller Anzeigebits */
```

~ANZEIGE ist ein Konstantenausdruck, der einem Oktalwert Ø17777Ø entspricht. Das gilt für Rechnerarchitekturen mit einer Wortlänge von 16 Bit. Bei Maschinen mit 32-Bit-Integerwerten entspricht ~ANZEIGE einem Oktalwert von Ø37777777770. Durch die UND-Verknüpfung werden alle Bitpositionen bis auf die Positionen Ø, 1 und 2 maskiert. Der Ausdruck

```
flag = flag &~ANZEIGE
```

ist einem Ausdruck

```
flag = flag & Ø17777Ø
```

vorzuziehen, da ~ANZEIGE unabhängig von der konkreten Länge des Datentyps **int** ist. Ausserdem erfolgt die Berechnung von Konstantenausdrücken bereits während der Compilierung. (Konstantenausdrücke werden im Abschn. 2.1.8. behandelt.)

In der nächsten Anweisung benutzen wir den Operator |, um die Bits der Positionen Ø und 2 der Variablen flag zu setzen.

```
flag = flag | BITPOS_Ø | BITPOS_2;  /* Positionen Ø und 2 setzen */
```

Der Vergleichsausdruck der folgenden **if**-Anweisung testet, ob das in Position 1 stehende Bit von flag auf 1 gesetzt ist.

```
if(flag & BITPOS_1) {      /* Abfrage, ob Bit 1 belegt */
   ...
}
```

Schliesslich prüfen wir, ob **nur** das Anzeigebit in Position Ø gesetzt ist.

```
if((flag & ANZEIGE) == BITPOS_Ø) {  /* Nur letztes Bit gesetzt? */
   ...
}
```

Dabei sichert der Teilausdruck

```
flag & ANZEIGE
```

dass nur die Bitpositionen Ø, 1 und 2 in den Vergleich eingehen.

Um die Anwendung bitorientierter Operatoren zu demonstrieren, stellen wir die Funktion wordlen vor. Sie berechnet die Länge des Datentyps **int** derjenigen Maschine, auf der die Funktion übersetzt und ausgeführt wird.

```
int wordlen()      /* Anzahl der Bits, die ein int-Wert belegt */
{
   int i;
   unsigned int wort;

   i = wort = Ø;
   wort = ~wort;
   while(wort != Ø) {
      wort = wort >> 1;
      i++;
   }
   return(i);
}
```

Da wort ursprünglich mit dem Wert Ø initialisiert wurde, setzt

```
wort = ~wort;
```

alle Bitpositionen von wort auf 1. Die Anweisung

```
wort = wort >> 1;
```

verschiebt diese **unsigned**-Variable um jeweils eine Bitstelle nach rechts und füllt von links mit Ø auf. Das geschieht so oft, bis die Variable den Wert Ø besitzt. Nach jeder Verschiebeoperation wird der Zähler i um 1 erhöht, er stellt letztlich den Funktionswert dar. Die Lösung ist portabel, d.h., der Quellkode der Funktion kann ohne Änderung auf einen beliebigen Maschinentyp übertragen werden.

**Aufgabe 2.3.** Programmieren und testen Sie ein Programm, das in einer Tabelle die Anzahl der Bits ausgibt, die Werte der Datentypen **char**, **short**, **int** und **long** auf "Ihrem" Rechner belegen. (Warum ist es günstiger, statt der Rechtsverschiebung die Linksverschiebung zu verwenden?)

**Aufgabe 2.4.** Es soll ein Programm geschrieben werden, dass zyklisch jeweils ein Zeichen liest und das zugehörige Bitmuster ausgibt.

### 2.1.3. Logische Operatoren

Es gibt die binären logischen Operatoren

```
&&          /*  bedingtes logisches UND   */
||          /*  bedingtes logisches ODER  */
```

sowie den unären Operator

```
!           /*  logische Negation  */
```

Die logische Negation liefert den Wert 1, wenn der Operand den numerischen Wert Ø besitzt, ansonsten den Wert Ø. Die Wirkung dieses Operators kann ausgenutzt werden, um den Wahrheitswert von Ausdrücken festzustellen. Wie bereits erwähnt, ist der Wahrheitswert eines Ausdrucks TRUE bzw. FALSE, wenn sein numerischer Wert ungleich Ø bzw. gleich Ø ist.

```
int schalter;
...
if(!schalter)
    <anweisung1>
else
    <anweisung2>
```

Besitzt die Variable schalter den Wert Ø, wird <anweisung$_1$> ausgeführt. In allen anderen Fällen erfolgt die Ausführung von <anweisung$_2$>.

Enthalten Ausdrücke die **bedingten** logischen Operatoren && bzw. ||, so erfolgt die Ausdrucksauswertung von links nach rechts so lange, bis der Wahrheitswert des Gesamtausdrucks eindeutig festgestellt werden kann. Demzufolge besitzt ein logischer UND-Ausdruck den Wert 1 (und damit den Wahrheitswert TRUE) genau dann, wenn beide Operandenwerte verschieden von Ø sind. Ist der Wert eines Operanden gleich Ø, so hat auch der UND-Ausdruck

diesen Wert. Der rechte Operand wird nur dann ausgewertet, wenn der linke Operand verschieden von Ø ist. Für einen logischen ODER-Ausdruck gilt analog, dass der rechte Operand nur ausgewertet wird, wenn der Wert des linken Operanden gleich Ø ist. Der Gesamtausdruck ist TRUE, wenn wenigstens einer der Operanden einen Wert ungleich Ø besitzt. Logische Operatoren kann man auf Operanden mit arithmetischem Typ sowie auf Zeigerwerte anwenden. Logische Ausdrücke liefern immer Werte vom Typ **int**.

Unser erstes Beispiel ist eine Variante des Filekopierprogramms aus Abschn. 1.5.3., wobei die nichtdruckbaren Zeichen eines Files in eine Zeichenfolge der Form \nnn transformiert werden.

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

main()              /* Transformation nichtdruckbarer Zeichen */
{
    int c;

    while((c = getchar()) != EOF)
        if(isprint(c) || isspace(c))
            putchar(c);
        else
            printf("\\%.3o", c);
}
```

Betrachten wir den Vergleichsausdruck der **if**-Anweisung etwas genauer. Liefert isprint einen Wert ungleich Ø, so steht der Wahrheitswert des Gesamtausdrucks bereits mit TRUE fest, und isspace wird nicht ausgeführt.

Als weiteres Beispiel zeigen wir die Funktion letter. Sie prüft, ob ihr Argument ein Buchstabe ist, und gibt in diesem Fall den Funktionswert 1 zurück, andernfalls Ø.

```
int letter(c)       /* liefert 1, falls c Buchstabe (nur ASCII) */
    int c;
{
    if( ((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z')) )
        return(1);
    else
        return(Ø);
}
```

Die Auswertung des Vergleichsausdrucks

```
((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))
```

erfolgt von links nach rechts. Wird der Funktion z.B. als Argumentwert ein Kleinbuchstabe übergeben, so besitzt der Teilausdruck

```
((c >= 'a') && (c <= 'z'))
```

den Wahrheitswert TRUE. Somit ist auch der Wahrheitswert des gesamten ODER-Ausdrucks mit TRUE festgestellt, und die Auswertung wird abgebrochen. Vergleichsoperatoren besitzen einen höheren Vorrang als logische Operatoren, d.h., vor der UND- bzw. ODER-Verknüpfung erfolgt jeweils die Auswertung der Vergleiche. Da der Vorrang von || ausserdem geringer als der von && ist, kann man sämtliche Klammern weglassen:

```
c >= 'a' && c <= 'z'  ||  c >= 'A' && c <= 'Z'
```

Die Funktion letter in der obigen Form arbeitet nur korrekt bei einem Zeichensatz, in dem die Gross- bzw. Kleinbuchstaben in geschlossener Folge angeordnet sind. Das ist z.B. im ASCII gegeben. Folgende Version funktioniert auch bei Verwendung des EBCDIC-Zeichensatzes.

```
int letter(c)       /* Prüfung, ob c Buchstabe (portable Version) */
    int c;
{
    if (  c >= 'a'  &&  c <= 'i'
       || c >= 'j'  &&  c <= 'r'
       || c >= 's'  &&  c <= 'z'
       || c >= 'A'  &&  c <= 'I'
       || c >= 'J'  &&  c <= 'R'
       || c >= 'S'  &&  c <= 'Z' )
        return(1);
    else
        return(Ø);
}
```

Man beachte den Unterschied, der zu den bitorientierten Operatoren & bzw. | besteht.

```
int x, y, z;
 ...
x = 1;
y = 2;
z = x & y;          /*  z erhält den Wert Ø  */
z = x && y;         /*  z erhält den Wert 1  */
z = x | y;          /*  z erhält den Wert 3  */
z = x || y;         /*  z erhält den Wert 1  */
```

**Aufgabe 2.5.** Erklären Sie die unterschiedlichen Ergebnisse dieser Anweisungen.

**Aufgabe 2.6.** Schreiben Sie die Funktion isdigit(c), die als Funktionswert 1 liefert, wenn das Zeichen c eine Dezimalziffer ist. In allen anderen Fällen soll Ø zurückgegeben werden.

**Aufgabe 2.7.** In analoger Weise soll die Funktion isxdigit(c) programmiert und getestet werden, die 1 zurückgibt, wenn c eine Hexadezimalziffer ist. (Sie können die Funktion isdigit zu Hilfe nehmen.)

### 2.1.4. Zusammengesetzte Zuweisungsoperatoren

Häufig auftretende Wertzuweisungen der allgemeinen Form

```
<lvalue> = <lvalue> <op> (<ausdruck>)
```

können verkürzt geschrieben werden:

```
<lvalue> <op>= <ausdruck>
```

Dabei kann <op> einer der binären arithmetischen (+,-,*,/,%) bzw. binären bitorientierten Operatoren (&,|,^,<<,>>) sein.

```
i += j;             /*  erhöht i um den Wert j  */
wort >>= 1;         /*  verschiebt wort um eine Stelle nach rechts  */
x1 *= x2 - 1;       /*  anstelle x1 = x1 * (x2 - 1);  */
```

Der Compiler muss den Ausdruck links vom Zuweisungsoperator nur einmal bewerten, es sind dadurch Voraussetzungen gegeben, effizienten Zielkode zu generieren. Ansonsten gelten die gleichen Aussagen wie bei normalen Wertzuweisungen, insbesondere muss der linke Operand ein Lvalue sein usw. Als Beispiel geben wir die Funktion ndigit an, welche die Anzahl der Ziffern einer Zahl vom Typ **int** ermittelt.

```
int ndigit(n)       /* Ziffernanzahl eines int-Wertes */
    int n;
{
    int ndigits, m;

    m = n;
    ndigits = 1;
    while(m /= 1Ø)
        ndigits++;
    return(ndigits);
}
```

Der Vergleichsausdruck

```
m  /= 1Ø
```

ist dabei eine Kurzform von:

```
(m = m/1Ø) != Ø
```

**Aufgabe 2.8.** Die Eulersche Zahl e kann über die Reihe

```
1 + 1 + 1/2! + 1/3! + ... + 1/n! + ...
```

mit beliebiger Genauigkeit angenähert werden. Entwickeln Sie ein Programm, dass für geeignete Eingabewerte n und unter Anwendung der bereits vorgestellten Funktion fakult (vgl. Abschn. 2.1.1.) einen Näherungswert für e berechnet.

### 2.1.5. Entscheidungsoperator

Der Entscheidungsoperator ?: verknüpft drei Ausdrücke:

<ausdruck$_1$> ? <ausdruck$_2$> : <ausdruck$_3$>

Der Wert des Gesamtausdrucks ist vom Wahrheitswert des Vergleichsausdrucks <ausdruck$_1$> abhängig und entspricht dem Wert von <ausdruck$_2$> im Fall TRUE bzw. dem Wert von <ausdruck$_3$> im Fall FALSE. Es wird also nur einer der Ausdrücke <ausdruck$_2$> bzw. <ausdruck$_3$> ausgewertet. Man kann den Entscheidungsoperator benutzen, um Zweiwegeentscheidungen der allgemeinen Form

```
if(<ausdruck1>)
    <lvalue> = <ausdruck2>;
else
    <lvalue> = <ausdruck3>;
```

verkürzt zu kodieren:

```
<lvalue> = <ausdruck1> ? <ausdruck2> : <ausdruck3>;
```

Um z.B. der Variablen max den grösseren der Werte x und y zuzuweisen, ist es möglich, anstelle von

```
if(x > y)
    max = x;
else
    max = y;
```

einfach

```
max = x > y ? x : y;
```

zu schreiben.

Es wird darauf hingewiesen, dass eine derartige Operation ein Ausdruck ist und demzufolge wie jeder andere Ausdruck verwendet werden kann. Als Beispiel folgt die Funktion abs zur Berechnung des absoluten Betrags eines Integerwertes.

```
int abs(x)          /* Absolutbetrag von x */
{
    return(x < 0 ? -x : x);
}
```

Der Wert des **return**-Ausdrucks ist -x (bzw. x), wenn der Argumentwert x negativ (bzw. nichtnegativ) ist.

Noch deutlicher kann man die typische Anwendung des Entscheidungsoperators am Beispiel des Primzahlprogramms (vgl. Abschn. 1.5.4.) zeigen. Bei dieser Variante werden immer 8 Primzahlen auf einer Zeile ausgegeben, bevor die Umschaltung auf die nächste Zeile erfolgt.

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

#define YES  1
#define NO   0

main()              /* Primzahlen des Intervalls [von,bis] */
{
    long int n, i, von, bis;

    printf("Berechnung der Primzahlen eines Zahlenintervalls\n");
    printf("\nGeben Sie die untere Intervallgrenze ein:");
    von = getlong();
    printf("\nGeben Sie die obere Intervallgrenze ein:");
    bis = getlong();
    n = von;
    i = 1;
    while(n <= bis)  {
        if(prim(n))
            printf(" %6d%c", n, i++ % 8  ?  ' '      '\n');
        n++;
    }
}
```

Gegenüber der ursprünglichen Version sind nur minimale Änderungen erforderlich, sie sind in der Anweisung

```
printf(" %6d%c", n, i++ % 8  ?  ' '      '\n');
```

konzentriert. Die Konvertierungsvorschrift %6d bedeutet, dass die Primzahl rechtsbündig als sechsstellige Dezimalzahl ausgegeben werden soll. Mit %c

wird die Ausgabe eines Zeichens angefordert. Der Wert des Ausdrucks

```
i++ % 8 ? ' '     '\n'
```

bestimmt, um welches Zeichen es sich dabei handelt. Da der aktuelle Wert von i angibt, wie viele Primzahlen bereits ausgegeben wurden, kann auf oben gezeigte Weise gesteuert werden, dass nach je 8 Zahlenwerten die Zeilenschaltung erfolgt.

**Aufgabe** 2.9. Überarbeiten Sie die Funktion isdigit(c) aus Aufgabe 2.6, indem Sie den Entscheidungsoperator anwenden. (Die Funktion liefert den Wert 1, wenn c eine Dezimalziffer ist, sonst Ø.)

### 2.1.6. Kommaoperator

Zwei durch Komma getrennte Ausdrücke

<ausdruck$_1$> , <ausdruck$_2$>

werden von links nach rechts ausgewertet. Typ und Wert des rechten Ausdrucks bestimmen Typ und Wert des Gesamtausdrucks. Zum Beispiel wird bei einem Ausdruck

```
x = 1, y += x
```

zunächst x der Wert 1 zugewiesen und anschliessend y um diesen Wert von x erhöht. Der Ausdruckswert ergibt sich aus dem neuen Wert von y.

Der Operator , darf nicht mit dem Trennzeichen , verwechselt werden, wie es in Parameterlisten bei Funktionsdefinitionen, Argumentlisten bei Funktionsaufrufen und Bezeichnerlisten bei Vereinbarungen vorkommt. Häufig wird der Kommaoperator im Zusammenhang mit **for**-Schleifen angewendet.

### 2.1.7. Vorrangregeln und Assoziativität

Die Auswertungreihenfolge von Ausdrücken wird durch den **Vorrang** der Operatoren bestimmt. Die **Assoziativität** gibt an, ob eine Folge von Operatoren gleichen Vorrangs von links oder von rechts abgearbeitet wird. Zum Beispiel ist der Vorrang des Vergleichsoperators == höher als der des bitorientierten Operators &, woraus die Notwendigkeit folgt, den &-Ausdruck in einer Konstruktion

```
if((flag & ANZEIGE) == BITPOS_Ø)
```

zu klammern. Zuweisungsoperatoren = und <op>= sind rechtsassoziativ, d.h., ein Ausdruck

```
x = y = z = Ø
```

wird behandelt, als wäre er folgendermassen geklammert:

```
x = (y = (z = Ø))
```

Vorrang und Assoziativität aller Operatoren sind im Bild 2.2 zusammengefasst, dabei steht L für Links- und R für Rechtsassoziativität. Die Operatoren in einer Zeile besitzen den gleichen Vorrang, der insgesamt von oben nach unten abnimmt.

| Operatoren | Assoziativität |
|---|---|
| () [] -> . | L |
| ! ~ ++ -- - (<typ_spezifikation>) * & sizeof | R |
| * / % | L |
| + - | L |
| << >> | L |
| < <= > >= | L |
| == != | L |
| & | L |
| ^ | L |
| \| | L |
| && | L |
| \|\| | L |
| ?: | R |
| = <op>= | R |
| , | L |

Bild 2.2. Vorrang und Assoziativität der Operatoren

Nicht alle der hier aufgeführten Operatoren wurden bislang behandelt. Der Operator () zum Aufruf einer Funktion wird im Abschn. 2.5.2. erläutert, der Operator [] zur Feldindizierung im Abschn. 2.2.1., der Operator . zur Auswahl einer Strukturkomponente im Abschn. 2.2.2., die explizite Typkonvertierung im Abschn. 2.8.4., der Längenoperator **sizeof** im Abschn. 2.2.7., die unären Zeigeroperatoren (*,&) im Abschn. 2.3.2. sowie der Operator -> zur Auswahl einer Strukturkomponente mittels Zeiger im Abschn. 2.3.5. Auf einige Besonderheiten im Zusammenhang mit der Reihenfolge der Ausdrucksauswertung soll eingegangen werden.

Sieht man vom Vorrang der Operatoren ab, so ist es von der jeweiligen Compilerimplementation abhängig, in welcher Folge die Auswertung der Operanden eines Operators erfolgt. Das kann insbesondere dann zu Problemen führen, wenn diese Ausdrücke Nebeneffekte enthalten, z.B. durch Inkrement- und Dekrementoperatoren verursacht. Ein Ausdruck besitzt einen undefinierten Wert, wenn sich der Wert einer Variablen in dem Ausdruck durch einen Nebeneffekt verändert und dieselbe Variable an einer anderen Stelle im gleichen Ausdruck benutzt wird. Ein typisches Beispiel hierfür ist

```
x = f(y) + g(y++);
```

Es bleibt der jeweiligen Compilerimplementation überlassen, ob zuerst die Funktion f oder die Funktion g ausgeführt wird. Demzufolge ist nicht klar, ob der Aufruf von f mit dem inkrementierten Argumentwert y oder dem ursprünglichen Wert von y erfolgt.

Allgemein gilt, dass der Compiler Ausdrücke, die assoziative und kommutative Operatoren (+, *, |, ^, &) enthalten, auch dann umordnen kann,

wenn einzelne Teilausdrücke explizit geklammert sind. Ein Ausdruck

```
x * (y * z)
```

könnte z.B. in die Form

```
(x * y) * z
```

transformiert werden. Es ist jedenfalls **nicht** möglich, durch Klammerung derartiger Ausdrücke eine bestimmte Auswertungsreihenfolge zu erzwingen. (Um eine bestimmte Reihenfolge der Ausdrucksauswertung zu erreichen, kann man temporäre Variablen verwenden, in denen die Werte von Teilausdrücken zwischengespeichert werden.)

Auf den ersten Blick vielleicht noch überraschender erscheint die Tatsache, dass die Reihenfolge der Argumentauswertung bei Funktionsaufrufen durch die Sprachdefinition nicht festgelegt ist. Abhängig vom Maschinentyp und der Implementation erfolgt die Auswertung entweder von rechts oder von links. Demzufolge ist z.B. der Ausdruck

```
f(x++, g(x))
```

nicht portabel.[1)]

Derartige "Lücken" in der Definition von C sollen den verschiedenen Compilerimplementationen genügend Spielraum lassen, die speziellen Gegebenheiten der jeweiligen Rechnerarchitektur zu einer möglichst effektiven Ausdrucksauswertung und Kodegenerierung auszunutzen. Auf jeden Fall muss sich der Programmierer der damit verbundenen Gefahren bewusst sein. Er sollte es grundsätzlich unterlassen, Ausdrücke zu kodieren, die von der Auswertungsreihenfolge abhängen.

Betrachten wir abschliessend die Funktion getint. Sie liest eine Folge von Dezimalziffern und konvertiert sie in einen Integerwert. Vor der ersten Ziffer kann das Zeichen - stehen, um einen negativen Wert anzukündigen.

```
int getint()        /* Lesen [-]zz...z und konvertieren in int-Wert */
{
    int i, c, s;

    s = 1;
    i = Ø;
    while((c = getchar()) != '-' && (c < 'Ø' || c > '9'))
        ;
    if(c == '-') {
        s = -1;
        c = getchar();
    }
    while(c >= 'Ø' && c <= '9') {
        i = i * 1Ø + c - 'Ø';
        c = getchar();
    }
    return(s * i);
}
```

Der Vergleichsausdruck

```
(c = getchar()) != '-' && (c < 'Ø' || c > '9')
```

---

1) Das Prüfprogramm lint erkennt viele dieser potentiellen Fehlerquellen.

realisiert das Überlesen aller Zeichen, die vor dem - bzw. der ersten Ziffer eingegeben werden. Man beachte dabei die Klammernsetzung. Der Körper der **while**-Schleife besteht nur aus einer leeren Anweisung, die erforderlich ist, um der **while**-Syntax zu genügen. (In der hier vorgestellten Form ist die Funktion getint sicher nur für die Eingabe vom Terminal geeignet, da sie auf eine Fileendebedingung (EOF) nicht reagiert, sondern hartnäckig das nächste Zeichen anfordert. Ausserdem liefert sie den Wert Ø auch dann, wenn nach dem Vorzeichen - keine Dezimalziffer folgt.)

### 2.1.8. Konstantenausdrücke

Im Zusammenhang mit bestimmten Sprachelementen, wie z.B. der **switch**-Anweisung und der Initialisierung, spielen **Konstantenausdrücke** eine Rolle. Darunter werden Ausdrücke verstanden, deren Operanden Konstanten mit integralem Typ sind, die mit folgenden Operatoren verknüpft sein können:

```
(<integraler_typ>)      /* Typkonvertierung  (s. Abschn. 2.8.4.) */
sizeof                  /* Längenoperator  (s. Abschn. 2.2.7.) */
-                       /* negatives Vorzeichen */
~                       /* Einerkomplement */
+, -, *, /, %           /* binäre arithmetische Operatoren  */
&, |, ^, <<, >>         /* binäre bitorientierte Operatoren */
==, !=, <, >, <=, >=    /* Vergleichsoperatoren */
&&, ||                  /* bedingte logische Operatoren */
?:                      /* Entscheidungsoperator */
```

Die Berechnung des Wertes von Konstantenausdrücken erfolgt grundsätzlich während der Compilierung.

## 2.2. Strukturierte Datentypen

Strukturierte Typen entstehen durch die Anwendung bestimmter Strukturierungsmethoden auf Grundtypen bzw. strukturierte Datentypen. Unter einem **Feld** (array) versteht man eine Zusammenfassung von Elementen des gleichen Typs, während eine **Struktur** (structure) aus einer Menge von Komponenten besteht, die unterschiedlichen Typ besitzen können. Durch die Überlagerung mehrerer Komponenten verschiedenen Typs entsteht eine **Union** (union).

Die sog. **Bitfelder** (bitfields) stehen in keiner Beziehung zu Feldern, sie stellen eine spezielle Form von Strukturkomponenten dar. Der **Aufzählungstyp** (enum) gehört eigentlich zu den Grunddatentypen. Trotzdem wird er in diesem Abschnitt erläutert, da er erst nachträglich in das Typkonzept von C aufgenommen wurde und die Syntax der **enum**-Vereinbarung an die Syntax der Strukturvereinbarung angelehnt ist.

**Typdefinitionen** mittels **typedef** erlauben es, Synonyme für beliebige Datentypen einzuführen.

### 2.2.1. Felder

Ein **Feld** stellt eine Zusammenfassung von Datenobjekten gleichen Typs dar. Auf die Elemente des Feldes kann man mittels Indizierung zugreifen. Beispielsweise wird durch

```
float array[1Ø];
```

ein Feld array mit 10 Elementen vom Typ **float** vereinbart. Zu beachten ist, dass der Index des ersten Elements Ø ist und dieses Feld aus den Elementen array[Ø] bis array[9] besteht. Die allgemeine Form einer Feldvereinbarung lautet:

```
<speicherklasse> <typ> <bezeichner>[<konst_ausdruck>]
```

Dabei muss <konst_ausdruck> einen Integerwert ergeben. Es ist üblich, dafür eine symbolische Konstante anzugeben, so dass beispielsweise

```
#define N 1ØØ
...
int z[N], i;
```

ein Feld mit 100 **int**-Elementen (z[Ø],...,z[99]) sowie eine **int**-Variable i vereinbart.

Um allen Elementen von z den Wert Ø zuzuweisen, kann eine **while**-Anweisung benutzt werden:

```
...
i = Ø;
while(i < N)
    z[i++] = Ø;
```

Die Variable i wird als Index benutzt. Erlaubt sind beliebige Ausdrücke, die Integerwerte liefern.

Wir wollen die Arbeit mit Feldern an vollständigen Programmen zeigen und wenden uns zunächst der Berechnung des Skalarprodukts der Vektoren A und B mit jeweils n Koordinaten zu. Die Formel lautet:

$$A \odot B = a_1 * b_1 + a_2 * b_2 + \ldots + a_n * b_n$$

Die Eingabe der Anzahl bzw. Werte der Vektorkoordinaten soll vom Programm angefordert werden.

```
#include <stdio.h>
#define N 2Ø

main()              /* Skalarprodukt A⊙B der Vektoren a und b */
{
    int n, i, sp, a[N], b[N];

    sp = Ø;
    printf("Skalarprodukt von A und B\n\n");
    printf("Geben Sie die Anzahl der Koordinaten ( < %d ) ein:", N);
    if((n = getint()) > 1  &&  n < N) {
        printf("\nund jetzt %d Koordinaten fuer Vektor 'A':\n", n);
        getvekt(a, n);
        printf("\nsowie %d Koordinaten fuer Vektor 'B':\n", n);
        getvekt(b, n);
        i = Ø;
        while(i < n) {
            sp += a[i] * b[i];
            i++;
        }
        printf("\nDas Skalarprodukt von A und B lautet: %d\n", sp);
    } else
        printf("unzulaessiger Wert fuer Anzahl der Koordinaten\n");
}
```

Die Funktion getvekt liest alle Koordinaten eines Vektors und ruft zu diesem Zweck die im Abschn. 2.1.7. vorgestellte Funktion getint für die Eingabe und Konvertierung einer Integerzahl auf.

```
void getvekt(a, n)      /* Lesen der n Koordinaten des Vektors a */
    int a[];
    int n;
{
    int i;

    i = 0;
    while(i < n)
        a[i++] = getint();
}
```

Da getvekt keinen Funktionswert liefert, handelt es sich hierbei genaugenommen um keine Funktion, sondern um eine Prozedur oder auch Subroutine. Diese Begriffe sind im C-Sprachgebrauch jedoch nicht üblich. Durch das Typattribut **void** wird explizit angezeigt, dass getvekt keinen Funktionswert zurückgibt.

Die Deklaration

```
int a[];
```

vereinbart den formalen Parameter a der Funktion getvekt als Feld mit **int**-Elementen. Beim Aufruf der Funktion getvekt wird als Argument die Adresse des ersten Elements eines Integerfeldes übergeben. (Darauf gehen wir im Abschn. 2.5.2. näher ein.)

Als zweites Beispiel greifen wir das Problem der Berechnung von Primzahlen auf.

```
#include <stdio.h>

#define YES 1
#define NO 0
#define N 20000

main()              /* Primzahlberechnung: Sieb des Eratosthenes */
{
    short prim_anz[N+1];
    int i, j, k, n;

    printf("Berechnung der Primzahlen bis (maximal %d):", N);
    if((n = getint()) >= 1  &&  n <= N) {
        putchar('\n');
        i = 2;
        while(i <= n)
            prim_anz[i++] = YES;
        i = 2;
        k = 1;
        while(i <= n)  {
            if(prim_anz[i])  {
                j = i;
                printf("  %6d%c", i, k++ % 8  ?  ' '      '\n');
                while(i*i < n && (j += i) <= n)
                    prim_anz[j] = NO;
            }
            i++;
        }
    } else
        printf("fehlerhafter Eingabewert\n");
}
```

Die Elemente des Feldes prim_anz zeigen an, ob der Indexwert eine Primzahl ist oder nicht. Das Prinzip des Algorithmus besteht darin, alle Feldelemente zu markieren (in unserem Beispiel mit NO), deren Indexwerte das Vielfache einer gefundenen Primzahl darstellen. Das geschieht so lange, bis eine Primzahl i gefunden wird, für die $i^2$ >= n gilt, wobei n die obere Grenze des betrachteten Intervalls angibt. Dieses Verfahren ist als "Sieb des Eratosthenes" überliefert.

Eine besondere Rolle spielen Felder, deren Elemente Zeichen sind, sog. Zeichenfelder oder auch Zeichenketten. Wie bei Zeichenkettenkonstanten sollten Zeichenfelder mit dem Zeichen \Ø abgeschlossen werden, um die Endeerkennung zu erleichtern. Es gibt keine Operatoren zur Verarbeitung kompletter Zeichenfelder, statt dessen stehen eine Reihe von Standardfunktionen zur Verfügung, z.B.

```
strlen(s)           /* Länge des Zeichenfeldes s */
strcpy(s1, s2)      /* Kopieren von s2 nach s1 */
strcmp(s1, s2)      /* Vergleich der Zeichenfelder s1 und s2 */
strncmp(s1, s2, n)  /* Vergleich von s1 und s2 (maximal n Zeichen) */
strcat(s1, s2)      /* Kopieren von s2 an das Ende von s1 */
strchr(s, c)        /* Suche des Zeichens c im Zeichenfeld s */
```

Die Einfachheit derartiger Funktionen soll an zwei Beispielen verdeutlicht werden. (Die hier vorgestellten Versionen oben aufgeführter Funktionen sind nicht identisch zu denen in der Standardbibliothek, aber das ist in diesem Zusammenhang ohne Belang.) Die erste Funktion kopiert die Elemente eines Zeichenfeldes in ein anderes Feld in der Annahme, dass das Zielfeld gross genug ist, um alle Zeichen aufzunehmen.

```
void strcpy(s1, s2)  /* Zeichenfeld s2 nach s1 kopieren */
    char s1[], s2[];
{
    int i;

    i = Ø;
    while((s1[i] = s2[i]) != '\Ø')
        i++;
}
```

Die Elemente des Feldes s2 werden einschliesslich des Zeichens \Ø kopiert. Da \Ø den Wert Ø besitzt, ist folgende Kurzschreibweise möglich:

```
void strcpy(s1, s2)  /* Zeichenfeld s2 nach s1 kopieren */
    char s1[], s2[];
{
    int i;

    i = Ø;
    while(s1[i] = s2[i])
        i++;
}
```

Die Funktion strncmp vergleicht maximal n Zeichen der Zeichenfelder s1 und s2 und liefert einen Funktionswert kleiner als Ø, grösser als Ø bzw. gleich Ø, falls s1 lexikographisch kleiner als s2, grösser als s2 bzw. gleich s2 ist.

```
int strncmp(s1, s2, n) /* Vergleich der Zeichenfelder s1 und s2 */
    char s1[], s2[];
{
    int i;

    i = Ø;
    while(i < n && s1[i] == s2[i])
        if(s1[i++] == '\Ø')
            return(Ø);
    return(i == n ? Ø : s1[i] - s2[i]);
}
```

Der Funktionswert ergibt sich aus der Differenz der numerischen Werte der Feldelemente s1[i] und s2[i], wobei i die Position im jeweiligen Feld angibt, in der die beiden Felder nicht mehr übereinstimmen.

C unterstützt mehrdimensionale Felder. Durch

```
int a[2][3];
```

erfolgt die Vereinbarung eines zweidimensionalen Feldes a mit den Elementen a[Ø][Ø], a[Ø][1], a[Ø][2], a[1][Ø], a[1][1] und a[1][2]. Die Elemente werden zeilenweise gespeichert, d.h., der am weitesten rechts stehende Index variiert am schnellsten. Demzufolge werden die 24 Elemente eines dreidimensionalen Feldes x[2][3][4] in der Reihenfolge

```
x[Ø][Ø][Ø], x[Ø][Ø][1], x[Ø][Ø][2], x[Ø][Ø][3],
x[Ø][1][Ø], x[Ø][1][1], x[Ø][1][2], x[Ø][1][3],
x[Ø][2][Ø], x[Ø][2][1], x[Ø][2][2], x[Ø][2][3],
x[1][Ø][Ø], x[1][Ø][1], x[1][Ø][2], x[1][Ø][3],
x[1][1][Ø], x[1][1][1], x[1][1][2], x[1][1][3],
x[1][2][Ø], x[1][2][1], x[1][2][2], x[1][2][3],
```

gespeichert.

Die Multiplikation zweier Matrizen A und B verdeutlicht die Arbeit mit zweidimensionalen Feldern. Das Produkt einer Matrix $A_{(m,n)}$ mit m Zeilen und n Spalten und einer Matrix $B_{(n,p)}$ ist wie folgt definiert:

$$C_{(m,p)} = A_{(m,n)}B_{(n,p)}$$

Dabei ergeben sich die Elemente $c_{ij}$ von C aus dem inneren Produkt der i-ten Zeile von A und der j-ten Spalte von B:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}$$

Es folgt ohne weitere Erläuterung der Quelltext des Programms. Die dabei benutzten Funktionen getvekt bzw. getint wurden in den Abschnitten 2.2.1. bzw. 2.1.7. vorgestellt.

```
#include <stdio.h>

#define N 40

main()              /* Matrixmultiplikation */
{
    int i, j, k, m, n, p, a[N][N], b[N][N], c[N][N];

    printf("Multiplikation zweier Matrizen 'A(m,n)B(n,p)'\n");
    printf("\nGeben Sie die Werte fuer m, n und p ein (<= %d):", N);
    if(  (m = getint()) > 0 && m < N
      && (n = getint()) > 0 && n < N
      && (p = getint()) > 0 && p < N  ) {
        printf("\n%d Elemente von 'A' (zeilenweise):\n", m * n);
        getmat(a, m, n);
        printf("\n%d Elemente von 'B' (zeilenweise):\n", n * p);
        getmat(b, n, p);
        i = 0;
        while(i < m) {
            j = 0;
            while(j < p) {
                c[i][j] = 0;
                k = 0;
                while(k < n) {
                    c[i][j] += a[i][k] * b[k][j];
                    k++;
                }
                j++;
            }
            i++;
        }
        printf("\nDas Ergebnis von 'A x B' ist:");
        printf("\n*****************************\n");
        i = 0;
        while(i < m) {
            j = 0;
            while(j < p)
                printf("%5d ", c[i][j++]);
            i++;
            putchar('\n');
        }
    } else {
        printf("fehlerhafter Eingabewert\n");
    }
}

void getmat(mat, m, n)  /* Lesen einer (m,n)-Matrix */
    int mat[N][N];
    int m, n;
{
    int i;

    i = 0;
    while(i < m)
        getvekt(mat[i++], n);
}
```

**Aufgabe 2.10.** Programmieren und testen Sie die Funktion trans(s, a, b), die jedes im Zeichenfeld s auftretende Zeichen a durch das Zeichen b ersetzt.

**Aufgabe 2.11.** Üben Sie den Umgang mit Zeichenfeldern, indem Sie die oben aufgeführten Funktionen strlen, strcmp, strcat und strchr implementieren.

### 2.2.2. Strukturen

Unter einer **Struktur** wird in C eine Zusammenfassung von Objekten unterschiedlichen Typs verstanden. (Strukturen sind vergleichbar mit dem **record**-Typ in PASCAL.) Eingeleitet wird eine Strukturvereinbarung durch das Schlüsselwort **struct**.

```
struct datum {
    short jahr;
    char monat[4];
    short tag;
};
```

Hierdurch wird ein **Strukturtyp** mit dem Bezeichner datum[1] deklariert. Diese Deklaration reserviert keinen Speicherplatz, sondern legt lediglich die Typen der zu diesem Strukturtyp gehörenden **Komponenten** fest. Um eine konkrete **Instanz** dieses Strukturtyps zu definieren, kann auf den Strukturtypbezeichner datum Bezug genommen werden. Durch

```
struct datum heute, geb_datum;
```

erfolgt die Definition der Variablenbezeichner heute und geb_datum als Strukturinstanzen des Typs datum und damit auch die Speicherplatzvergabe. Jede Instanz enthält als Strukturkomponenten drei Variablen: jahr und tag sind Komponenten vom Typ **short**, während monat ein Zeichenfeld der Länge 4 darstellt. Der gleiche Effekt kann auch auf andere Weise erreicht werden:

```
struct datum {
    short jahr;
    char monat[4];
    short tag;
} heute, geb_datum;
```

Mit der Deklaration eines Strukturtyps kann also gleich die Definition von Variablen dieses Typs erfolgen. Syntaktisch gesehen ist das vergleichbar mit

```
int heute, geb_datum;
```

Übrigens kann in dem Fall der Strukturtypbezeichner weggelassen werden:

```
struct {
    ...
} heute, geb_datum;
```

Allerdings besteht dann nachfolgend keine Möglichkeit, weitere Instanzen dieses Strukturtyps zu definieren.

Der von einer Strukturinstanz belegte Speicherplatz ergibt sich nicht zwangsläufig aus der Summe des Speicherplatzbedarfes der zugehörigen Komponenten. Auf Grund eventueller Anforderungen hinsichtlich Ausrichtung (alignment) der Strukturkomponenten können zwischen einzelnen Komponenten "Löcher" entstehen. Ausserdem muss man sich im klaren darüber sein, dass die verschiedenen Maschinentypen unterschiedliche Ausrichtungsanforderungen stellen. Probleme können insbesondere auftreten, wenn man Strukturen als "Datensätze" auf externe Speichermedien (z.B. Magnetplatte) ausgibt

---

1) In /34/ wird hierfür der Begriff 'structure tag' verwendet.

und auf einem anderen Rechnertyp in eine Struktur gleichen Typs wieder einliest. Eine korrekte Zuordnung der gelesenen Daten zu den einzelnen Strukturkomponenten ist dabei nicht garantiert /50/.

Als Komponenten einer Struktur können beliebige Datentypen vereinbart werden, insbesondere auch Strukturen. Die einzige Einschränkung besteht darin, dass sich eine Struktur nicht selbst als Komponente enthalten kann:

```
struct datum {
    short jahr;
    char monat[4];
    short tag;
} heute, geb_datum;

struct adresse {
    int plz;
    char ort[1Ø];
    char str[2Ø];
    int nummer;
};

struct person {
    char name[1Ø];
    char vorname[1Ø];
    struct datum geb_datum;
    struct adresse wohnung;
};

struct person leute[1Ø], person;
```

Die letzte Vereinbarung verdient besondere Aufmerksamkeit. Sie definiert ein Feld leute mit 10 Elementen. Jedes Feldelement ist eine Struktur vom Typ person und enthält als Komponenten u.a. wiederum Strukturen der Typen adresse und datum.

Ausser dem Feld leute wird noch eine Variable mit dem Namen person als Instanz des Strukturtyps person definiert. Die Definition der Sprache erlaubt tatsächlich, dass für den Strukturtyp und irgendeine Variable die gleichen Bezeichner verwendet werden. In C ist die Namensgleichheit von Strukturkomponenten, Strukturtypen und "normalen" Variablen möglich. Man kann auch unterschiedliche Strukturen vereinbaren, in denen gleiche Bezeichner für Komponenten verwendet werden. Zum Beispiel würden zwischen dem oben deklarierten Strukturtyp person und

```
struct student {
    int sem_gruppe;
    char name[1Ø];
    char vorname[1Ø];
    struct datum geb_datum;
};
```

keine Namenskonflikte bezüglich deren Strukturkomponenten auftreten. Die einzige Forderung besteht darin, dass alle zu einer Struktur gehörenden Komponenten unterschiedliche Namen besitzen müssen.[1)]

Wie kann nun auf eine Struktur bzw. deren Strukturkomponenten zugegriffen werden, und welche Operationen sind mit Strukturen erlaubt?

---

1) Ältere Compilerversionen realisieren die in /34/ festgelegten Regeln. Danach ist gefordert, dass Strukturtypen und Strukturkomponenten alle voneinander verschiedene Namen besitzen müssen. Unterschiedliche Strukturen können lediglich mit der gleichen Komponentenfolge beginnen, ohne dass diesbezüglich Namenskonflikte auftreten.

Es gibt die Operatoren . und -> für den Zugriff auf einzelne Komponenten einer Struktur. ( Auf den Operator -> gehen wir nach Einführung des Zeigertyps im Abschn. 2.3.5. ein.)

Wenn beispielsweise heute eine Instanz des oben beschriebenen Strukturtyps datum bezeichnet, so weist

```
heute.jahr = 1987;
```

der **short**-Komponente jahr den Wert 1987 zu. Angenommen, leute ist als Feld von Strukturen des Typs person definiert. Dann kann man in analoger Weise wie oben schreiben:

```
leute[Ø].geb_datum.jahr = heute.jahr;
```

Formal handelt es sich bei derartigen Konstruktionen um Ausdrücke der Form:

```
<einfacher_ausdruck>.<bezeichner>
```

Dabei ist <einfacher_ausdruck> ein Bezeichner für eine Strukturinstanz und <bezeichner> der Name einer Komponente dieser Struktur. Der Ausdruck liefert diese Strukturkomponente und ist ein Lvalue genau dann, wenn links vom Operator auch ein Lvalue steht.[1)]

Strukturzuweisungen sind möglich, wenn es sich um Strukturen gleichen Typs handelt.

```
struct datum heute, geb_datum;
  ...
heute.jahr = 1987;
strcpy(heute.monat, "Mai");
heute.tag = 3;
geb_datum = heute;
```

Jeder Komponente von geb_datum wird der aktuelle Wert der korrespondierenden heute-Komponente zugewiesen.

Weiterhin kann eine komplette Struktur als Funktionsargument übergeben sowie als Funktionswert zurückgeliefert werden.

Es folgen zwei Funktionen, die mit den oben deklarierten Strukturen person, adresse und datum arbeiten. Herausgesucht werden alle Personen, die in einem bestimmten Monat Geburtstag haben. Die Daten dieser Mitarbeiter sind auszugeben und Mitarbeiter mit "runden" Geburtstagen (2Ø, 3Ø,...) in der Ausgabe besonders zu kennzeichnen.

Wir gehen von der Annahme aus, dass in einem Feld mit NMAX Elementen vom Typ person die Personaldaten der Mitarbeiter eines Betriebes gespeichert sind. (Wie diese Angaben erfasst und verwaltet werden, soll uns im Moment nicht interessieren.) Die Deklarationen der Strukturtypen datum, adresse und person sowie die Definition des Feldes person, deren Elemente Strukturinstanzen des Typs person sind, befinden sich am Anfang des Quelltextfiles. Dadurch ist es möglich, dass sich alle in diesem File enthaltenen Funktionen auf die Strukturtypen und das Feld beziehen können.

---

1) Die Sprachdefinition in /34/ erlaubt es, als linken Operanden einen beliebigen Lvalue anzugeben. Entsprechende Compilerversionen gehen von der Annahme aus, dass dadurch grundsätzlich eine Instanz eines Strukturtyps bezeichnet wird, in der der rechte Operand als Komponente enthalten ist!

```
#define NMAX 1ØØ

struct datum {
    short jahr;
    char monat[4];
    short tag;
};

struct  adresse  {
    int plz;
    char ort[1Ø];
    char str[2Ø];
    int nummer;
};

struct  person  {
    char name[1Ø];
    char vorname[1Ø];
    struct datum geb_datum;
    struct adresse wohnung;
};

struct person person[NMAX];

void geb_mon(mon, jahr) /* lineares Suchen nach Monat */
    char mon[];
    int jahr;
{
    int i;

    i = Ø;
    while(i < NMAX) {
        if(strcmp(mon, person[i].geb_datum.monat) == Ø)
            p_person(person[i], (jahr-person[i].geb_datum.jahr)%1Ø);
        i++;
    }
}

void p_person(p, mark)  /* Ausgabe der in person enthaltenen Daten */
    struct person p;
    int mark;
{
    if(mark == Ø)
        printf("***********************************************\n");
    printf(" %s, %s\n", p.name, p.vorname);
    printf(" %2d. %s %4d\n", p.geb_datum.tag, p.geb_datum.monat,
                                        p.geb_datum.jahr);
    printf("%d %s, %s %d\n", p.adresse.plz, p.adresse.ort,
                        p.adresse.str, p.adresse.nummer);
    if(mark == Ø)
        printf("***********************************************\n");
}
```

**Aufgabe 2.12.** Vervollständigen Sie dieses Beispiel um die main-Funktion, deren Aufgabe darin besteht, am Terminal eingegebene Personaldaten in ein Feld von Strukturen des Typs person zu speichern und mit Hilfe der Funktion geb_mon die "Geburtstagskinder" herauszufinden.

### 2.2.3. Bitfelder

Bitfelder, eine spezielle Form von Strukturkomponenten, werden auf Teile eines Maschinenwortes abgebildet. Sie erlauben den symbolischen Zu-

griff auf Bits und stellen somit eine zu den bitorientierten Operationen alternative Möglichkeit der Bitmanipulation dar.

Bezogen auf das Setzen, Löschen und Abfragen von Anzeigebits in einer **int**-Variablen flag soll die Arbeit mit Bitfeldern verdeutlicht werden.

```
struct anzeige  {
    unsigned bit_Ø : 1;
    unsigned bit_1 : 1;
    unsigned bit_2 : 1;
};

struct anzeige flag;
```

Die Strukturkomponenten bit_Ø, bit_1 und bit_2 stellen Bitfelder der Feldbreite 1 dar. Die allgemeine Form einer Bitfeldvereinbarung lautet:

```
<typ>  <bezeichner> : <konst_ausdruck>
```

Obwohl die Sprachdefinition dies nicht ausdrücklich fordert, sollten Bitfelder grundsätzlich vom Typ **unsigned** vereinbart werden. Die Feldbreite wird durch einen nichtnegativen <konst_ausdruck> festgelegt. Sie gibt die Anzahl der Bits an und ist durch die Anzahl der Bits eines Maschinenworts begrenzt. Ein Bitfeld belegt also maximal soviel Speicherplatz wie eine **int**-Grösse. Kann ein Bitfeld nicht mehr in einem Wort untergebracht werden, so erfolgt automatisch eine Ausrichtung auf die nächste Wortgrenze. Durch den speziellen Feldbreitenwert Ø kann man die Ausrichtung auf eine implementationsabhängige Grenze anfordern. Das Übergehen einer bestimmten Anzahl von Bits ist möglich, indem <typ> und <bezeichner> bei der Vereinbarung weggelassen werden.

Bitfelder können in arithmetischen Ausdrücken auftreten und verhalten sich wie vorzeichenlose Zahlen.

```
flag.bit_Ø = flag.bit_1 = flag.bit_2 = 1;  /* Setzen Bitfelder */
 ...

flag.bit_Ø = flag.bit_1 = flag.bit_2 = Ø;  /* Löschen Bitfelder */
 ...
flag.bit_Ø = 1;                            /* Setzen bit_Ø */
 ...

if(flag.bit_1) {                           /* bit_1 gesetzt? */
   ...
}

if(flag.bit_Ø  &&  !flag.bit_1  &&  !flag.bit_2) {
   ...                                     /* nur bit_Ø gesetzt? */
}
```

Einige Besonderheiten sind beim Umgang mit Bitfeldern zu beachten. Ob Bitfelder in einem Wort von links nach rechts oder von rechts nach links gespeichert werden, ist von der Rechnerarchitektur und der Compilerimplementation abhängig. Es ist nicht möglich, Felder zu definieren, deren Elemente Bitfelder sind. Bitfelder besitzen keine Adressen. Demzufolge sind Zeigeroperatoren nicht anwendbar.

**Aufgabe 2.13.** Ergänzen Sie die Strukturdeklaration person um Informationen über Familienstand und Geschlecht der jeweiligen Person. Verwenden Sie dafür Bitfelder, und erweitern Sie die Funktion p_person so, dass diese Angaben behandelt werden.

### 2.2.4. Unions

Eine weitere Form des Strukturtyps stellt der sog. Uniontyp dar. Dabei wird jede Komponente einer Union-Instanz auf den gleichen Speicherbereich abgebildet, d.h., alle Komponenten der Union beginnen mit der Verschiebung Ø bezüglich Union-Anfang. Der Speicherplatz, den eine Union belegt, wird von der "längsten" Komponente bestimmt. Bei der Vereinbarung einer Union gelten die gleichen syntaktischen Regeln wie für eine Struktur. Beispielsweise deklariert

```
union operand {
    int i;
    float f;
    long l;
};
```

einen Uniontyp operand, und

```
union operand u;
```

definiert die Instanz u dieses Typs. Auf Unions sind die gleichen Operationen anwendbar wie auf Strukturen. Unions können als Funktionsargumente übergeben und als Funktionswerte zurückgegeben werden, ebenso sind Zuweisungen von Unions erlaubt. Die Bezugnahme auf Union-Komponenten geschieht analog zu Strukturen. Mittels

```
u.l
```

erfolgt der Zugriff auf die Komponente l der Union u. Laut /50/ sollte eine Union-Komponente nur dann benutzt werden, wenn der aktuelle Wert der Union aus der Zuweisung eines Wertes an die gleiche Komponente resultiert.[1] Sicher hat es (bezogen auf obige Vereinbarung) keinen Sinn, der Komponente u.f einen Wert zuzuweisen und anschliessend mit u.i weiterzuarbeiten. Oft ist es zweckmässig, sich die "aktive" Union-Komponente mit Hilfe einer sog. varianten Struktur /19/ (vergleichbar mit dem Typ **variant record** in PASCAL) zu merken. Ausgehend von den Vereinbarungen

```
#define INT   1
#define LONG  2
#define FLOAT 3

struct operand {
    int typ;
    union variante  {
        int i;
        float f;
        long l;
    } value;
} op;
```

wäre folgende Arbeitsweise denkbar:

---

1) Von den meisten Compilern wird diese strenge Forderung nicht geprüft. Der Programmierer trägt selbst die Verantwortung, wenn er einen Wert eines Typs in die Union speichert und ihn später als Darstellung eines anderen Typs interpretiert. Auch wenn damit in einer konkreten Programmierumgebung sinnvolle Ergebnisse erzielt werden, ist eine solche Vorgehensweise nicht portabel.

```
...
op.typ = FLOAT;
op.value.f = 3.1415;
  ...
if(op.typ == INT) {
    printf("%d", op.value.i);
    ...
} else if(op.typ == LONG) {
    printf("%ld", op.value.l);
    ...
} else if(op.typ == FLOAT) {
    printf("%f", op.value.f);
    ...
} else
    fprintf(stderr,"Fehler\n");
```

Die Sprachdefinition in /50/ lässt folgendes zu: Es seien in einer Union als Komponenten mehrere Strukturen vereinbart, die alle mit der gleichen Folge von Strukturkomponenten beginnen:

```
#define INT   1
#define LONG  2
#define FLOAT 3

union variante {
    struct {
        int typ;
    } value;
    struct {
        int typ;
        int i;
    } value_i;
    struct {
        int typ;
        float f;
    } value_f;
    struct {
        int typ;
        long l;
    } value_l;
} ;

union variante op;
```

Wenn die Union op zu einem bestimmten Zeitpunkt eine der Strukturen enthält, dann kann man auf beliebige Komponenten der gemeinsamen Anfangsfolge zugreifen:

```
...
op.value_f.typ = FLOAT;
op.value_f.f = 3.1415;
  ...
if(op.value.typ == INT) {
    printf("%d", op.value_i.i);
    ...
} else if(op.value.typ == LONG) {
    printf("%ld", op.value_l.l);
    ...
} else if(op.value.typ == FLOAT) {
    printf("%f", op.value_f.f);
    ...
} else
    fprintf(stderr,"Fehler\n");
```

### 2.2.5. Aufzählungstyp

Es gibt die Möglichkeit, Datentypen zu definieren, indem der Wertebereich explizit durch Angabe einer Bezeichnerliste festgelegt wird. Zum Beispiel erfolgt über

```
enum tag  {
    montag, dienstag, mittwoch,
    donnerstag, freitag,
    sonnabend, sonntag
};
```

die Deklaration des Aufzählungstyps tag mit den möglichen konstanten Werten montag, dienstag, ... , sonnabend, sonntag. Die einzelnen Elemente eines Aufzählungstyps werden intern entsprechend ihrer Reihenfolge in den Bereich der natürlichen Zahlen Ø, 1, ... abgebildet. Einem Element kann jedoch auch durch einen Konstantenausdruck explizit ein bestimmter Integerwert zugeordnet werden:

```
enum tag  {
    montag, dienstag, mittwoch = 1ØØ,
    donnerstag, freitag,
    sonnabend, sonntag
};
```

In diesem Beispiel erhält montag den Wert Ø, dienstag den Wert 1, mittwoch den Wert 1ØØ, donnerstag den Wert 1Ø1 usw. Auf den Typbezeichner tag kann im weiteren Bezug genommen werden, um Instanzen dieses Aufzählungstyps zu definieren:

```
enum tag wochentag;
```

Hierdurch wird wochentag als Variable des Typs tag vereinbart.

Wie bei Strukturen ist es erlaubt, bei der Deklaration des Aufzählungstyps gleichzeitig auch Variablen dieses Typs zu definieren:

```
enum tag  {
    montag, dienstag, mittwoch,
    donnerstag, freitag,
    sonnabend, sonntag
} wochentag;
```

Schliesslich kann in diesem Fall der Typbezeichner tag fehlen:

```
enum  {
    montag, dienstag, mittwoch,
    donnerstag, freitag,
    sonnabend, sonntag
}  wochentag;
```

Hinsichtlich der Namensbildung gehören Bezeichner für **enum**-Elemente zur gleichen Klasse wie Bezeichner für Variablen, während **enum**-Typbezeichner zur Klasse der **struct**- und **union**-Typbezeichner gehören. Mit anderen Worten: **enum**-Elemente verschiedener Aufzählungstypen und "gewöhnliche" Variable müssen alle voneinander verschiedene Namen besitzen, hingegen treten zwischen Bezeichnern für **enum**-Elemente, Komponentenbezeichnern für Strukturen und Unions sowie Typbezeichnern für Aufzählungstypen, Strukturen und Unions keine Namenskonflikte auf.

Variablen eines Aufzählungstyps werden wie Integervariable behandelt, während die zu einer **enum**-Variablen gehörenden Elemente praktisch symbolische Integerkonstanten sind.[1] Die folgenden Anweisungen deuten an, wie die oben definierte **enum**-Variable wochentag benutzt werden kann:

```
...
wochentag = montag;
...
if(wochentag == sonnabend) {
    ...
} else if(wochentag == sonntag) {
    ...
} else {
    ...
}
```

**Aufgabe 2.14.** Zur Lösung von Aufgabe 2.13 könnte man anstelle von Bitfeldern auch **enum**-Variablen verwenden. Vergleichen und bewerten Sie beide Vorgehensweisen.

### 2.2.6. Typdefinitionen

Eine Vereinbarung der Form

**typedef** <typ> <bezeichner>

definiert, dass <bezeichner> als Synonym für <typ> benutzt werden soll. Solche Typdefinitionen sind für beliebige Datentypen einschliesslich Zeiger erlaubt. In den folgenden Beispielen werden für Namen neu definierter Typbezeichner ausschliesslich Grossbuchstaben verwendet, um sie hervorzuheben. Es besteht dazu allerdings keine Notwendigkeit, da sie den gleichen Regeln der Namensbildung unterliegen wie Bezeichner für Variablen. Die Namen solcher Typbezeichner müssen sich jedoch von allen Variablenbezeichnern unterscheiden, da sie zur gleichen Klasse von Namen gehören.

```
typedef int INDEX, FLAG, BOOLEAN;
typedef char TEXT[100];
typedef struct  {
    short jahr;
    char monat[4];
    short tag;
}  DATUM;
```

Die hiermit eingeführten neuen Typbezeichner INDEX, FLAG und BOOLEAN können nachfolgend verwendet werden, um **int**-Variablen zu vereinbaren, z.B.

```
INDEX i, j, k;
FLAG anzeige;
BOOLEAN x;
```

---

[1] Das Programm lint prüft, ob in Ausdrücken mit **enum**-Variablen und **enum**-Elementen Datenobjekte anderer Typen vorkommen bzw. andere Operationen als die Wertzuweisung, der Test auf Gleichheit oder Ungleichheit angewendet werden. In diesen Fällen liefert lint eine Warnung.

Das gleiche gilt für den Typ TEXT, der nichts weiter als ein Synonym für ein Zeichenfeld mit 100 Elementen darstellt, sowie den Strukturtyp DATUM:

```
TEXT string;
DATUM heute, geb_datum;
```

Durch **typedef** werden **keine** neuen Datentypen erzeugt, sondern lediglich Synonyme für andere (bereits existierende) Typen. Nicht möglich ist es, mittels **typedef** eingeführte Typbezeichner mit anderen Typbezeichnern zu kombinieren /50/, wie z.B.

```
typedef int INDEX;
...
long INDEX n;       /*  Fehler!  */
```

Typdefinitionen werden hauptsächlich dazu verwendet, die Lesbarkeit von C-Programmen zu erhöhen. Ausserdem dienen sie zur Unterstützung der Portabilität, indem maschinenabhängige Typen mittels **typedef** vereinbart werden. Bei der Übertragung auf einen anderen Rechnertyp müssen diese Typdefinitionen entsprechend modifiziert werden.

### 2.2.7. sizeof-Operator

Der unäre Operator **sizeof** liefert einen Wert vom Typ **unsigned**, der den Speicherplatz (gemessen in Bytes) angibt, den der Operand benötigt. Es sind zwei syntaktische Formen erlaubt:

```
sizeof <ausdruck>
sizeof(<typ_spezifikation>)
```

Aus Gründen einer einheitlichen Benutzung beider Formen wird empfohlen, <ausdruck> in runde Klammern zu setzen:

```
sizeof(<ausdruck>)
```

Ein durch den **sizeof**-Operator gebildeter Ausdruck ist ein Konstantenausdruck. Er wird bereits während der Compilierung ausgewertet, wobei der Compiler den konkreten Speicherplatzbedarf aus den Vereinbarungen der betreffenden Objekte ableitet. Wendet man **sizeof** z.B. auf einen Feldbezeichner an, liefert dieser Konstantenausdruck die Länge des gesamten Feldes:

```
int a[10], n;
  ...
n = sizeof(a);
```

Der in diesem Beispiel n zugewiesene Wert entspricht der Byteanzahl, die bei der Speicherung von 10 **int**-Elementen belegt werden.

```
int n;
struct datum  {
    short jahr;
    char monat;
    short tag;
}  heute;
   ...
n = sizeof(heute);
```

Der Ausdruck **sizeof**(heute) liefert die Anzahl der Bytes, die durch heute (eine Instanz des Strukturtyps datum) belegt werden. Zu beachten ist, dass dieser Wert bedingt durch eventuelle Ausrichtungsforderungen der Komponententypen möglicherweise grösser ist als die Summe der Speicherplatzwerte der einzelnen Strukturkomponenten.

Als Operand von **sizeof** kann auch ein Typ angegeben werden, z.B.

```
sizeof(int)
sizeof(struct datum)
```

Der Wert eines solchen Ausdrucks entspricht dem Speicherplatzbedarf, den ein Objekt des angegebenen Typs benötigt. Die exakte Beschreibung der möglichen syntaktischen Formen von <typ_spezifikation> wird im Zusammenhang mit der expliziten Typkonvertierung (s. Abschn. 2.8.4.) erläutert.

## 2.3. Zeigertyp

Zeiger enthalten die Adresse eines Objekts eines bestimmten Datentyps. Somit ist es möglich, auf diese Objekte "indirekt" mittels Zeiger zuzugreifen. In der Sprache C spielen Zeiger eine zentrale Rolle. Im Gegensatz zu anderen Programmiersprachen existieren bei der Arbeit mit Zeigern praktisch kaum Einschränkungen.

### 2.3.1. Vereinbarung

Die Vereinbarung des Bezeichners p als Zeiger auf Objekte vom Typ **int** geschieht folgendermassen:

```
int *p;
```

Die allgemeine Form einer Zeigervereinbarung

```
<speicherklasse>  <typ>  *<bezeichner>
```

soll an weiteren Beispielen verdeutlicht werden:

```
char *cp;              /* Zeiger auf Typ char */
int x, *px;            /* x ist int-Variable, px ist Zeiger auf int */
float *fp[1Ø];         /* Feld von Zeigern auf Typ float */
struct datum *dat;     /* Zeiger auf Strukturtyp datum */
struct datum *pd[2Ø]; /* Feld von Zeigern auf Typ datum */
char **ppc;            /* Zeiger auf Zeiger auf Typ char */
```

Selbstverständlich können auch Strukturkomponenten vom Zeigertyp sein.

```
struct tnode  {
    int op;
    int typ;
    struct tnode *ptleft;
    struct tnode *ptright;
};
```

Diese Strukturdeklaration beschreibt den Aufbau eines Knotens in einem binären Baum. Die Struktur tnode enthält zwei Zeiger ptleft und ptright, die selbst auf derartige Knoten verweisen können. Wie man sieht, sind Strukturtypbezeichner notwendig, um rekursive Strukturen zu vereinbaren.

Durch

```
struct tnode tree[2Ø], *ptnode;
```

wird ein Feld tree definiert, das aus 20 Knotenelementen besteht, sowie ein Zeiger ptnode, der als Werte die Adressen von tnode-Strukturen annehmen kann.

### 2.3.2. Zeigeroperatoren

Der unäre Operator & kann auf einen Lvalue angewendet werden

```
&<lvalue>
```

und liefert die Adresse seines Operanden. Bei einem Operandentyp <typ> ist das Ergebnis der Operation vom Typ <zeiger auf typ>.

Mit dem unären Operator * kann man eine Adressbezugnahme auflösen, d.h. auf das Datenobjekt "indirekt" zugreifen, auf das der Operand verweist:

```
*<zeiger>
```

Der Ausdruck liefert einen Lvalue. Besitzt der Operand den Typ <zeiger auf typ>, so ergibt sich für den Ausdruck der Ergebnistyp <typ>. Betrachten wir z.B. die Vereinbarung

```
int x, y, *pint;
```

Zunächst ordnen wir x den Wert 1Ø und pint die Adresse von x zu:

```
x = 1Ø;
pint = &x;          /* pint "zeigt" auf x */
```

Jetzt kann über pint auf den Wert von x Bezug genommen werden.

```
y = *pint;          /* identisch zu y = x; */
```

Nach dieser Operation besitzt y den gleichen Wert wie x, nämlich 1Ø. Allgemein gilt, dass überall dort, wo in Ausdrücken x vorkommen kann, auch *pint benutzt werden kann.

```
y = *pint - 1;      /* y = x - 1; */
*pint = Ø;          /* x = Ø; */
*pint += 2;         /* x += 2; */
```

### 2.3.3. Beziehung zwischen Zeigern und Feldern

In C stehen Felder in enger Beziehung zu Zeigern. Gegeben sei

```
int f[N], i, *pint;
```

Man kann dem Zeiger pint die Adresse des ersten Elements von f zuweisen durch:

```
pint = &f[Ø];
```

Kommt ein Feldbezeichner in einem Ausdruck vor, so wird er vom Compiler grundsätzlich in einen Zeiger auf das erste Element des Feldes transformiert. Somit könnte auch geschrieben werden

```
pint = f;
```

Weiterhin gilt: Falls pint auf irgendein Feldelement von f zeigt, so bezieht sich der Ausdruck

```
pint + 1
```

auf das **nächste** Feldelement und

```
pint - 1
```

auf das **vorhergehende** Feldelement. Allgemein bedeutet ein Ausdruck

```
pint + i
```

die Adresse desjenigen Feldelements, das sich i Elemente nach dem durch pint adressierten Element befindet, während

```
pint - i
```

auf ein Element zeigt, das i Elemente davor gespeichert ist. Daraus ergeben sich einige Konsequenzen. Zeigt pint auf f[Ø], so liefert der Ausdruck

```
pint + i
```

die Adresse von f[i] und

```
*(pint + i)
```

bezieht sich auf den Wert von f[i]. Da andererseits der Feldbezeichner f in einen Zeiger auf f[Ø] umgewandelt wird, ist

```
f[i]
```

nur eine andere Notation für

```
*(f + i)
```

Demzufolge ist

```
&f[i]
```

äquivalent zu

```
f + i
```

Umgekehrt gilt, dass ein Zeiger auch indiziert werden kann. Anstelle von

```
*(pint + i)
```

könnte man auch schreiben:

```
pint[i]
```

Allerdings besteht keine allgemeine Identität zwischen einem Feldbezeichner f und einem Zeiger pint. Da f in Ausdrücken wie eine Adresskonstante behandelt wird, können auf f keine Operatoren angewendet werden, die einen Lvalue als Operand erfordern. Nicht zulässig sind z.B. solche Ausdrücke wie &f, f = ... oder f++ usw.

Häufig kommt der Zeigeroperator * in Ausdrücken zusammen mit Inkrement- und Dekrementoperatoren vor.

```
int f[N], *p1, *p2, *p3, *p4, x;
int i;
  ...
i = Ø;
p1 = p2 = p3 = p4 = &f[i];
```

Unter diesen Voraussetzungen betrachten wir folgende Anweisungen:

```
x = *p1++;          /*  analog zu x = f[i++];  */
x = *++p2;          /*  analog zu x = f[++i];  */
x = (*p3)++;        /*  analog zu x = f[i]++;  */
x = ++*p4;          /*  analog zu x = ++f[i];  */
```

Der erste Eindruck beim Betrachten dieser Anweisungen ist möglicherweise etwas zwiespältig. Da jedoch derartige Konstruktionen sehr häufig in C-Programmen zu finden sind, zumindest die beiden oberen, soll hier etwas genauer darauf eingegangen werden.

Zunächst sind zwei Fragen zu klären: Worauf bezieht sich die Inkrementierung, auf den Zeigerwert oder auf den Speicherplatzinhalt, den der Zeiger adressiert? Wann erfolgt die Wertzuweisung, vor oder nach der Inkrementierung? Um die zweite Frage beantworten zu können, muss man sich die Wirkungsweise der Post- bzw. Präfixinkrementierung in Erinnerung rufen, und zur Beantwortung der ersten Frage ziehen wir die Assoziativität zu Rate, da die Operatoren * und ++ den gleichen Vorrang besitzen. Nichtgeklammerte Ausdrücke, die mehrere dieser Operatoren enthalten, werden so ausgewertet, als ob sie von rechts geklammert wären.

Demzufolge ist

```
*p1++
```

gleichbedeutend zu

```
*(p1++)
```

Die Inkrementierung bezieht sich also auf den Zeiger. Der Wert von *p1++ entspricht dem Feldelement, auf das p1 vor der Postfixinkrementierung zeigt. Der Haupteffekt des Ausdrucks

```
x = *p1++
```

besteht in der Zuweisung des Wertes des i-ten Feldelements an die Variable x. Als Nebeneffekt wird p1 die Adresse des nächsten Elements von f zugeordnet.

Anstelle von

```
x = *++p2;
```

könnte man auch

```
x = *(++p2);
```

schreiben. Als erstes erfolgt die Inkrementierung von p2 und anschliessend die Auflösung der Adressbezugnahme, so dass an x der Wert des (i+1)-ten Feldelements zugewiesen wird.

Im Unterschied zur ersten Anweisung bezieht sich die Inkrementierung bei

```
x = (*p3)++;
```

auf *p3, d.h. den Wert des Feldelements und nicht auf den Zeiger p3. Da es sich hierbei um die Postfixnotation handelt, erhält x den Wert des Integerelements vor der Inkrementierung.

Schliesslich erfolgt bei

```
x = ++*p4;
```

die Inkrementierung des Feldelements vor der Wertzuweisung.

Verdeutlichen wir die Arbeit mit Zeigern an einigen Beispielen. Als erstes soll ein Zeichenfeld kopiert werden.

```
char von[N], nach[N];
char *pv, *pn;
  ...
pv = von;
pn = nach;
while((*pn = *pv) != '\Ø') {
    pv++;
    pn++;
}
```

Betrachten wir die **while**-Schleife etwas genauer. Zunächst kann auch hier der Vergleichsausdruck kürzer geschrieben werden:

```
while(*pn = *pv) {
    pn++;
    pv++;
}
```

Jetzt beziehen wir die Inkrementierung der beiden Zeiger in den Vergleich ein:

```
while(*pn++ = *pv++)
    ;
```

Es erfolgt die Ersetzung des Feldelements, auf das sich pn vor der Inkrementierung bezieht, durch den Wert von *pv++, d.h. das Zeichen, auf das pv gerade zeigt. Als Nebeneffekt werden beide Zeiger inkrementiert, so dass sie auf das jeweils nächste Element in dem betreffenden Feld zeigen.

Bei einem Feld als Funktionsargument wird die Adresse des ersten Feldelements übergeben. Aus dieser Tatsache ergibt sich die Möglichkeit, den entsprechenden formalen Parameter als Zeiger zu deklarieren, wie folgende Variante von strcpy verdeutlicht (vgl. Abschn. 2.2.1.).

```
void strcpy(s1, s2) /* Kopieren s2 nach s1 (Zeigerversion) */
    char *s1, *s2;
{
    while(*s1++ = *s2++)
        ;
}
```

Die Funktionen zur Zeichenkettenverarbeitung zeigen sehr anschaulich den Umgang mit Zeigern. Deshalb geben wir abschliessend eine Version von stromp an, die zwei Zeichenketten lexikographisch miteinander vergleicht.

```
int strcmp(a, b)    /* Vergleich der Zeichenketten a und b */
    char *a, *b;
{
    while(*a == *b++)
        if(*a++ == '\Ø')
            return(Ø);
    return(*a - *--b);
}
```

Die Funktion liefert den Wert Ø, wenn beide Zeichenketten aus der gleichen Zeichenfolge bestehen. Sonst ergibt sich der Funktionswert aus der Differenz des numerischen Wertes der beiden Zeichen, in denen a und b nicht mehr übereinstimmen. Ist dieser Wert positiv, so ist die Zeichenkette a im lexikographischen Sinn "grösser" als b, bei einem negativen Wert ist a lexikographisch "kleiner" als b.

**Aufgabe 2.15.** Programmieren Sie Zeigerversionen der im Abschn. 2.2.1. erwähnten Funktionen strlen, strncmp, strcat, und vergleichen Sie die Ergebnisse mit denen von Übungsaufgabe 2.11.

### 2.3.4. Zeigerarithmetik

Die Arithmetik mit Zeigern beruht auf den Regeln der Behandlung von Feldern und ist auch nur sinnvoll im Zusammenhang mit Objekten, die Elemente ein und desselben Feldes darstellen.

Erlaubt sind der Vergleich (==, !=, <, <=, >, >=) und die Subtraktion zweier Zeigerwerte sowie die Subtraktion und Addition eines Zeiger- und eines Integerwertes. Grundsätzlich erfolgt die Arithmetik in Speichereinheiten des Typs, für den der Zeiger definiert ist. Sei p ein Zeiger auf den Typ <typ>, so bewirkt z.B.

```
p++                 /*  bzw.  p = p + 1  oder auch  p += 1  */
```

die Erhöhung des Wertes von p um die Anzahl von Bytes, die ein Objekt des Typs <typ> belegt. Für <typ> gleich **char** bedeutet das die Addition von 1, für <typ> gleich **int** bedeutet das abhängig vom Maschinentyp die Addition von 2 bzw. 4 usw.

Gegeben seien ein Integerwert i, ein Zeiger p auf <typ> sowie ein Ausdruck

```
p += i              /*  bzw.  p = p + i  */
```

Bei der Ausdrucksauswertung wird i zunächst mit der Länge von <typ> multipliziert und der Wert anschliessend zum Zeigerwert p addiert. Das Ergebnis ist wieder ein Zeigerwert. Analoges gilt für die Subtraktion eines Integerwertes von einem Zeigerwert:

```
p -= i              /* p = p - i */
```

Wenn zwei Zeiger px und py auf Elemente des gleichen Feldes zeigen, dann ergibt die Subtraktion einen Integerwert, und zwar die Anzahl der Elemente zwischen px und py (das Element *px mitgerechnet). Das gilt

wiederum unabhängig vom Typ der Feldelemente. Die folgende Funktion verkettet zwei Zeichenfelder miteinander und liefert als Ergebnis die Gesamtlänge, d.h. die Anzahl der Zeichen.

```
int catlen(a, b)   /* Zeichenfelder verketten und Länge berechnen */
    char *a, *b;
{
    char *p;

    p = a;
    while(*p)
        p++;
    while(*p++ = *b++)
        ;
    return(--p - a);
}
```

Man beachte die Dekrementierung des Zeigers p vor der Subtraktion von a. Sie ist notwendig, da das Endekennzeichen \Ø nicht mitgerechnet wird. Eine spezielle Bedeutung besitzt der Zeigerwert NULL. Er ist laut Definition der einzige Integerwert, der einer Zeigervariablen zugewiesen oder mit dem eine Zeigervariable verglichen werden kann. Besitzt ein Zeiger den Wert NULL, so ist garantiert, dass der Zeiger auf kein Datenobjekt zeigt. (Demzufolge ist auch der "indirekte" Zugriff mittels * auf Zeiger mit diesem Wert nicht erlaubt.) Das gilt unabhängig vom Typ der Objekte, auf die sich die Zeigervariable bezieht. Gewöhnlich ist NULL im File stdio.h als Ø definiert, es sollte jedoch grundsätzlich NULL benutzt werden.

**Aufgabe 2.16.** Schreiben Sie eine Zeigerversion von strlen.

**Aufgabe 2.17.** Entwerfen Sie die Funktionen zur Berechnung des Skalarprodukts neu, wobei die Arbeit mit den Feldern über Zeigervariable erfolgt.

### 2.3.5. Zeiger auf Strukturen

Der Zugriff auf Strukturkomponenten ist auch indirekt über Zeiger möglich. Betrachten wir hierzu folgende Vereinbarungen und Anweisungen:

```
struct tnode  {
    int count;
    int *pint;
    struct tnode *ptleft;
    struct tnode *ptright;
};

struct tnode tree[2Ø], *ptnode;
int i, x, f[2Ø];

  ...
i = Ø;
ptnode = &tree[i];
  ...
(*ptnode).count = Ø;
```

Die Zeigervariable ptnode zeigt auf das erste Element des Feldes tree. Jedes Feldelement stellt selbst eine Instanz des Strukturtyps tnode dar, so dass

```
(*ptnode).count
```

die Strukturkomponente count auswählt, der der Wert Ø zugewiesen wird. Man beachte die Klammern in diesem Ausdruck, sie sind auf Grund des höheren Vorrangs des Operators . gegenüber dem Operator * notwendig. Da diese Schreibweise recht aufwendig und unbequem ist und ausserdem in der Praxis häufig über Zeiger auf Strukturkomponenten zugegriffen wird, existiert für diesen Zweck der Operator -> :

```
ptnode->count = Ø;
  ...
ptnode->ptleft = &tree[i+1];
ptnode->ptleft->count = ...;
```

Der linke Operand muss ein Zeiger auf eine Struktur sein, der rechte Operand bezeichnet eine Komponente dieser Struktur.[1)]

Mit den folgenden Beispielen soll an einige Aspekte der Zeigerarithmetik, des Vorrangs und der Assoziativität der Operatoren ->, *, ++ sowie an die Wirkungsweise der Präfix- bzw. Postfixinkrementierung erinnert werden. Obige Vereinbarungen und Zuweisungen vorausgesetzt, ist die Wirkungsweise folgender Anweisungen identisch:

```
ptnode++;
++ptnode;
ptnode = ptnode + 1;
ptnode += 1;
```

Jedesmal wird hierbei der Zeiger ptnode um einen Wert inkrementiert, der dem Speicherplatzbedarf eines Feldelements von tree entspricht.

Der Vorrang von -> ist höher als der des Operators ++ :

```
++ptnode->count;    /* entspricht ++(ptnode->count); */
```

Demzufolge wird count inkrementiert (und nicht etwa ptnode). Um die Inkrementierung von ptnode zu erreichen, muss die Klammerung erfolgen, beispielsweise entspricht

```
x = (++ptnode)->count;
```

den beiden Anweisungen

```
++ptnode;           /* oder: ptnode++; */
x = ptnode->count;
```

Bei Verwendung der Postfixnotation erfolgt die Inkrementierung von ptnode nach der Wertzuweisung, demzufolge ist die Wirkung von

```
x = (ptnode++)->count;
```

gleich der von

```
x = ptnode->count;
ptnode++;           /* oder: ++ptnode; */
```

---

1) Einige Compilerversionen lassen folgende "Typaufweichungen" entsprechend /34/ zu: Wenn der linke Operand irgendeinen Zeiger- oder Integerwert besitzt, so wird angenommen, dass sich dieser Wert auf eine Instanz desjenigen Strukturtyps bezieht, in welcher der rechte Operand als Strukturkomponente enthalten ist.

Ausserdem können in diesem Fall die Klammern entfallen:

```
x = ptnode++->count;
```

In der Annahme, dass den Elementen des oben definierten Feldes f irgendwelche Werte zugewiesen worden sind und pint auf ein Element dieses **int**-Feldes zeigt,

```
f[Ø] = f[1] = ... = f[19] = ... ;
ptnode->pint = &f[i];
```

betrachten wir folgende Anweisungen:

```
x = *ptnode->pint;
```

Die Adressauflösung bezieht sich auf Grund der Vorrangregeln auf pint, d.h., x wird der Wert des **int**-Elements zugewiesen, auf das pint zeigt.
Semantisch gesehen sind die Anweisungen

```
x = *ptnode->pint++;
```

und

```
x = *ptnode->pint;
ptnode->pint++;
```

identisch. Analog hierzu können die Anweisungen

```
ptnode->pint++;     /* oder ++ptnode->pint; */
x = *ptnode->pint;
```

verkürzt folgendermassen geschrieben werden:

```
x = *++ptnode->pint;
```

Angenommen, die Inkrementierung soll sich nicht auf den Zeiger pint, sondern auf den Integerwert beziehen, auf den ptnode->pint gerade zeigt. Das erreicht man folgendermassen (Postfix- bzw. Präfixinkrementierung):

```
x = (*ptnode->pint)++;     /* x = *ptnode->pint; (*ptnode->pint)++; */
x = ++*ptnode->pint;       /* ++*ptnode->pint; x = *ptnode->pint; */
```

Schliesslich kann sich die Inkrementierung auch auf ptnode beziehen:

```
x = *ptnode++->pint;       /* x = *ptnode->pint; ptnode++; */
x = *(++ptnode)->pint;     /* ++ptnode; x = *ptnode->pint; */
```

**Aufgabe 2.18.** Überarbeiten Sie die in Abschn. 2.2.2. vorgestellten Funktionen geb_mon und p_person dahingehend, dass der Funktion p_person anstelle einer Strukturinstanz die Adresse der Struktur übergeben wird.

**Aufgabe 2.19.** Gegeben sei

```
struct tnode **p;        /* Deklaration von tnode s. Seite 69 */
...
(*p)->count = Ø;
```

Warum sind im Ausdruck (*p)->count die Klammern erforderlich?

## 2.4. Steuerstrukturen

Die sprachlichen Mittel zur Steuerung des Programmablaufs ermöglichen es, gut strukturierte Programme zu schreiben. Syntax und Semantik sind mit der Zielstellung entworfen, dass der Programmierer kompakte und gut lesbare Quellprogramme erstellen kann. Vor allem soll die Generierung effizienten Zielkodes durch den Compiler unterstützt werden.

Ausser den bereits diskutierten Anweisungen **if-else**, **do-while** und **while** existieren weitere Steueranweisungen, die teilweise syntaktischen und semantischen Besonderheiten unterliegen:

- die **switch**-Anweisung zur Mehrwegeauswahl
- die universelle Schleifenanweisung **for**
- die Anweisungen **break**, **continue** und **goto** zur unbedingten Steuerungsübergabe.

### 2.4.1. switch-Anweisung

Die allgemeine Form der Mehrwegeauswahl lautet:

```
switch(<ausdruck>)  {
        case <konst_ausdruck_1> :
                <anweisungs_liste_1>
          ...
        case <konst_ausdruck_i> :
                <anweisungs_liste_i>
        default :
                <anweisungs_liste_d>
        case <konst_ausdruck_i+1> :
                <anweisungs_liste_i+1>
          ...
        case <konst_ausdruck_n> :
                <anweisungs_liste_n>
}
```

Dabei stellt die Konstruktion <anweisungs_liste$_i$> eine Folge von Anweisungen der Form

<anweisung$_1$> ... <anweisung$_m$>

dar oder ist leer. Insbesondere können fehlen

der **default**-Zweig oder
die **case**-Zweige **vor** dem **default**-Zweig oder
die **case**-Zweige **nach** dem **default**-Zweig.

Die **case**-Werte, d.h. die Werte der Konstantenausdrücke <konst_ausdruck$_i$>, müssen voneinander verschieden sein. Ausserdem ist gefordert, dass die Auswertung von <ausdruck> einen Integerwert liefert. Der **switch**-Wert wird nacheinander so lange mit den einzelnen **case**-Werten verglichen, bis Gleichheit festgestellt wird. In dem Fall beginnt die Abarbeitung bei der zu diesem **case**-Zweig gehörenden Anweisungsfolge. Am Ende eines **case**-Zweigs wird die Verarbeitung der **switch**-Anweisung nicht automatisch been-

det, sondern die Anweisungsfolge des darauffolgenden **case**- oder **default**-Zweigs ausgeführt. Um die Ausführung an einer bestimmten Stelle, z.B. am Ende eines **case**- bzw. des **default**-Zweigs, unbedingt abzubrechen, kann die **break**-Anweisung (Abschn. 2.4.3.) benutzt werden. Stimmt keiner der **case**-Werte mit dem **switch**-Wert überein und existiert der **default**-Zweig, so werden die zugehörigen Anweisungen und anschliessend die Anweisungen eventuell nachfolgender **case**-Zweige abgearbeitet.

Die Anwendung der **switch**-Anweisung zeigen wir am Beispiel eines Filekopierprogramms, das Folgen von Leerzeichen bzw. Tabulatoren jeweils durch genau ein Leerzeichen ersetzt.

```
#include <stdio.h>

#define YES 1
#define NO  0

main()              /* Folgen von ' ' bzw. '\t' durch ' ' ersetzen */
{
    int c;
    int inleer;

    inleer = NO;
    while((c = getchar()) != EOF)
        switch(c)  {
            case '\t':
            case ' ' :
                if(inleer == NO)  {
                    inleer = YES;
                    putchar(' ');
                }
                break;
            default:
                inleer = NO;
                putchar(c);
                break;
        }
}
```

Jedes gelesene Zeichen wird getestet, ob es ein Leerzeichen oder Tabulator ist. In beiden Fällen erfolgt die Abarbeitung der **if**-Anweisung. Die **break**-Anweisung im **case**-Zweig verhindert, dass als nächstes mit den Anweisungen des **default**-Zweigs fortgesetzt wird. Sie bewirkt den Abbruch der **switch**-Verarbeitung, dadurch erfolgt das Lesen des nächsten Zeichens. Die **break**-Anweisung am Ende des **default**-Zweigs ist zwar nicht notwendig, sollte aber vorsichtshalber kodiert werden, um mögliche Fehler bei einer Erweiterung der **switch**-Anweisung von vornherein auszuschliessen.

Das nächste Beispiel zählt die Zeilen, Zeichen und Worte eines Files. Unter einem "Wort" wird dabei eine beliebig lange Zeichenfolge verstanden, die durch Leerzeichen, Tabulator oder Newline abgeschlossen ist. Es verdeutlicht, dass **switch**-Konstruktionen eine kompakte, weitgehend redundanzfreie Kodierung erlauben. Der einzige Unterschied zwischen der Behandlung von Newline, Leerzeichen bzw. Tabulator besteht darin, dass bei Newline der Zeilenzähler um 1 erhöht wird. In allen drei Fällen muss der Schalter inwort (sicherheitshalber) auf NO gestellt werden.

```
#include <stdio.h>
#define YES 1
#define NO Ø

main()              /* Zeilen, Worte, Zeichen eines Files zählen */
{
    int c, inwort, zeilen, worte;
    long zeichen;

    inwort = NO;
    zeilen = worte = zeichen = Ø;
    while((c = getchar()) != EOF)  {
        zeichen++;
        switch(c)  {
            case '\n':
                zeilen++;
            case ' ' :
            case '\t':
                inwort = NO;
                break;
            default:
                if(inwort == NO)  {
                    inwort = YES;
                    worte++;
                }
                break;
        }
    }
    printf("%d  %d  %ld\n", zeilen, worte, zeichen);
}
```

**Aufgabe 2.20.** Vergleichen Sie die **switch**-Anweisung mit der Anweisungsfolge **if-else-if**. Welche Unterschiede bestehen zwischen beiden Formen der Mehrwegeauswahl?

### 2.4.2. for-Anweisung

Die **for**-Anweisung der Sprache C hat folgende Syntax:

**for**(<ausdruck$_1$>; <ausdruck$_2$>; <ausdruck$_3$>)
    <anweisung>

Dabei entspricht <ausdruck$_1$> der Schleifeninitialisierung, <ausdruck$_2$> ist der Wiederholungstest, und durch <ausdruck$_3$> erfolgt die Wiederinitialisierung der **for**-Schleife.

Ein einfaches Beispiel stellt die Verarbeitung eines Feldes von N Elementen dar.

```
int f[N];
  ...
for(i = Ø; i < N; i++)
    f[i] = Ø;
```

Die Ausdrücke einer **for**-Konstruktion können beliebige Ausdrücke sein. Das wird deutlich, wenn man die folgende zu **for** semantisch äquivalente [1]

---

1) Die Äquivalenz ist genaugenommen nur dann gegeben, wenn innerhalb des Schleifenkörpers keine **continue**-Anweisung auftritt (s. Abschn. 2.4.3.).

Anweisungsfolge betrachtet:

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

Zunächst wird die **for**-Schleife initialisiert, indem <ausdruck$_1$> berechnet wird. Anschliessend erfolgt die Prüfung, ob <ausdruck$_2$> einen Wert verschieden von Ø besitzt. Trifft das zu, wird der zugehörige Anweisungsteil abgearbeitet, <ausdruck$_3$> ausgewertet (Wiederinitialisierung) und der Wiederholungstest erneut durchgeführt. Das geschieht zyklisch so lange, bis schliesslich <ausdruck$_2$> den Wert Ø besitzt.

Die folgenden Beispiele verdeutlichen einige Aspekte der Anwendung von **for** im Zusammenhang mit der Verarbeitung von Feldern.

Die Funktion index(z, c) prüft, ob das Zeichen c in dem Zeichenfeld z vorkommt, und liefert als Ergebnis den Index oder -1, falls c nicht in z enthalten ist.

```
int index(z, c)     /* Index von c innerhalb des Zeichenfeldes z */
    char z[], c;
{
    int i;

    for(i=Ø; z[i] != c && z[i] != '\Ø'; i++)
        ;
    return(z[i] == c ? i : -1);
}
```

In diesem Beispiel besteht der Schleifenkörper aus einem einzelnen Semikolon, d.h. einer leeren Anweisung.

In einer **for**-Konstruktion können auch einzelne oder alle Ausdrücke weggelassen werden. Zu beachten ist jedoch, dass die Semikolons angegeben werden müssen. Die obige Funktion könnte man beispielsweise dahingehend modifizieren, dass sie die Suche nach dem Zeichen c von rechts beginnt und somit den Feldindex des letzten Vorkommens von c in z als Funktionswert liefert.

```
int rindex(z, c)    /* Index des letzten Auftretens von c in z */
    char z[], c;
{
    int i;

    for(i = Ø; z[i]; i++)
        ;
    for(; i >= Ø; i--)
        if(z[i] == c)
            return(i);
    return(-1);
}
```

Mit der ersten **for**-Anweisung wird auf das Ende des Zeichenfeldes z positioniert, indem nach dem Endekennzeichen \Ø gesucht wird. Dann enthält i den entsprechenden Indexwert und muss deshalb in der zweiten **for**-Konstruktion nicht initialisiert werden.

Bei der Behandlung des Kommaoperators im Abschn. 2.1.6. wurde bereits erwähnt, dass er häufig in **for**-Konstruktionen Anwendung findet. Zu diesem

Zweck soll eine dritte Variante der index-Funktion gezeigt werden. Sie unterscheidet sich von den vorhergehenden Versionen dadurch, dass in z nicht nach einem einzelnen Zeichen, sondern nach einem Zeichenfeld gesucht wird. Mit anderen Worten: mindex prüft, ob ein bestimmtes Muster in einer Zeichenfolge enthalten ist.

```
mindex(z, f)        /* Index des Musters f in z */
    char z[], f[];
{
    int i, j, n;

    for(i = Ø; z[i]; i++) {
        for(j = i, n = Ø; z[j] == f[n] && z[j]; j++, n++)
            ;
        if(f[n] == '\Ø')
            return(i);
    }
    return(-1);
}
```

Die gleichzeitige Verarbeitung zweier Indizes innerhalb einer **for**-Schleife stellt einen typischen Anwendungsfall des Kommaoperators dar. Mittels

```
j = i, n = Ø
```

werden j und n initialisiert. Analog erfolgt am Ende jedes Schleifendurchlaufs das Erhöhen beider Indexwerte durch den Ausdruck

```
j++, n++
```

**Aufgabe 2.21.** Überarbeiten Sie das Programm zur Multiplikation von Matrizen, indem anstelle der **while**-Konstruktionen **for**-Anweisungen verwendet werden. Begründen Sie, warum **for** in diesem Zusammenhang günstiger ist.

**Aufgabe 2.22.** Untersuchen Sie weitere Beispiele, die **while**-Schleifen enthalten, und entscheiden Sie sich für die Verwendung von **for** oder **while**.

### 2.4.3. Anweisungen zur unbedingten Steuerungsübergabe

Es gibt folgende Anweisungen zur unbedingten Steuerungsübergabe:

```
break;
continue;
goto <marke>;
```

Die **break**-Anweisung wird verwendet, um die Abarbeitung einer unmittelbar übergeordneten **switch**-, **while**-, **do-while**- bzw. **for**-Anweisung abzubrechen. Man kann beispielsweise mittels **break** eine "unendliche" **for**-Schleife verlassen.

```
for (;;) {
    ...
    if (<ausdruck>)
        break;

}
```

Häufig ist **break** innerhalb von **switch**-Anweisungen als letzte Anweisung eines **case**- oder **default**-Zweiges erforderlich, um den automatischen Übergang zum Anweisungsteil des darauffolgenden Zweiges zu verhindern.

Eine **continue**-Anweisung bewirkt das Einleiten der nächsten Iteration der umgebenden Schleife. Demzufolge wird bei **while** bzw. **do-while** als nächstes der Schleifentest ausgeführt, d.h. der Vergleichsausdruck ausgewertet. Bei einer **for**-Schleife hingegen erfolgt vor dem Wiederholungstest zunächst die Wiederinitialisierung. (Insofern besteht also doch ein semantischer Unterschied zwischen einer **for**-Schleife und der im Abschn. 2.4.2. angegebenen **while**-Konstruktion.)

Eine dritte Möglichkeit der unbedingten Steuerungsübergabe besteht in der Verwendung einer Anweisung

**goto** <marke>;

Fortgesetzt wird mit derjenigen Anweisung, die durch <marke>: als Sprungziel einer **goto**-Anweisung markiert ist:

<marke>: <anweisung>

Grundsätzlich kann jede Anweisung markiert sein. Dabei muss sich <marke>: im gleichen Funktionsblock wie die **goto**-Anweisung befinden, d.h., Sprünge sind nur innerhalb einer Funktion erlaubt. Für Marken gelten die gleichen Regeln der Namensbildung wie für Variablenbezeichner. Bezeichner für Marken sind implizit für den gesamten Funktionsblock gültig.

Die Anwendung von **goto** ist eigentlich nur sinnvoll, um abhängig von einer Bedingung aus tieferen Schachtelungsstufen nach aussen zu gelangen.

Wirkungsweise und Nutzung von **break**, **continue** und **goto** sollen an einem Beispiel gezeigt werden. Das folgende Programm liest ein File und gibt die einzelnen Zeilen zeichenverkehrt aus, d.h. das letzte Zeichen einer Zeile als erstes, das vorletzte Zeichen als zweites usw. Der Einfachheit halber wird angenommen, dass eine Zeile nicht mehr als 512 Zeichen enthält.

```
#include <stdio.h>
#define MAX 512

main()              /* zeichenverkehrte Ausgabe von Zeilen */
{
    char zeile[MAX];
    int i;

    for(;;)  {
        for(i=Ø; i < MAX; i++)  {
            switch(zeile[i] = getchar())  {
                case '\n':
                    break;
                case EOF :
                    goto ende;
                default :
                    continue;
            }
            break;
        }
        while(--i >= Ø)
            putchar(zeile[i]);
        putchar('\n');
    }
    ende: ;
}
```

Die Untersuchung eines gelesenen Zeichens erfolgt in der **switch**-Anweisung. Wird Newline erkannt, so liegt eine vollständige Zeile vor. In diesem Fall muss die **switch**-Verarbeitung abgebrochen (erstes **break**) und ausserdem die innere **for**-Schleife beendet werden (zweites **break**), um zur Ausgabe überzugehen (**while**-Schleife). Bei Fileende reicht **break** nicht aus, da auch die äussere, "unendliche" **for**-Schleife verlassen werden muss. Bei einem beliebigen anderen Zeichen wird mittels **continue** zur Wiederinitialisierung des inneren **for** gesprungen.

## 2.5. Funktionen

Die bisher im Umgang mit Funktionen erworbenen Kenntnisse werden in diesem Abschnitt zusammengefasst und vervollständigt bis auf Fragen der Speicherklasse von Funktionen und Parametern bzw. Probleme der Typkonvertierung von Funktionsargumenten und -werten, auf die wir in den Abschnitten 2.6. bzw. 2.8. eingehen werden.

Ein C-Programm besteht aus einer Menge von Funktionsdefinitionen, die in beliebiger Folge angeordnet sein können. Funktionsdefinitionen dürfen jedoch nicht geschachtelt werden. Die Funktionen eines Programms können über mehrere Quelltextfiles verteilt sein, solange eine einzelne Funktion vollständig in einem File enthalten ist. Es ist möglich, jedes dieser Files separat zu übersetzen. Mehrere im Ergebnis getrennter Übersetzungen entstandene Objektfiles können (zusammen mit den Objekten von Bibliotheksfunktionen) zu einem kompletten Programm verbunden werden.

Die Ausführung eines Programms beginnt grundsätzlich bei der **main**-Funktion.

### 2.5.1. Funktionsdefinition

Eine Funktionsdefinition besteht aus dem Funktionskopf und dem Funktionskörper. Im Kopf sind die Informationen enthalten, die für die Benutzung der Funktion erforderlich sind:

Name der Funktion
Reihenfolge und Typ der formalen Parameter
Typ des Funktionsergebnisses.

Der Funktionskörper ist ein Block, der sog. Funktionsblock. Er enthält Variablenvereinbarungen und Anweisungen, die in geschweifte Klammern eingeschlossen sind.

Die allgemeine syntaktische Form einer Funktionsdefinition lautet:

```
<speicherklasse> <typ> <funktions_name>(<parameter_liste>)
      <parameter_deklarations_liste>
{
      <vereinbarungs_liste>
      <anweisungs_liste>
}
```

Was das Funktionsergebnis betrifft, so sind zwei Fragen zu beantworten. Welche Typen sind erlaubt? Auf welche Weise erfolgt die Typvereinbarung?

Als Typen von Funktionsergebnissen sind möglich:

- alle Grundtypen, wie **char**, **int**, **unsigned**, **float**, **double** usw.
- Strukturen und Unions
- Zeiger auf beliebige Objekte einschliesslich Zeiger auf Funktionen (s. Abschn. 2.5.5.).

Nicht zulässig ist die Rückgabe von Feldern und Funktionen.

Die Typfestlegung für das Funktionsergebnis erfolgt bei der Funktionsdefinition in analoger Weise wie bei der Vereinbarung von Variablen. Standardmässig wird der Typ **int** angenommen, so dass in diesem Fall die Typangabe **int** nicht zwingend ist. Es wird jedoch empfohlen, das Typattribut **int** bei der Funktionsdefinition stets anzugeben:

```
int f(...)
    ...
{

}
```

Um das Prinzip der Festlegung des Typs für das Funktionsergebnis zu erklären, betrachten wir folgende Fragmente von Funktionsdefinitionen:

```
float f1(...)
{

}
struct datum f2(...)
{

}
int *f3(...)
{

}
struct datum *f4(...)
{
    ...
}
int (*f5(...)) []
{
    ...
}
int (*f6(...)) ()
{

}
struct datum (*f7(...)) []
{

}
struct datum (*f8(...)) ()
{

}
struct datum *(*f9(...)) ()
{

}
struct datum (*(*f1Ø(...))()) []
{

}
```

Die einzelnen Funktionen liefern Werte folgender Typen:

f1: **float**
f2: Strukturtyp datum
f3: Zeiger auf **int**
f4: Zeiger auf Strukturtyp datum
f5: Zeiger auf ein Feld von **int**-Werten
f6: Zeiger auf eine Funktion, die selbst einen **int**-Wert zurückgibt
f7: Zeiger auf ein Feld von datum-Strukturen
f8: Zeiger auf eine Funktion, die eine datum-Struktur liefert
f9: Zeiger auf eine Funktion, die Zeiger auf den Typ datum liefert
f1Ø: Zeiger auf eine Funktion, die einen Zeiger auf ein Feld von datum-Strukturen liefert

Das Hauptproblem bei der Konstruktion solcher Typspezifikationen besteht in der korrekten Klammerung unter Beachtung des Vorrangs und der Assoziativität der Operatoren (),[] bzw. * entsprechend Abschn. 2.1.7. Der Operator () zur Vereinbarung einer Funktion besitzt den gleichen Vorrang wie der Operator [] zur Vereinbarung eines Feldes. Eine Folge dieser Operatoren wird von rechts nach links abgearbeitet. Der Vorrang des Zeigeroperators * ist geringer als der von () bzw. []. Weiterhin ist * rechtsassoziativ, d.h., stehen mehrere *-Operatoren nacheinander, so erfolgt die Auswertung von rechts nach links. Ausserdem muss man beachten, dass * vor dem zugehörigen Operanden steht (Präfixoperator), während () und [] Postfixoperatoren sind und demzufolge nach dem Operanden auftreten.

Betrachten wir z.B. die Aufgabe, eine Funktion f zu definieren, die als Ergebnis einen Zeiger auf ein Feld von Zeigern auf Zeichen liefert. Dabei kann man die Definition schrittweise von "innen" nach "aussen" unter Beachtung von Vorrang und Assoziativität wie folgt aufbauen:

```
f(...)                  f ist eine Funktion
*f(...)                 die einen Zeiger
(*f(...))[]             auf ein Feld
*(*f(...))[]            von Zeigern
char *(*f(...))[]       auf Zeichen liefert
```

Solche komplizierten Typvereinbarungen für Funktionsergebnisse treten in der Praxis äusserst selten auf. Die folgende Funktion strchr durchsucht ein Zeichenfeld nach einem bestimmten Zeichen und liefert als Ergebnis den Zeigerwert der entsprechenden Position. Wenn das gesuchte Zeichen überhaupt nicht vorkommt, dann soll der Zeigerwert NULL zurückgegeben werden. Nebenbei bemerkt, ist strchr eine Funktion der Standardbibliothek.

```
char *strchr(s, c)      /* Zeiger auf c in s */
    char *s, c;
{
    do {
        if(*s == c)
            return(s);
    } while(*s++);
    return(NULL);
}
```

In der Sprache C wird nicht zwischen den Begriffen Funktion und Prozedur unterschieden wie in vielen anderen Programmiersprachen. Man spricht auch dann von einer Funktion, wenn kein Funktionswert oder genauer ein undefinierter Wert an die aufrufende Funktion zurückgegeben wird. Funktionen, die keinen Funktionswert liefern, werden bei der Definition durch das

spezielle Typattribut **void** [1] gekennzeichnet, z.B.

```
void f(...)
    ...
{

}
```

Für die Bildung von <funktions_name> gelten analoge Regeln wie für Variablenbezeichner, wobei die Anzahl der signifikanten Zeichen implementationsabhängig ist.

Nach dem Funktionsnamen sind, in runde Klammern eingeschlossen und jeweils durch Komma getrennt, die formalen Parameter anzugeben. Es folgen die Typdeklarationen für die einzelnen Parameter, wobei für Parameter vom Typ **int** keine Deklarationen notwendig sind. Beispielsweise wird durch

```
void f(x, y, z)
    int x, *y;
    struct datum z;
{

}
```

eine Funktion f mit drei formalen Parametern definiert; x ist ein **int**-Parameter, y wird als Zeiger auf **int**-Werte und z als Struktur vom Typ datum deklariert. (Voraussetzung ist, dass dieser Strukturtyp vorher vereinbart worden ist.)

Eine Besonderheit liegt vor, wenn ein Feldbezeichner als Argument an eine Funktion übergeben werden soll.

```
int f()
{
    int x[10];
    ...
    g(x);
    ...
}

void g(feld)
    int feld[];
{

}
```

Bei der Deklaration des Parameters feld als Feld von Integerwerten wird die Dimensionsangabe weggelassen. Handelt es sich um mehrdimensionale Felder, so müssen jedoch ausser der ersten alle weiteren Dimensionsangaben vorhanden sein.

Wie im nächsten Abschnitt genauer erläutert wird, erfolgt die Übergabe eines Feldbezeichners als Zeiger auf das erste Feldelement. Demzufolge kann man feld auch als Zeiger auf Werte vom Typ **int** deklarieren:

```
void g(feld)
    int *feld;
{

}
```

---

[1] Da der Typ **void** nachträglich in das Typkonzept aufgenommen wurde, unterstützen verschiedene Compiler diesen Typ nicht.

**Aufgabe 2.23.** Üben Sie den schrittweisen Aufbau komplizierter Typspezifikationen für Funktionsergebnisse am Beispiel der Funktionen f5, f6, ..., f1Ø.

**Aufgabe 2.24.** Warum ist die Funktionsdefinition

```
int *f(...)[]
{

}
```

nicht korrekt?

### 2.5.2. Funktionsaufruf und Argumentübergabe

Ein Funktionsaufruf ist ein Ausdruck, dessen Wert sich aus dem Funktionsergebnis der gerufenen Funktion ergibt. Die Typspezifikation bei der Definition der Funktion bestimmt den Typ des Ausdruckswertes. Die rufende Funktion muss dabei eine Typdeklaration für die zu rufende Funktion enthalten. Die Syntax einer solchen Deklaration entspricht der Syntax von Funktionsdefinitionen mit dem Unterschied, dass die Parameterliste weggelassen wird.

```
void g()
{
    double *f(), *p;
    ...
    p = f(a, b);
    ...
}

double *f(x, y)
    ...
{

}
```

Ein vorher noch nicht vereinbarter Bezeichner wird implizit als Name einer Funktion deklariert, wenn er in einem Ausdruck vorkommt und unmittelbar danach eine öffnende runde Klammer folgt. Ausserdem wird in diesem Fall als Ergebnistyp der Funktion **int** angenommen. Deshalb müssen Funktionen, die Werte vom Typ **int** zurückgeben, in der rufenden Funktion nicht explizit deklariert werden.

Der Aufruf einer Funktion erfolgt somit durch Angabe des Funktionsnamens und der in runden Klammern eingeschlossenen Argumentliste. Die einzelnen Argumente sind durch Komma zu trennen, dabei handelt es sich nicht um einen Kommaoperator. Die Sprachdefinition von C lässt ausdrücklich offen, ob die Auswertung der Funktionsargumente von links nach rechts oder von rechts nach links vorgenommen wird. Demzufolge ist die Reihenfolge der Argumentauswertung abhängig von der Compilerimplementation.

Besitzt die gerufene Funktion keine Parameter, so muss eine leere Argumentliste angegeben werden. Argumente können beliebige Ausdrücke sein, wobei man darauf achten sollte, dass der Typ eines Arguments mit dem Typ des entsprechenden formalen Parameters übereinstimmt. Eine Prüfung auf Übereinstimmung von Argument- und Parametertyp wird durch den Compiler jedoch nicht vorgenommen.

Der Compiler untersucht auch nicht, ob die Anzahl der Argumente der Parameteranzahl entspricht. Eine Funktion kann im Prinzip mit einer variablen Anzahl von Funktionsargumenten gerufen werden. Das setzt voraus, dass eine Konvention existiert, aus der die gerufene Funktion erkennen kann, wie viele Argumente beim Aufruf übergeben worden sind. Das typische Beispiel einer solchen Funktion stellt printf dar. Dabei legt die Anzahl der im ersten Argument enthaltenen Konvertierungsvorschriften fest, wieviel weitere Argumente übergeben werden (s. Abschn. 3.3.2.). Zum Beispiel werden beim folgenden Aufruf von printf vier Argumente übergeben.

```
int zeilen, worte;
long zeichen;
...
printf("%d  %d  %ld\n", zeilen, worte, zeichen);
...
```

Die Funktion printf untersucht die Zeichenkettenkonstante

```
"%d  %d  %ld\n"
```

und stellt dadurch fest, dass noch drei weitere Argumente folgen.

In C erfolgt die Argumentübergabe für Objekte irgendeines arithmetischen Typs, für Strukturen und Unions sowie für Zeiger als Wert (call by value). Genaugenommen wird ein Parameter der gerufenen Funktion mit einer Kopie des entsprechenden Argumentwertes initialisiert, d.h., die Wertekopie wird im Kellerspeicher (Stack) abgelegt. Eine Änderung des Parameterwertes in der gerufenen Funktion besitzt keinerlei Auswirkung auf den ursprünglichen Argumentwert.

Bei der Übergabe von Feldern, Zeichenkettenkonstanten und Funktionen werden Zeiger auf das betreffende Objekt übermittelt (call by reference). Zum Beispiel erfolgt bei der Angabe eines Feldbezeichners als Argument die Zuweisung der Adresse des ersten Feldelements an den entsprechenden Parameter. Damit ist es möglich, über diesen Zeigerwert auf die einzelnen Elemente des Feldes zuzugreifen und deren Werte zu modifizieren.

**Aufgabe 2.25.** Schreiben Sie eine Funktion swap(x, y), die die Werte ihrer Argumente vertauscht.

### 2.5.3. Übergabe des Funktionswertes

Die Abarbeitung einer Funktion endet bei einer Anweisung der Form

```
return <ausdruck>;
```

Hierbei wird der Wert von <ausdruck> berechnet und vor der Rückgabe an die rufende Funktion eventuell in den Typ konvertiert, der bei der Funktionsdefinition festgelegt wurde.

Funktionen, die keinen Funktionswert zurückgeben, beenden ihre Arbeit durch eine Anweisung

```
return;
```

bzw. beim Erreichen der schliessenden geschweiften Klammer des Funktionsblockes. Dabei ist zu beachten, dass in diesem Fall auch der Ausdruckswert des Funktionsaufrufes in der rufenden Funktion undefiniert ist.

### 2.5.4. Rekursiver Aufruf von Funktionen

Funktionen können in C rekursiv aufgerufen werden. Angewendet wird dies oft bei rekursiv definierten mathematischen Verfahren, wie z.B. die Berechnung der Potenz $x^i$ einer reellen Zahl x und einer ganzen Zahl i:

$x^i = x*x^{i-1}$ für $i>0$
$x^i = 1/x^{-i}$ für $i<0$ und $x<>0$
$x^i = 1$ für $i=0$ und $x<>0$

Hier ist, direkt aus der Definition abgeleitet, der Quelltext der rekursiven Funktion power:

```
double power(x, i)      /* Berechnung von x^i (rekursiv) */
    double x;
    int i;
{
    if(i == 0)
        return(1);
    if(i < 0)
        return(1/power(x, -i));
    else
        return(x * power(x, i-1));
}
```

Eine wichtige Rolle für die rekursive Anwendung einer Funktion spielen die Daten, über die die Funktion operiert. Bei power werden die Variablen x und i bei jedem Funktionseintritt neu angelegt. Rekursiv wird also nur der Programmtext genutzt. Betrachtet man beispielsweise die Berechnung von $2^3$, so ergeben sich insgesamt 4 Aufrufe von power. Bild 2.3 zeigt die rekursive Aufruffolge und die zugehörige Folge der Funktionswerte.

Mit jeder Rekursion werden die Befehlsfolgen für den Funktionsaufruf (entry sequence) und die Funktionsbeendigung (exit sequence) durchlaufen. Ausserdem wird jeweils Kellerspeicher für die Funktionsargumente benötigt. Allgemein gilt, dass rekursive Lösungen speicherplatz- und laufzeitintensiver als iterative Lösungen, dafür jedoch meist kompakter und eleganter sind.

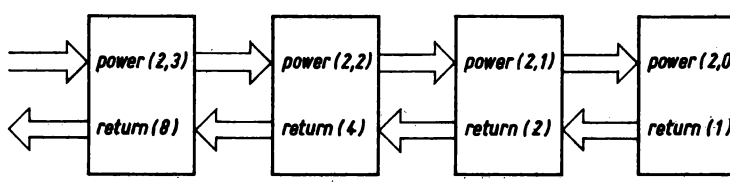



Bild 2.3. Rekursiver Funktionsaufruf und Ergebnisrückgabe

Als weiteres Beispiel zeigen wir die rekursive Funktion search, die in einem sortiert vorliegenden Feld array mit n Elementen nach einem Wert key sucht. Das Ergebnis der Suche ist die Adresse des gefundenen Feldelements oder der Zeigerwert NULL, wenn die Suche erfolglos war. Als Suchverfahren wählen wir das sog. binäre Suchen (binary search). Hierbei wird das Element in der Mitte des Feldes mit key verglichen. Bei Gleichheit ist die

Suche erfolgreich beendet. Ist das Feldelement grösser als key, so erfolgt die Fortsetzung des Suchprozesses in gleicher Weise in der ersten Feldhälfte, ansonsten in der zweiten Feldhälfte.

Die folgende Version von search geht davon aus, dass das Feld Elemente vom Typ **int** enthält.

```
int *search(key, array, n)   /* Binärsuche von key im Feld array */
    int key, array[], n;
{
    int mid;

    if(n <= 0)
        return(NULL);
    mid = n / 2;
    if(key == array[mid])
        return(&array[mid]);
    if(key < array[mid])
        return(search(key, array, mid-1));
    else
        return(search(key, &array[mid+1], n-mid-1));
}
```

**Aufgabe 2.26.** Entwerfen Sie eine rekursive Funktion zur Berechnung von n! (Hinweis: n! = n*(n-1)! bei n>0 und 0! = 1).

**Aufgabe 2.27.** Entwickeln Sie die rekursive sowie die nichtrekursive Version einer Funktion, die den grössten gemeinsamen Teiler zweier positiver ganzer Zahlen n und m nach dem euklidschen Algorithmus bestimmt, und vergleichen Sie beide Versionen.

```
ggt(n, m) = ggt(m, n)        für  m>n
ggt(n, m) = n                für  m=0
ggt(n, m) = ggt(m, n%m)      sonst
```

### 2.5.5. Zeiger auf Funktionen

Es ist erlaubt, einen Zeiger auf eine Funktion zu vereinbaren. Zum Beispiel wird durch

```
int (*zfint)();
```

der Bezeichner zfint als Variable vereinbart, die als Werte die Adressen von Funktionen annehmen kann, die selbst **int**-Funktionswerte liefern. Angenommen, f(a,b) sei eine solche Funktion, dann kann durch die Anweisung

```
zfint = f;
```

an zfint die Adresse der Funktion f zugewiesen und anschliessend diese Funktion indirekt aufgerufen werden über

```
x = (*zfint)(y, z);
```

Nach Ausführung dieser Anweisung enthält x den Funktionswert, der sich bei Anwendung der Funktion f auf die Argumente y und z ergibt, vorausgesetzt x wurde als **int**-Variable vereinbart.

Wenn ein Funktionsname in einem Ausdruck vorkommt und keinen Funktionsaufruf darstellt, so wird er als Zeigerwert dieser Funktion interpretiert. Der Operator & ist in diesem Fall nicht notwendig. Zeiger auf Funktionen können benutzt werden, um Funktionen als Funktionsargumente zu übergeben.

Die Funktion search im vorhergehenden Abschnitt ist auf das Suchen in Feldern mit **int**-Elementen beschränkt. Wir wollen die Problemstellung in zwei Richtungen verallgemeinern: Erstens soll die Funktion in der Lage sein, Elemente beliebiger Typen zu behandeln, und zweitens soll der Nutzer dieser Funktion selbst festlegen können, nach welchen Kriterien die Elemente sortiert sind. Die folgende Funktion bsearch realisiert den gleichen Funktionsumfang wie die Funktion gleichen Namens der Standardbibliothek (vgl. Anhang B).

```
char *bsearch(key, base, nel, w, compar)   /* Binäres Suchen */
    char *key;          /* Zeiger auf gesuchtes Element */
    char *base;         /* Zeiger auf erstes Element des Feldes */
    int nel;            /* Anzahl der Elemente */
    int w;              /* Elementlänge in Byte */
    int (*compar)();    /* Zeiger auf Vergleichsfunktion */
{
    int i, mid;

    if(nel <= 0)
        return(NULL);
    mid = nel/2;
    if((i = (*compar)(key, &base[mid*w])) == 0)
        return(&base[mid*w]);
    if(i < 0)
        return(bsearch(key, base, mid-1, w, compar));
    else
        return(bsearch(key, &base[(mid+1)*w], nel-mid-1, w, compar));
}
```

Angenommen, es liegt ein Feld z mit n in lexikographischer Folge geordneten Zeichen z[0], ..., z[n-1] vor und es soll geprüft werden, ob sich das Zeichen c darunter befindet. Ausserdem sei die Funktion ccomp für den Vergleich zweier Zeichen gegeben:

```
int ccomp(pa, pb)       /* Vergleich, ob *pa < *pb */
    char *pa, *pb;
{
    return(*pa - *pb);
}
```

Unter diesen Voraussetzungen könnte der Aufruf von bsearch folgendermassen aussehen:

```
...
char z[n], c, *pc;
int ccomp();
...
if((pc = bsearch(&c, z, n, sizeof(char), ccomp)) == NULL)
    printf("Das Zeichen '%c' ist in z nicht vorhanden\n", c);
else {
    ...                 /* Zeichen wurde gefunden */
}
```

## 2.6. Speicherklasse, Gültigkeit und Lebensdauer

Wie in früheren Abschnitten bereits erwähnt, wird bei der Vereinbarung von Variablen, Funktionen und formalen Parametern ein Speicherklassenattribut vergeben. Die **Speicherklasse** beeinflusst den Gültigkeitsbereich (scope) dieser Objekte und bestimmt die Lebensdauer (lifetime) von Variablen und Parametern. Bei einer Variablenvereinbarung der Form

<speicherklasse> <typ> <bezeichner$_1$>, ..., <bezeichner$_n$>;

sind für <speicherklasse> folgende Angaben möglich:

**extern**
**auto**
**static**
**register**

Bei fehlender Speicherklassenangabe werden bestimmte Standardannahmen getroffen. Erfolgt die Vereinbarung einer Variablen in einem Block (Funktionsblock oder geschachtelter Block), so wird standardmässig **auto** angenommen. Dagegen wird einer ausserhalb von Funktionen definierten Variablen implizit das Attribut **extern** zugeordnet. Funktionen erhalten bei ihrer Definition standardmässig die Speicherklasse **extern**, wenn nicht explizit **static** angegeben wird. Für die explizite Angabe von <speicherklasse> gibt es einige Einschränkungen. Erfolgt die Variablenvereinbarung ausserhalb einer Funktionsdefinition, so sind die Attribute **auto** und **register** nicht erlaubt. Die einzig mögliche Speicherklassenangabe für Funktionsparameter ist **register**.

Die Syntax von C lässt Vereinbarungen der Form

<speicherklasse> <bezeichner$_1$>, ..., <bezeichner$_n$>;

zu. In diesen Fällen wird für <typ> implizit **int** angenommen.

Jeder Bezeichner ist durch seinen **Gültigkeitsbereich** charakterisiert, unabhängig davon, ob es sich um Variablen, Parameter, Funktionen, Marken, Aufzählungs-, Struktur- bzw. Uniontypbezeichner, Elemente solcher Typen oder um Typbezeichner handelt, die durch **typedef** spezifiziert wurden. Unter dem Gültigkeitsbereich versteht man dabei den Abschnitt des Programmtextes, in dem der Bezeichner sichtbar ist, d.h. Bezugnahmen auf ihn möglich sind. Die Sichtbarkeit eines Bezeichners hängt insbesondere davon ab, an welcher Stelle im Quellprogramm die Vereinbarung vorgenommen wird. Der Programmierer kann z.B. festlegen, dass eine Variable in einem Block, einer Funktion, einem Quelltextfile oder allen zum Programm gehörenden Files bekannt ist. Der Gültigkeitsbereich von Marken und Parametern erstreckt sich auf den entsprechenden Funktionsblock. Normalerweise sind Funktionsnamen in allen Quelltextfiles des Programms bekannt. Die Sichtbarkeit einer Funktion kann aber auch auf das Quelltextfile beschränkt werden, in dem diese Funktion definiert ist.

Unter dem Begriff **Lebensdauer** ist der Zeitraum zu verstehen, in dem eine Variable oder ein Parameter während der Programmausführung existiert. Funktionsparameter existieren jeweils für die Dauer der Funktionsabarbeitung, während sich die Lebensdauer von Variablen über den gesamten Zeitraum der Ausführung des Programms, einer Funktion oder eines Blockes erstrecken kann.

### 2.6.1. Speicherklasse auto

Der Gültigkeitsbereich von Variablen mit der Speicherklasse **auto** entspricht dem Block, in dem die Vereinbarung erfolgt. Von ausserhalb kann auf diese Variablen nicht zugegriffen werden.

Datenobjekte mit der Speicherklasse **auto** existieren jeweils so lange, wie der Block, in dem sie vereinbart sind, aktiv ist. Obwohl für Parameter von Funktionen die Speicherklassenangabe **auto** nicht erlaubt ist, unterscheiden sie sich von **auto**-Variablen praktisch nur durch ihre Anfangswerte. Variablen besitzen bei jedem Eintritt in den vereinbarten Block zunächst undefinierte Werte, während Parameter beim Aufruf der Funktion mit ihrem Argumentwert initialisiert werden. Mit dem Verlassen des betreffenden Blocks verlieren **auto**-Variablen und Parameter ihre bisherigen Werte. Es ist nicht möglich, ausserhalb von Funktionen **auto**-Variablen zu definieren.

Wenn in einer Funktion eine **auto**-Variable definiert wird, so ist es üblich, das Schlüsselwort **auto** wegzulassen. Alle bisherigen Beispiele arbeiten mit Variablen, denen dieses Speicherklassenattribut implizit zugeordnet ist. Die folgende Funktionsdefinition verdeutlicht die Möglichkeiten, Variablen mit der Speicherklasse **auto** zu vereinbaren:

```
void f(x, n)
    int x[], n;
{                                   /* Beginn Funktionsblock */
    auto int i, y;
    ...
    i = 0;
    y = x[i];
    ...
    {                                       /* Beginn Block 1 */
        auto int z;
        ...
        z = x[i];
        ...
    }                                       /* Ende Block 1 */
    ...
}                                   /* Ende Funktionsblock */
```

Bezogen auf dieses Beispiel erstreckt sich die Gültigkeit von x, n, i und y über den gesammten Funktionsblock, während auf z nur innerhalb des geschachtelten Blocks (Block 1) zugegriffen werden kann. Die Parameter x und n besitzen beim Eintritt in den Funktionsblock die Werte der entsprechenden Aufrufargumente. Für i, y und z sind explizite Wertzuweisungen notwendig. Man beachte, dass die Variable z beim Verlassen von Block 1 ihren Wert verliert, d.h., bei einem erneuten Eintritt in diesen Block ist der Wert von z zunächst undefiniert. Nebenbei bemerkt ist die Vereinbarung

```
auto int i, y;
```

äquivalent zu

```
int i, y;
```

bzw.

```
auto i, y;
```

### 2.6.2. Speicherklasse extern

Jedes C-Programm besteht aus einer Reihe externer Objekte, d.h. Funktions- und Variablendefinitionen. Externe Objekte können das Speicherklassenattribut **extern** oder **static** besitzen. Ausserhalb von Funktionen vereinbarte Variablen erhalten implizit das Attribut **extern**, wenn keine Speicherklasse angegeben ist. In diesem Fall spricht man von einer **Definition** der Variablen. Genauso wie bei Funktionsnamen ist die Anzahl signifikanter Zeichen bei Bezeichnern für **extern**-Variablen implementationsabhängig.[1)]

```
file1.c:

void f()
{

}

int x;

void g()
{

}
```

```
file2.c:

void h()
{

}

void k()
{

}
```

Die Gültigkeit einer **extern**-Variablen überdeckt zunächst den Abschnitt vom Punkt der Definition bis zum Ende des Quelltextfiles. Im Beispiel kann x nur in der Funktion g benutzt werden. Weder in f noch im File file2.c ist diese Variable bekannt. Um auf sie auch in f zugreifen zu können, muss x in dieser Funktion entsprechend **deklariert** werden. Das geschieht durch explizite Angabe des Schlüsselwortes **extern**.

```
file1.c:

void f()
{
    extern int x;
    ...
}

int x;

void g()
{

}
```

Den gleichen Effekt kann man erreichen, indem die Variablendefinition am Anfang des Quelltextfiles vorgenommen wird. Um den Gültigkeitsbereich einer so definierten Variablen auf andere Quelltextfiles auszudehnen, ist dort eine **extern**-Deklaration erforderlich. Dabei muss in der Deklaration das gleiche Typattribut wie in der Definition angegeben werden.

---

1) Laut Sprachdefinition können bei Bezeichnern für Variablen mit dem Speicherklassenattribut **extern** weniger als 8 Zeichen signifikant sein!

file1.c:

```
int x;

void f()
{

}

void g()
{

}
```

file2.c:

```
extern int x;

void h()
{

}

void k()
{

}
```

Jetzt kann man sowohl in der Funktion h als auch in k auf x Bezug nehmen. Schliesslich kann der Zugriff auf eine bestimmte Funktion oder einen Block beschränkt werden, beispielsweise auf die Funktion h:

file2.c:

```
void h()
{
    extern int x;
    ...
}

void k()
{

}
```

Jede Variable mit dem Speicherklassenattribut **extern** muss in einem der zum Programm gehörenden Files definiert werden. Die **Definition** einer **extern**-Variablen muss sorgsam von einer **Deklaration** unterschieden werden. Mit der Definition ist die Vergabe von Speicherplatz für diese Variable verbunden. Eine Variable darf in allen Quelltextfiles insgesamt nur einmal definiert werden.[1)]

Beim Übergang von einer Funktion auf eine andere bleiben die Werte von **extern**-Variablen erhalten, d.h., ihre Lebensdauer entspricht der Ausführungszeit des Programms. Somit stellen solche Variablen eine weitere Möglichkeit der Kommunikation zwischen Funktionen dar. Allerdings sollte man **extern**-Variablen nur dann benutzen, wenn es notwendig ist, da sich bei einer sorglosen Verwendung die Übersichtlichkeit des Programms (insbesondere der Datenverbindungen) verringert und gleichzeitig die Fehleranfälligkeit erhöht. Ausserdem wird empfohlen, die zu einem Programm gehörenden **extern**-Deklarationen am Beginn eines Quelltextfiles vorzunehmen.

### 2.6.3. Speicherklasse static

Genauso wie bei Variablen mit der Speicherklasse **extern** erstreckt sich die Lebensdauer von **static**-Variablen über die gesamte Zeit der Programmausführung, d.h., sie behalten ihre Werte beim Übergang von einer Funktion zur nächsten. Andererseits bieten **static**-Variablen Möglichkeiten eines differenzierten Zugriffsschutzes.

---

1) Laut /50/ ist die mehrfache Definition externer Variabler möglich, wenn damit keine Initialisierung verbunden ist.

```
static int x;

void f()
{

}
...
static int y;

void g()
{

}
```

Ausserhalb von Funktionen definierte **static**-Variablen sind bis zum Ende des Quellfiles sichtbar. Im Beispiel kann auf x sowohl in der Funktion f als auch in g zugegriffen werden, während y nur in der Funktion g bekannt ist. Im Unterschied zu **extern**-Variablen besteht jedoch keine Möglichkeit des Zugriffs aus anderen Quelltextfiles, die zu diesem Programm gehören.

Erfolgt die Definition einer **static**-Variablen in einem Block, so ist sie nur dort sichtbar. Ausserhalb dieses Blocks kann auf sie nicht zugegriffen werden.

```
void f()
{
    static int x;
    ...
    {
        static int y;
        ...
    }

}

void g()
{

}
```

Hinsichtlich ihrer Sichtbarkeit verhalten sich x und y genauso wie **auto**-Variablen. Andererseits behalten **static**-Variablen ihre Werte auch nach dem Verlassen des Gültigkeitsbereichs.

### 2.6.4. Speicherklasse register

Die Speicherklasse **register** bietet dem Programmierer die Möglichkeit, auf die Verbesserung der Kodeeffizienz Einfluss zu nehmen. Eine innerhalb von Funktionen oder geschachtelten Blöcken definierte **register**-Variable bringt der Compiler in einem Maschinenregister unter. Hinsichtlich Gültigkeit und Lebensdauer besteht kein Unterschied zu **auto**-Variablen.

```
void f()
{
    register int x;
    ...
        {
        register int y;
        ...
        }
}
```

Es kann auch gefordert werden, dass formale Parameter in Registern gehalten werden.

```
f(x)
   register int x;
{
...
}
```

Die Anwendung der Speicherklasse **register** unterliegt einigen Einschränkungen. In Maschinenregistern können nur Variablen bestimmter Typen gespeichert werden. Im allgemeinen gehören dazu die Typen **char**, **short**, **int** sowie Zeiger auf beliebige Datenobjekte.

```
void f()
{
    register short x;
    register char c;
    register int *pi;
    register struct datum *pd;
    ...
}
```

Ob **long**-Variablen in Registern gehalten werden können, hängt von der jeweiligen Rechnerarchitektur ab. Es ist nicht möglich, über das Speicherklassenattribut **register** zu erreichen, dass **float**- oder **double**-Variablen in Gleitkommaregistern untergebracht werden.

Man muss sich auch im klaren darüber sein, dass **register**-Variablen nicht adressierbar sind und somit der Operator & nicht auf sie angewendet werden kann.

Die Anzahl der zur Verfügung stehenden Register ist beschränkt. Sie hängt vom Maschinentyp und der Compilerimplementation ab. Bei überzähligen Anforderungen wird (genauso wie bei nicht erlaubten Typangaben) das Schlüsselwort **register** einfach ignoriert und implizit die Speicherklasse **auto** angenommen.

**Aufgabe** 2.28. Untersuchen Sie bisherige Beispielfunktionen nach Möglichkeiten, **register**-Variablen zu verwenden.

### 2.6.5. Speicherklasse von Funktionen

Funktionen besitzen implizit die Speicherklasse **extern**, da eine Verschachtelung von Funktionsdefinitionen nicht erlaubt ist. Wie bei externen Variablen erstreckt sich der Gültigkeitsbereich zunächst bis zum Ende des Quelltextfiles.

file1.c:

```
void f()
{

}

double g(a, b, c)
{

}
```

file2.c:

```
void h()
{

}

void k()
{

}
```

In diesem Beispiel liefert die Funktion g einen Wert vom Typ **double**. Damit g in der Funktion f aufgerufen werden kann, muss dort die Deklaration von g erfolgen.

```
file1.c:

void f()
{
    extern double g();
    ...
}

double g(a, b, c)
{

}
```

Im Gegensatz zu externen Variablendeklarationen ist hierbei das Schlüsselwort **extern** nicht notwendig.

Der Aufruf von g in den Funktionen h und k erfordert ebenfalls, dass g in diesen Funktionen deklariert ist. Die externe Deklaration könnte in dem Fall für beide Funktionen gemeinsam vorgenommen werden.

```
file2.c:

extern double g();

void h()
{

}

void k()
{

}
```

In bestimmten Situationen ist es sinnvoll, die Sichbarkeit einer Funktion einzuschränken. Das geschieht, indem der Funktion bei ihrer Definition das Speicherklassenattribut **static** zugeordnet wird:

```
file1.c:                          file2.c:

void f()                          void h()
{                                 {
    extern double g();
    ...                           }
}

static double g(a, b, c)          void k()
{                                 {

}                                 }
```

Die Funktion g ist nur im File file1.c bekannt. Es besteht keine Möglichkeit des Aufrufs in den Funktionen h oder k. Zu beachten ist, dass bei der Deklaration von g in f das Speicherklassenattribut **static** nicht angegeben werden darf.

### 2.6.6. Blockstruktur und Gültigkeit

Als letztes soll der Zusammenhang zwischen der Blockstruktur und dem Gültigkeitsbereich von Bezeichnern diskutiert werden. Hinsichtlich der Bildung von Namen für Bezeichner existieren drei voneinander unabhängige Klassen. Zur ersten gehören die Namen von Variablen, Parametern, Funktionen und **enum**-Elementen sowie Typbezeichner, die durch **typedef** spezifiziert wurden. In der zweiten Klasse sind die Bezeichner für Struktur-, Union- und Aufzählungstypen zusammengefasst. Die Bezeichner von Struktur- und Unionkomponenten schliesslich bilden die dritte Klasse. Namenskonflikte zwischen Bezeichnern dieser Klassen können somit nicht auftreten.

Allgemein gilt, dass sich die Gültigkeit eines Bezeichners über alle geschachtelten Blöcke erstreckt, wenn nicht in einem dieser Blöcke der gleiche Bezeichner neu vereinbart wird.

```
f()
{                           /* Beginn Funktionsblock */
    int x;
    ...
    {                               /* Beginn Block 1 */
        float x;
        ...
    }                               /* Ende Block 1 */
    ...
}                           /* Ende Funktionsblock */
```

Die **int**-Variable x ist in der gesamten Funktion bekannt, ausgenommen Block 1, in dem x als Variable vom Typ **float** neu definiert wird. Diese Regel gilt unabhängig von den Speicherklassen- und Typangaben für x. Eine Ausnahme besteht dann, wenn in einem geschachtelten Block der Zugriff auf eine extern definierte Variable ermöglicht werden soll.

```
int x;

f()
{
    int x;
    ...
    {
        extern int x;
        ...
    }
    ...
}
```

Die ausserhalb von f definierte Variable x steht in keiner Beziehung zu der im Funktionsblock definierten Variablen x gleichen Typs. Die **extern**-Deklaration hebt die Gültigkeit der **auto**-Variablen x in dem geschachtelten Block auf und stellt eine Verbindung zu der extern definierten Variablen x her.

Eine unangenehme Situation kann auftreten, wenn durch **typedef** definierte Typbezeichner neu vereinbart werden.

```
typedef int index;

f()
{
    register index;             /* Fehler ! */
    ...
}
```

Die Absicht ist klar erkennbar, es soll in der Funktion f der Bezeichner index als eine **register**-Variable vom Typ **int** definiert werden. Da jedoch die Typdefinition noch wirksam ist, interpretiert der Compiler diese Vereinbarung wie

```
register int;
```

In dieser Situation wird die Freizügigkeit zum Problem, das nur durch die explizite Typangabe bei der Definition vermieden werden kann:

```
typedef int index;

f()
{
    register int index;        /* in Ordnung ! */
    ...
}
```

## 2.7. Initialisierung

Unter Initialisierung eines Datenobjekts versteht man die implizite oder explizite Zuweisung eines Anfangswertes bei der Definition. Allgemein gilt, dass Daten der Speicherklassen **extern** und **static** implizit mit Null initialisiert werden. Dagegen besitzen Variablen mit den Speicherklassenattributen **auto** oder **register** undefinierte Werte, falls keine explizite Initialisierung erfolgt. Die explizite Initialisierung von Unions ist grundsätzlich nicht erlaubt. Das gleiche gilt für Felder und Strukturen der Speicherklasse **auto**. Abhängig vom Speicherklassenattribut wird die Initialisierung auf unterschiedliche Weise vorgenommen. Bei **extern** und **static** erhalten die Variablen ihre Anfangswerte während der Compilierung. Dagegen werden **auto**- und **register**-Variablen zur Laufzeit initialisiert. Das geschieht bei jedem Eintritt in den Block, in dem die Variablen definiert sind. Der Eintritt muss dabei über den Blockanfang erfolgen, was dann nicht der Fall ist, wenn mittels **goto** aus irgendeinem anderen Block mitten in diesen Block gesprungen wird. Anders ausgedrückt besteht der Unterschied darin, dass der Compiler für die Initialisierung von **auto**- und **register**-Variablen Kode erzeugen muss, während die Anfangswerte von Variablen der Speicherklassen **extern** und **static** bereits vor der Programmausführung festgelegt sind.

### 2.7.1. Initialisierung von extern- und static-Variablen

Variablen mit arithmetischen Typen und Zeiger können explizit durch einen Konstantenausdruck initialisiert werden, wie folgende Beispiele verdeutlichen:

```
int x = 1;
char space = ' ', tab = '\t';
static int anz = Ø;
#define MAX 512
int i = MAX / 2;
double pi = 3.14159;
```

Die Definition von pi zeigt, dass als Anfangswerte auch Realkonstanten verwendet werden dürfen.

Bei Zeigern sind Adressbezugnahmen auf bereits definierte Datenobjekte möglich.

```
char buf[MAX];
...
char *first = buf;          /* identisch zu ... = &buf[Ø]; */
char *last = &buf[MAX-1];
```

Ausserdem kann ein solcher Zeigerwert mit einer Integer- oder Zeichenkonstanten additiv verknüpft werden.

```
char *bufp = buf + 4;
```

Einen Spezialfall stellt der Zeigerwert NULL dar:

```
char *bufp1 = NULL;
```

Die Definition der Syntax erlaubt es, solche Initialisierungsausdrücke in geschweiften Klammern anzugeben, z.B. ist

```
static int j = {MAX - 1};
```

identisch zu

```
static int j = MAX - 1;
```

Die Initialisierung von Feldern und Strukturen erfolgt nach dem gleichen Prinzip. Dabei werden die Konstantenausdrücke zur Initialisierung der einzelnen Struktur- oder Feldelemente untereinander durch Komma getrennt. Die gesamte Liste der Anfangswerte muss in geschweifte Klammern eingeschlossen werden.

```
int feld1[5] = {Ø, 1, 2, 3, 4};
```

Möchte man wie im obigen Beispiel allen Feldelementen Anfangswerte zuweisen, so kann man die Angabe der Feldgrösse weglassen.

```
int feld1[] = {Ø, 1, 2, 3, 4};
```

Die folgenden beiden Felder besitzen unterschiedlich viele Feldelemente:

```
int felda[5] = {Ø, 1};
int feldb[] = {Ø, 1};
```

Während zu felda 5 Elemente gehören, berechnet der Compiler die Feldgrösse von feldb aus der Anzahl der angegebenen Anfangswerte. Somit besteht feldb aus den Elementen feldb[Ø] und feldb[1]. Im übrigen erhalten die Elemente felda[2], felda[3] und felda[4] implizit den Wert Ø.

Die Zuordnung der Anfangswerte beginnt grundsätzlich beim ersten Element. Es existert auch keine Möglichkeit der Angabe eines Wiederholungsfaktors. Ausserdem darf man nur so viele Anfangswerte angeben, wie Feldelemente vorhanden sind.

Zeichenfelder können durch Zeichenkettenkonstanten initialisiert werden. Anstelle der aufwendigen Notation

```
char z[] = {'P','r','o','g','r','a','m','m','i','e','r','e','n',
            ' ','m','i','t',' ','C','\Ø'};
```

verwendet man die günstigere Form

```
char z[] = "Programmieren mit C";
```

In beiden Fällen besteht z aus 20 Zeichen, da eine Zeichenkettenkonstante immer mit \Ø abgeschlossen wird.

Das nächste Beispiel zeigt die Initialisierung eines Feldes, dessen Elemente Zeiger auf Zeichenketten darstellen.

```
static char *keyword[] = {"if",
                          "else",
                          ...,
                          "while"};
```

Es erfolgt dabei nicht die Zuordnung der Zeichenkettenkonstanten, sondern es wird jedem Element von keyword die Adresse des Beginns einer Zeichenkette zugewiesen. Wo die Zeichenkettenkonstanten selbst gespeichert sind, ist uninteressant.

Strukturen werden in der gleichen Weise wie Felder initialisiert.

```
struct datum {
    short jahr;
    char monat[4];
    short tag;
} heute = {1986, "Okt", 31},
  geb_datum = {1952, "Mai", 13};
```

An Unions kann man jedoch bei der Definition keine Anfangswerte zuweisen.

Welche syntaktischen Besonderheiten sind nun zu beachten, wenn Strukturen oder Felder initialisiert werden sollen, die als Elemente selbst strukturierte Datentypen enthalten? Betrachten wir dazu ein zweidimensionales Feld von **int**-Grössen:

```
static int a[2][3] = {
                          {5, 5, 5},
                          {4, 4, 4}
                     };
```

In C wird ein zweidimensionales Feld a[2][3] als ein Feld mit den Elementen a[Ø] und a[1] verstanden, wobei diese Elemente selbst Felder mit je 3 Elementen darstellen. Daraus resultiert die Anordnung der geschweiften Klammern. Durch

```
{5, 5, 5}
```

erfolgt die Wertebelegung der Elemente von a[Ø], d.h. a[Ø][Ø], a[Ø][1] bzw. a[Ø][2], und durch

```
{4 ,4 ,4}
```

werden die Elemente a[1][Ø], a[1][1] und a[1][2] von a[1] initialisiert.

Wenn beispielsweise nur den Elementen a[Ø][Ø] und a[1][Ø] Anfangswerte ungleich Ø zugewiesen werden sollen, so könnte das analog zu eindimensionalen Feldern folgendermassen geschehen:

```
static int b[2][3] = {
                          {5},
                          {4}
                     };
```

Die Tatsache, dass die Elemente zweidimensionaler Felder zeilenweise gespeichert werden, kann ausgenutzt werden, um geschweifte Klammern einzusparen.

```
int c[2][3] = {5, 5, 5, 4, 4, 4};
int d[][3]  = {5, 5, 5, 4, 4, 4};
int e[3][3] = {5, 5, 5, 4, 4, 4};
```

Die Felder c und d haben jeweils 6 Elemente und die gleiche Anfangswertebelegung. Dagegen besteht e aus insgesamt 9 Elementen, wobei e[2][Ø], e[2][1] und e[2][2] implizit mit Ø initialisiert werden.

### 2.7.2. Initialisierung von auto- und register-Variablen

Werden Variablen der Speicherklassen **auto** oder **register** explizit initialisiert, so erfolgt die Berechnung und Zuordnung der Werte zur Laufzeit. Demzufolge sind als Anfangswerte beliebige Ausdrücke zulässig, wenn sie sich auf bereits definierte Datenobjekte beziehen.

```
float pi = 3.14159;
int i = Ø, c = f(i);
register int pc = &c;
```

Die zweite Definition zeigt, dass sich der Anfangswert von c aus dem Funktionswert von f(Ø) ergibt, vorausgesetzt f ist definiert.

Im Prinzip stellt die explizite Initialisierung von automatischen und Registervariablen nichts weiter dar als eine Kurzform von Definition und Wertzuweisung:

```
int i, c;

i = Ø;
c = f(i);
```

## 2.8. Typkonvertierung

Eine automatische Typanpassung erfolgt, wenn in Ausdrücken Teilausdrücke unterschiedlichen Typs auftreten. Die folgenden Abschnitte erläutern die Regeln, nach denen derartige Umwandlungen vorgenommen werden. Darüber hinaus kann man eine Typkonvertierung explizit anfordern.

### 2.8.1. Konvertierungen zwischen den Grundtypen

In Ausdrücken mit binären arithmetischen Operatoren finden u.U. folgende Konvertierungen statt:

- Zunächst werden Operanden der Typen **char** und **short** in den Typ **int**, Operanden des Typs **unsigned char** oder **unsigned short** in den Typ **unsigned int** und Operanden des Typs **float** in den Typ **double** konvertiert.
- Danach wird eine weitere Umwandlung genau dann vorgenommen, wenn beide Operanden einen unterschiedlichen Typ besitzen. Die Konvertierung erfolgt als eine Angleichung an den höheren Typ. Die Reihenfolge in diesem Sinne ist **double**, **unsigned long**, **long** sowie **unsigned int**. Besitzt also

beispielsweise einer der Operanden den Typ **double**, so wird auch der andere Operand in diesen Typ transformiert. Die Ausdrucksberechnung ergibt wiederum einen **double**-Wert. Abweichend von diesen Regeln erfolgt laut /50/ die Konvertierung, wenn der eine Operand den Typ **long** und der andere Operand den Typ **unsigned int** besitzt. In diesem Fall erfolgt die Konvertierung beider Operandentypen in den Typ **unsigned long**.

Ob bei der Konvertierung von **char**-Werten in irgendeinen Integertyp eine Vorzeichenausdehnung erfolgt (und damit negative Werte entstehen), ist implementationsabhängig. Garantiert ist lediglich, dass Werte aus dem Standardzeichensatz in nichtnegative Integerwerte umgewandelt werden. Bei Rechnertypen mit Vorzeichenausdehnung nehmen Variablen des Typs **char** numerische Werte im Bereich von -127 bis 128 an, während Variablen des Typs **unsigned char** einen Wertebereich von Ø bis 255 besitzen. Die Konvertierung von **char**, **short**, **int** bzw. **long** in den Typ **unsigned int** bzw. **unsigned long** erfolgt, indem die Bitfolge des Ausgangswertes als vorzeichenloser Integerwert interpretiert wird.

Die in arithmetischen Ausdrücken durchgeführten Typkonvertierungen sind im Bild 2.4 zusammengefasst. Die Konvertierung erfolgt immer in Richtung des Pfeils. Die durchgezogenen Pfeile deuten an, dass diese Typumwandlung grundsätzlich vorgenommen wird.

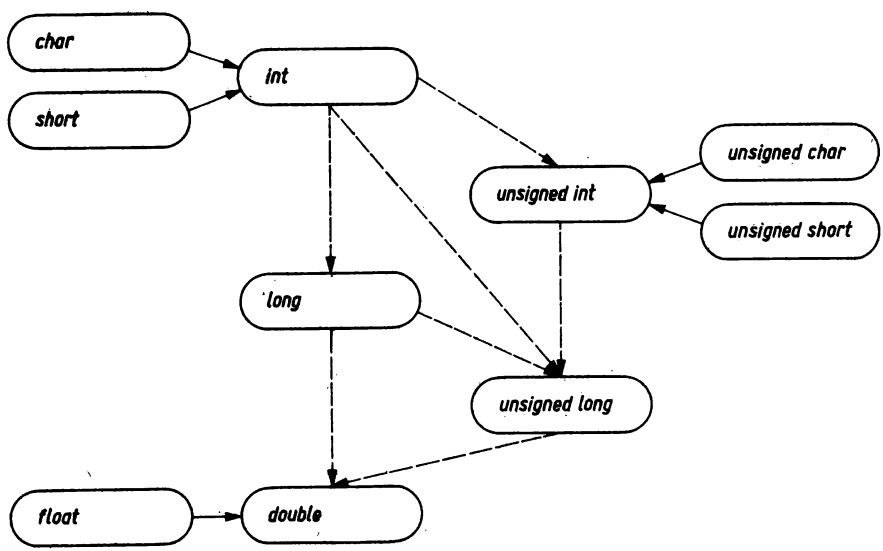



Bild 2.4. Typkonvertierung in arithmetischen Ausdrücken

Unter Umständen ist in Wertzuweisungen der Form

<lvalue> = <ausdruck>

bzw.

<lvalue> <op>= <ausdruck>

eine Konvertierung des Wertes von <ausdruck> in den Typ von <lvalue> erforderlich. Im weiteren wird nur auf die Problemfälle Bezug genommen:

- Besitzt <lvalue> den Typ **float** und der Ausdruck auf der rechten Seite den Typ **double**, so erfolgt eine Rundung und nötigenfalls das Abschneiden der Mantisse. Das Ergebnis der Konvertierung ist undefiniert, wenn der entstehende Wert nicht als **float**-Grösse dargestellt werden kann.
- Wird eine Konvertierung von **float** oder **double** in einen Integer- oder den Zeichentyp erforderlich, so können undefinierte Werte entstehen. Zumindest erfolgt das Abschneiden des gebrochenen Anteils, wobei die Behandlung negativer Werte maschinenabhängig ist.
- Konvertierungen von einem "höheren" in einen "niederen" Integertyp (z.B. von **long** in **int**) sowie Umwandlungen vom Integer- in den Zeichentyp erfolgen durch Abschneiden der höherstelligen Bits.

### 2.8.2. Zeiger- und Integerwerte

Erlaubt sind die Addition eines Zeiger- und Integerwertes sowie die Subtraktion einer Integergrösse von einem Zeigerwert. Diese Operationen erfolgen in Speichereinheiten des Typs, auf den der Zeiger vereinbart ist. Sie sind nur sinnvoll, wenn dabei auf Elemente desselben Feldes Bezug genommen wird. Der Integerwert wird mit der Länge des Typs multipliziert und das Resultat zum bzw. vom Zeigerwert addiert bzw. subtrahiert. Das Ergebnis einer solchen Verknüpfung ist wiederum ein Zeigerwert des betreffenden Typs. Praktisch bedeutet die Addition eines Integerwertes i und eines Zeigers die Verschiebung um i Elemente in einem Feld nach rechts. Die Subtraktion von i stellt dementsprechend eine Verschiebung um i Elemente nach links dar.

Die Zuweisung eines Integerwertes an eine Zeigervariable erfolgt ebenso wie die Zuordnung eines Zeigerwertes an eine Variable vom Typ **int** ohne Umwandlung, d.h. als einfache Kopie des entsprechenden Wertes. Bis auf die Zuweisung des Integerwertes NULL sind derartige Operationen nicht portabel.

### 2.8.3. Konvertierung von Funktionsargumenten und -werten

Analog zur Konvertierung in arithmetischen Ausdrücken werden Argumente des Typs **float** vor der Übergabe grundsätzlich in den Typ **double** konvertiert, ebenso **char**- bzw. **short**-Typen in den Typ **int**. Weiterhin erfolgt die Umwandlung eines Feldbezeichners in einen Zeiger auf das erste Feldelement. Da durch den Compiler keine Prüfung auf Übereinstimmung von Argument- und Parametertyp stattfindet, werden auch keinerlei weitere Typkonvertierungen vorgenommen. Eine eventuell notwendige Typangleichung muss explizit erzwungen werden (s. Abschn. 2.8.4.).

Gegebenenfalls wird vor der Rückgabe des Funktionswertes durch eine Anweisung

```
return <ausdruck>;
```

der Typ des Wertes von <ausdruck> in den Typ der Funktion umgewandelt.

### 2.8.4. Explizite Typkonvertierung

Über eine sog. **cast**-Konstruktion kann man die Konvertierung des Typs eines Ausdrucks explizit erzwingen. Die allgemeine Form lautet:

```
(<typ_spezifikation>) <ausdruck>
```

Dabei gibt <typ_spezifikation> den Zieltyp an. Die Syntax entspricht hierbei der Deklaration eines Datenobjektes ohne Angabe eines Bezeichners. Um das zu verdeutlichen, betrachten wir zunächst einige Deklarationen:

```
int a;            /* int */
int *b;           /* Zeiger auf int */
struct datum *c; /* Zeiger auf Strukturtyp datum */
float *d[4];      /* Feld von 4 Zeigern auf float-Werte */
float (*e)[4];    /* Zeiger auf ein Feld von 4 float-Elementen */
int *f();         /* Funktion, die einen Zeiger auf int liefert */
int (*g)();       /* Zeiger auf Funktion, die int-Wert liefert */
int (*h[4])();    /* Feld von 4 Zeigern auf Funktionen mit Wert int */
```

Hinsichtlich des Typs korrespondieren dazu die folgenden Angaben für <typ_spezifikation>:

```
int
int *
struct datum *
float * [4]
float (*) [4]
int * ()
int (*) ()
int (*[4]) ()
```

Die Wirkung einer cast-Konstruktion entspricht praktisch einer Wertzuweisung der Form

```
<lvalue> = <ausdruck>
```

Dabei bezeichnet <lvalue> ein anonymes Datenobjekt mit dem Datentyp <typ_spezifikation>.

Anwendung findet die explizite Typkonvertierung z.B., wenn bei einem Funktionsaufruf die Angleichung des Argumenttyps an den Parametertyp erforderlich ist. Erinnern wir uns an die binäre Suchfunktion bsearch aus Abschn. 2.5.5. Soll diese Funktion auf ein aufsteigend sortiertes Feld a mit n **int**-Elementen angewendet werden, so kann bsearch folgendermassen aufgerufen werden:

```
char *bsearch();
int a[n], key, *pi, icomp();
...
pi = (int *)bsearch((char *)key, (char *)a, n, sizeof(int), icomp);
...
```

Der Vollständigkeit halber zeigen wir noch die Vergleichsfunktion icomp.

```
int icomp(a, b)     /* Vergleich zweier int-Werte *a und *b */
    char *a, *b;
{
    return(*(int *)a - *(int *)b)
}
```

Eine Konvertierung in den Typ **void** ist möglich und wird z.B. angewendet, um explizit anzuzeigen, dass ein von einer Funktion zurückgegebener Wert nicht weiter benutzt wird.

```
...
char s1[n], s2[m];
...
(void) strcpy(s1, s2);
```

Andererseits kann der Typ **void** natürlich nicht in einen anderen Typ umgewandelt werden. Schliesslich muss darauf hingewiesen werden, dass eine explizite Konvertierung von oder in Struktur-, bzw. Uniontypen nicht zulässig ist.

Bild 2.5 gibt eine Übersicht über die möglichen Typen der Operanden bei Anwendung der einzelnen Operatoren sowie die möglichen Typen des Ergebnisses der Operation.

| Operation | Operandentypen | Ergebnistypen |
|---|---|---|
| x + y | arithmetisch | **int,unsigned long,double** |
| | Zeiger auf Feldelement und Operand mit integralem Typ | Zeiger auf Feldelement |
| x - y | arithmetisch | **int,unsigned, long,double** |
| | Zeiger auf Feldelement und Operand mit integralem Typ | Zeiger auf Feldelement |
| | Zeiger auf Elemente des gleichen Feldes | **int** |
| x * y | arithmetisch | **int,unsigned long,double** |
| x / y | arithmetisch | **int,unsigned long,double** |
| x % y | integraler Typ | **int,unsigned long** |
| -x | arithmetisch | **int,unsigned long,double** |
| x++ (++x)<br>x-- (--x) | arithmetisch (Lvalue!) | **int,unsigned long,double** |
| | Zeiger auf Feldelement | Zeiger auf Feldelement |
| x = y<br>x <op>= y | arithmetisch (x Lvalue!) | arithmetisch entsprechend x |
| | gleiche Struktur- bzw. Uniontypen | Typ der Struktur bzw. Union |
| | Zeiger auf Objekte gleichen Typs | Zeigertyp |
| x & y | integraler Typ | **int,unsigned long** |
| x \| y | integraler Typ | **int,unsigned long** |
| x ^ y | integraler Typ | **int,unsigned long** |
| ~x | integraler Typ | **int,unsigned long** |
| x << n<br>x >> n | integraler Typ | Typ von x |

| Operation | Operandentypen | Ergebnistypen |
|---|---|---|
| x && y | arithmetisch bzw. Zeigertyp | **int** (Ø oder 1) |
| x \|\| y | arithmetisch bzw. Zeigertyp | **int** (Ø oder 1) |
| !x | arithmetisch bzw. Zeigertyp | **int** (Ø oder 1) |
| x < y<br>x > y<br>x <= y<br>x >= y | arithmetisch bzw. Zeiger (auf Elemente des gleichen Feldes) | **int** (Ø oder 1) |
| x == y<br>x != y | arithmetisch bzw. Zeiger (auf Elemente des gleichen Feldes) bzw. Zeigertyp und Integerwert NULL | **int** (Ø oder 1) |
| x ? y : z | arithmetisch | **int,unsigned long,double** |
| | y und z vom gleichen Struktur-, Union- oder Zeigertyp | Struktur-, Union- oder Zeigertyp |
| *x | Zeigertyp <typ> | <typ> |
| &x | beliebiger Typ <typ> ausser **void** (Lvalue!) | Zeiger auf <typ> |
| x[n] | x: Zeiger auf beliebigen Typ <typ> ausser **void** (x ist normalerweise ein Feldbezeichner)<br>n: **int** | <typ> |
| x.y | x: Struktur- oder Uniontyp<br>y: beliebiger Typ <typ> ausser **void** (Komponente von x!) | <typ> |
| x->y | x: Zeiger auf Struktur oder Union<br>y: beliebiger Typ <typ> ausser **void** (Komponente von *x!) | <typ> |
| (<typ_sp>) x | x: beliebiger Typ ausser **void** sowie Struktur- und Uniontypen<br><typ_sp> entspricht <typ_spezifikation> | <typ> |
| **sizeof** x<br>**sizeof**(<typ_sp>) | x: beliebiger Typ<br><typ_sp> entspricht <typ_spezifikation> | **unsigned** |
| x,y | beliebige Typen | Typ von y |

Bild 2.5. Typen der Operanden und Ergebnisse

## 2.9. Programmbeispiele

Zum Abschluss von Teil 2 stellen wir einige Beispiele zu etwas komplexeren Fragestellungen vor, um verschiedene Aspekte von C zu verdeutlichen, insbesondere die Arbeit mit Zeigern.

### 2.9.1. Textformatierung

Es soll ein Programm geschrieben werden, das Text einliest, formatiert und wieder ausgibt, so dass jede Textzeile (unabhängig von der ursprünglichen Zeilenstruktur) eine vorgegebene Zeilenlänge nicht überschreitet. Die Trennung der Zeilen erfolgt grundsätzlich an einer Wortgrenze, wobei als "Wort" eine Folge von Zeichen ungleich Leerzeichen, Tabulator und Newline betrachtet wird /31/. Der Einfachheit halber werden Tabulator und Newline in Leerzeichen umgewandelt. Die wesentliche Leistung dieses einfachen Textformatierers besteht dabei darin, dass die Ausrichtung des Textes auf den rechten Rand einer Zeile erfolgt (in der gleichen Weise, wie der Text eines Absatzes dieses Buches).

Um das Problem überschaubar zu halten, nehmen wir an, dass es sich bei dem Eingabefile um "normalen", fortlaufenden Text ohne spezielle Besonderheiten, wie Steuerzeichen, Tabellen usw., handelt. Obwohl es sicher sinnvoll wäre, bestimmte Möglichkeiten der Drucksteuerung und Formatierung zu besitzen (z.B. zur Gestaltung von Überschriften und Absätzen oder zum Einfügen von speziell aufbereiteten Beispielen), soll hier auf diese Dinge verzichtet werden.

Zum besseren Verständnis abstrahieren wir zunächst von den Details und beschreiben den Grundalgorithmus in der folgenden Form:

```
while(in der Eingabe sind noch Zeichen vorhanden) {
   Einlesen in den Puffer ab aktuellem Pufferzeiger;
   while(Zeichenanzahl ist grösser als Zeilenlänge) {
      if(Zeile liegt bereits formatiert vor)
         i.O.;
      else
         formatiere eine Zeile;
      gib diese Zeile aus;
      verschiebe restlichen Text an den Pufferanfang;
      berechne Anzahl der Zeichen im Puffer;
   }
   setze Pufferzeiger hinter letztes Zeichen im Puffer;
}
gib Resttext im Puffer aus;
```

Hieraus lässt sich unmittelbar die **main**-Funktion ableiten, wobei die einzelnen Detailprobleme immer noch in speziellen Funktionen versteckt bleiben.

Wir gehen davon aus, dass alle zum Programm gehörenden Funktionen in einem Quelltextfile untergebracht sind.

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

#define   MAX  512
#define   LAENGE 8Ø

char inbuf[MAX + LAENGE+1];

main()              /* einfacher Textformatierer */
{
    register int n;
    register char *bufp = inbuf;
    register char *pright;
    extern char *find();

    while(n = readbuf(bufp, inbuf + MAX - bufp)) {
        while(n > LAENGE) {
            pright = inbuf + LAENGE - 1;
            if(*pright != ' ' && *(pright + 1) == ' ')
                ;
            else
                format(find(pright), pright);
            putline(inbuf, LAENGE);
            while(*++pright == ' ')
                ;
            strcpy(inbuf, pright);
            n = strlen(inbuf);
        }
        bufp = inbuf + n;
    }
    putline(inbuf, bufp - inbuf);
}
```

Das File string.h enthält die Deklarationen für die Bibliotheksfunktionen für Zeichenkettenverarbeitung.

Die Formatierung einer Zeile erfolgt in zwei Schritten. Die Funktion find sucht ausgehend von der am weitesten rechts befindlichen Position (pright) das Ende des letzten Wortes (plz).

```
char *find(pright) /* Suche Ende des letzten Wortes */
    char *pright;
{
    extern char inbuf[];
    register char *plz = pright;

    while(*plz != ' ' && plz > inbuf)
        --plz;
    if(plz == inbuf)
        return(pright);
    while(*--plz == ' ' && plz > inbuf)
        ;
    return(plz);
}
```

Die Funktion format realisiert die Ausrichtung des Textes einer Zeile auf den rechten Rand durch Einfügen von Leerzeichen.

```
void format(plz, pright)                    /* Formatieren einer Zeile */
    char *plz;                              /* Ende des letzten Wortes */
    char *pright;                                  /* Ende der Zeile */
{
    register char *pakt = plz;  /* Adresse des aktuellen Zeichens */
    register int i;     /* Anzahl bereits eingefügter Leerzeichen */
    int lzanz = pright - plz; /* Anzahl einzufügender Leerzeichen */
    char bufsave[MAX];                          /* Zwischenspeicher */
    extern char inbuf[];                          /* aktuelle Zeile */

    for(i = Ø; i < lzanz; i++) {
        while(*--pakt != ' ')                                    /*1*/
            if(pakt == inbuf)                                    /*2*/
                pakt = pright - lzanz + i;                       /*3*/
        strcpy(bufsave, pakt + 1);                               /*4*/
        *(pakt + 1) = ' ';                                       /*5*/
        strcpy(pakt + 2, bufsave);                               /*6*/
        while(*--pakt == ' ')                                    /*7*/
            ;
    }
}
```

Die Differenz pright-plz ergibt die Anzahl der Leerzeichen (lzanz), die gleich verteilt zwischen die einzelnen Worte der Zeile einzufügen sind. Bei jedem Durchlauf der **for**-Schleife wird ein Leerzeichen zwischen zwei Worte eingefügt und damit der Rest der Zeile um eine Stelle nach rechts verschoben. Die Formatierung einer Zeile erfolgt von rechts nach links, d.h., das erste Leerzeichen wird vor das letzte Wort gespeichert, das zweite Leerzeichen vor das vorletzte Wort usw. Im Detail geschieht das folgendermassen: Zunächst erfolgt die Suche des vor einem Wort befindlichen Leerzeichens (/*1*/). Dann werden alle rechts davon stehenden Zeichen in bufsave zwischengespeichert (/*4*/), ein Leerzeichen eingefügt (/*5*/) und der Zeilenrest aus bufsave dahinter gespeichert (/*6*/). Nach dem Übergehen eventueller Leerzeichen zeigt pakt auf das Ende des davorstehenden Wortes (/*7*/). Ist die Formatierung beim Erreichen des Zeilenanfangs noch nicht abgeschlossen (/*2*/), so wird mit dem Einfügen von Leerzeichen am Ende der Zeile (/*3*/) fortgesetzt.

Schliesslich benötigen wir noch die Funktionen readbuf zum Einlesen in den Puffer ab einer vorgegebenen Position sowie putline zur Ausgabe einer aufbereiteten Zeile .

```
void readbuf(linep, max)                    /* Füllen des Eingabepuffers */
    register int max;
    char *linep;
{
    register int c;
    register char *pz;
    static int eof = Ø;

    if(eof)
        return(Ø);
    for(pz = linep; --max > Ø && (c = getchar()) != EOF; pz++)
        if((*pz = c) == '\n' || *pz == '\t')
            *pz = ' ';
    if(c == EOF) {
        eof = 1;
        if(pz == linep)
            return(Ø);
    }
    *pz = '\Ø';
    return(pz - linep);
}
```

```
void putline(linep, anz)      /* Ausgabe einer Zeile */
    register int anz;
    register char *linep;
{
    while(--anz >= Ø)
        putchar(*linep++);
    putchar('\n');
}
```

**Aufgabe 2.29.** Entwickeln Sie ein Programm, das neben der Textausrichtung auf den rechten Rand weitere Formatierungsmöglichkeiten bietet, z.B. die Bildung von Absätzen und das zeitweise Unterdrücken der Formatierung.

### 2.9.2. Sortieren

Die Aufgabenstellung besteht zunächst darin, eine Funktion zu schreiben, die ein Feld k mit n **int**-Elementen in aufsteigender Folge sortiert. Das erste Verfahren, das wir vorstellen, geht von der Annahme aus, dass bereits i-1 Elemente des Feldes sortiert vorliegen:

```
k[Ø] <= k[1]  <= ... <= k[i-1]
```

Nun wird das nächste Element k[i] in absteigender Folge so lange mit seinen Vorgängern verglichen und ausgetauscht, bis wieder eine Sortierfolge

```
k[Ø] <= k[1] <= ... <= k[i-1] <= k[i]
```

entstanden ist.

```
void ssort(k, n)          /* straight insertion sort (Version 1) */
    int k[], n;
{
    register int j, i, temp;

    for(i = 1; i < n; i++)
        for(j = i-1; j >= Ø && k[j] > k[j+1]; j--) {
            temp = k[j+1];
            k[j+1] = k[j];
            k[j] = temp;
        }
}
```

Die innere **for**-Schleife steuert den Vergleich und den Austausch, während durch das äussere **for** jeweils das nächste Feldelement in den Sortiermechanismus einbezogen wird.

Um den Umgang mit Zeigern zu üben, stellen wir eine Zeigerversion dieser Funktion vor.

```
void ssort(first, last) /* straight insertion sort (Version 2) */
    int *first, *last;
{
    register int *i, *j, temp;

    for(i = first; i <= last; i++)
        for(j = i-1; j >= first && *j > *(j+1); j--) {
            temp = *(j+1);
            *(j+1) = *j;
            *j = temp;
        }
}
```

Im Gegensatz zur vorhergehenden Version wird als zweites Aufrufargument die Adresse des letzten Feldelements gefordert.

Als nächstes wird der Algorithmus **Quicksort** nach C.A.R. Hoare vorgestellt /20/. Der Grundansatz besteht darin, das zu sortierende Feld durch Umordnen einzelner Elemente so in zwei Teilfelder zu zerlegen, dass alle Elemente des einen Teilfeldes kleiner oder gleich sind als ein beliebiges Element des anderen Teilfeldes. Das gleiche Verfahren wird so lange rekursiv auf die Teilfelder angewendet, bis alle Elemente sortiert sind.

Es folgt der Quelltext der Funktion qsort(first, last) zur Sortierung eines Feldes von **int**-Elementen. Beim Aufruf dieser Funktion wird first die Adresse des ersten und last die Adresse des letzten Elements eines Feldes oder Teilfeldes zugewiesen.

```
void qsort(first, last)      /* Quicksort (nur für int-Werte) */
    int *first;
    int *last;
{
    register int *i;
    register int *j;

    if(first >= last)
        return;
    for(i = first, j = last; ; j--) {
        while(i != j && *i <= *j)
            j--;
        if(i == j)
            break;
        austausch(i, j);
        do
            i++;
        while(i != j && *i <= *j)
            ;
        if(i == j)
            break;
        austausch(i, j);
    }
    qsort(first, i-1);
    qsort(i+1, last);
}
```

Das Hauptproblem dieses Sortierverfahrens besteht in dem Zerlegungsalgorithmus. Angenommen, i bzw. j sind zwei Zeiger auf das erste bzw. das letzte Feldelement und die Elemente *i und *j werden miteinander verglichen. Dann wird im Falle *i <= *j der Zeiger j dekrementiert - damit zeigt j auf das vorhergehende Element - und der Vergleich wiederholt. Andernfalls erfolgt der Austausch beider Elemente und die Inkrementierung des Zeigers i, der somit auf das nächste Element zeigt. Nun wird zyklisch *i mit *j verglichen und i inkrementiert, bis wiederum ein Austausch erforderlich ist, d.h. *i grösser als *j ist. Nach dem Vertauschen der Elemente beginnt der ganze Mechanismus von vorn, d.h., j wird dekrementiert, der Vergleich durchgeführt usw. Der Algorithmus bricht ab, falls i und j auf die gleichen Elemente zeigen. In dieser Situation ist die Zerlegung des Feldes abgeschlossen, alle Elemente links von i (oder j) bilden ein Teilfeld, und alle Elemente rechts von i gehören zum anderen Teilfeld.

Es fehlt nur noch die Funktion zum Vertauschen zweier Elemente im Feld:

```
void austausch(i, j)          /* Austausch zweier Elemente */
    int *i, *j;
{
    register int temp;

    temp = *i;
    *i = *j;
    *j = temp;
}
```

Die bisher vorgestellten Sortierfunktionen sind nur in der Lage, Felder mit **int**-Elementen aufsteigend zu sortieren. Es stellt kein Problem dar, diese Funktionen dahingehend zu modifizieren, dass sie Elemente eines anderen Typs behandeln oder die Sortierung nach anderen Kriterien durchführen. Wir wollen jedoch einen Schritt weiter gehen und eine allgemeine Sortierfunktion vorstellen, die auf Felder beliebiger Elementetypen anwendbar ist. Ausserdem kann der Nutzer beim Aufruf dieser Funktion festlegen, nach welchen Kriterien sortiert werden soll.

```
static int (*qs_comp)();
static int qs_length;

void sort(first, n, length, compare)   /* Aufruf von Quicksort */
    char *first;        /* Zeiger auf erstes Element */
    unsigned n;         /* Anzahl der Elemente */
    int length;         /* Elementlänge in Byte */
    int (*compare)();   /* Zeiger auf Vergleichsroutine des Nutzers */
{
    qs_comp = compare;
    qs_length = length;
    qsort(first, first + (n-1) * length);
}
```

Betrachten wir zunächst die Parameter von sort. Beim Aufruf werden vier Argumente gefordert. Das erste Argument ist ein Zeiger auf das erste Element des zu sortierenden Feldes, das zweite gibt die Anzahl der Elemente und das dritte die Länge eines Elements an. Von Interesse ist insbesondere der Parameter compare. Die Parameterdeklaration

```
int (*compare)();
```

legt fest, dass es sich bei dem Parameter compare um einen Zeiger auf eine Funktion handelt (vgl. Abschn. 2.5.5.). Auf diese Weise wird beim Aufruf von sort die Adresse einer Funktion übergeben, die den Vergleich der zu sortierenden Elemente durchführt. Wir gehen gleich etwas genauer auf dieses Problem ein.

Die Funktion sort selbst weist nur den **static**-Variablen qs_length bzw. qs_comp die Werte von length bzw. compare zu und ruft anschliessend die eigentliche Sortierfunktion, eine Modifikation von qsort, auf. Der wesentliche Unterschied zwischen dieser und der ursprünglichen qsort-Version besteht darin, dass der Vergleich zweier Feldelemente in der Funktion nicht "fest verdrahtet" ist. Statt dessen erfolgt der Aufruf der vom Nutzer bereitgestellten Vergleichsfunktion, die zwei durch i und j adressierte Feldelemente miteinander vergleicht:

```
(*qs_comp)(i,j)
```

Nach welchen Kriterien und auf welche Weise der Vergleich in der gerufenen Funktion *qs_comp durchgeführt wird, bleibt nach aussen verborgen und ist

an dieser Stelle auch uninteressant. Festgelegt ist nur, dass diese Funktion einen Ergebniswert nach folgender Konvention liefert: Der Funktionswert ist kleiner als Ø, falls das durch i adressierte Feldelement im Sinne des Sortierkriteriums "kleiner" als das durch j adressierte Element ist. Analoge Relationen gelten für Funktionswerte grösser als Ø bzw. gleich Ø.

```
static void qsort(first, last)     /* Quicksort */
    char *first, *last;
{
    register char *i, *j;

    if(last-first < qs_length)
        return;
    for(i = first, j = last;; j -= qs_length)  {
        while(i != j && (*qs_comp)(i, j) <= Ø)
            j -= qs_length;
        if(i == j)
            break;
        swap(i, j);
        do
            i += qs_length;
        while(i != j && (*qs_comp)(i, j) <= Ø);
        if(i == j)
            break;
        swap(i, j);
    }
       qsort(first, i - qs_length);
       qsort(i + qs_length, last);
}
```

Die Funktion swap ist eine verallgemeinerte Variante von austausch:

```
static void swap(x, y)             /* Austausch zweier Elemente */
    register char *x, *y;
{
    register char temp;
    register int i;

    for(i = Ø; i < qs_length; i++)  {
        temp = *x;
        *x++ = *y;
        *y++ = temp;
    }
}
```

Die Verwendung des Speicherklassenattributes **static** erfolgt unter der Voraussetzung, dass sich die Definitionen von qs_comp und qs_length sowie die Funktionsdefinitionen von sort, qsort und swap in einem Quelltextfile befinden und somit gemeinsam compiliert werden. Die Variablen qs_comp bzw. qs_length sind dann nur innerhalb dieses Quelltextfiles bekannt. Nach aussen sind sie nicht sichtbar, d.h., der Benutzer von sort kann auf diese Variablen weder absichtlich noch unbeabsichtigt zugreifen. Das gleiche gilt für die Funktionen qsort und swap.

**Aufgabe 2.30.** Testen Sie die vorgestellten Sortierfunktionen.

**Aufgabe 2.31.** Ersetzen Sie in der Sortierfunktion sort das Sortierverfahren Quicksort (Funktion qsort) durch eine verallgemeinerte Variante des Verfahrens "straight insertion sort" (Funktion ssort).

### 2.9.3. Durchmustern von Binärbäumen

Ein Binärbaum (binary tree, s. Bild 2.6) ist eine endliche Menge von von Elementen, die leer ist oder aus einem Wurzelelement und zwei disjunkten Teilmengen besteht. Die disjunkten Teilmengen werden linker und rechter Unterbaum genannt und sind selbst wieder Binärbäume. Jedes Element des Baums wird als Knoten (node) bezeichnet. Ein Knoten, dessen linker und rechter Unterbaum leer ist, heisst Blatt (leaf) oder Blattknoten. Die Niveauebene (level) eines Knotens ist wie folgt definiert. Der Wurzelknoten befindet sich auf Niveau Ø. Befindet sich ein Knoten auf Niveau i, so ergibt sich für die Wurzelknoten der zugehörigen Unterbäume die Niveauebene i+1. Bei einem Binärbaum handelt es sich um eine rekursive Datenstruktur.

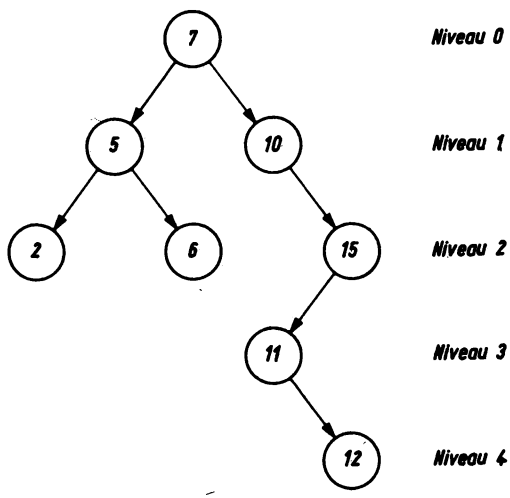



Bild 2.6. Binärbaum

Wie in Bild 2.6 angedeutet, kann man für Binärbäume Ordnungsrelationen definieren. Für die Arbeit mit Binärbäumen, in denen eine Ordnungsrelation definiert ist, existieren in der Standardbibliothek folgende Funktionen:

| | | |
|---|---|---|
| tfind | - | Suchen eines Elements |
| tsearch | - | Suchen und Installieren eines Elements |
| tdelete | - | Entfernen eines Elements |
| twalk | - | Durchmustern eines Baums |

In diesem Abschnitt beschäftigen wir uns mit dem Durchmustern eines Binärbaums. Ausgehend vom Wurzelknoten erfolgt zunächst die Untersuchung des linken und anschliessend die Untersuchung des rechten Unterbaums. Dieser Vorgang wird für jeden Unterbaum wiederholt. Ein Knoten wird beim Durchmustern mehrfach getroffen (visit), und zwar vor dem Abstieg nach links (preorder), vor dem Abstieg nach rechts (postorder) und nach dem Abstieg

nach rechts (endorder). Beispielsweise entstehen beim Durchmustern des Binärbaums im Bild 2.6 die Knotenfolgen

```
7, 5, 2, 6, 10, 15, 11, 12        (preorder)
2, 5, 6, 7, 10, 11, 12, 15        (postorder)
2, 6, 5, 12, 11, 15, 10, 7        (endorder)
```

Die Argumente beim Aufruf der Funktion twalk sind die Adresse des Wurzelknotens root und die Adresse einer vom Anwender bereitzustellenden Funktion action. Dieser Funktion wird die Adresse eines Zeigers auf die aktuelle Information übergeben, ausserdem die Mitteilung, wann der Knoten getroffen wurde (preorder, postorder, endorder, leaf) sowie die Niveauebene, in der sich der Knoten befindet. Die rekursive Funktion nodewalk erledigt diese Aufgabe.

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

struct node {
    char *key;
    struct node *left;
    struct node *right;
};

static void (*_action)();

void twalk(root, action)                    /* Aufruf von nodewalk */
    char *root;
    void (*action)();
{
    _action = action;
    if(root != NULL)
        nodewalk((struct node *)root, 0);
}

static void nodewalk(root, level) /* Durchmustern eines Binärbaums */
    struct node *root;
    int level;
{
    if(root->left == NULL && root->right == NULL) {
        (*_action)(&root->key, leaf, level);
        return;
    }
    (*_action)(&root->key, preorder, level);
    if(root->left != NULL)
        nodewalk(root->left, level + 1);
    (*_action)(&root->key, postorder, level);
    if(root->right != NULL)
        nodewalk(root->right, level + 1);
    (*_action)(&root->key, endorder, level);
}
```

Die in der Funktion benutzten Bezeichner preorder, postorder, endorder und leaf sind im File search.h definiert:

```
typedef enum {
    preorder,
    postorder,
    endorder,
    leaf
} VISIT;
```

Als Beispiel für die vom Nutzer bereitzustellende Funktion *action zeigen wir p_zahl, welche die Werte des im Bild 2.6 dargestellten Binärbaums sortiert ausgibt.

```
#include <search.h>

void p_zahl(pz, visit, level)          /* sortierte Ausgabe */
    char **pz;
    VISIT visit;
    int level;
{
    if(visit == postorder || visit == leaf)
        printf("%d\n", **(int **)pz);
}
```

Für jedes Blatt erfolgt der Aufruf dieser Funktion mit dem Argumentwert leaf. Für jeden anderen Knoten wird p_zahl dreimal aufgerufen, wobei nur der Aufruf mit dem Argument postorder von Bedeutung ist.

Betrachten wir nun den Aufbau eines Knotens:

```
struct node {
    char *key;
    struct node *left;
    struct node *right;
};
```

Ein einzelner Knoten enthält zwei Zeiger auf die Unterbäume und die Adresse der zum aktuellen Knoten gehörenden Information. In unserem Beispiel besteht die Information aus einem Integerwert. Der Zeiger pz in der Funktion p_zahl wird deshalb konvertiert ((int **)).

Dem Anwender der Bibliotheksfunktionen tdelete, tsearch, tfind und twalk ist der konkrete Aufbau einer Knoteninstanz des Typs node unbekannt. Aus diesem Grund ist der Zeiger auf den Wurzelknoten (Argument für twalk) einfach mit

```
char *root;
```

deklariert. Dieser Zeiger muss im Anwenderprogramm definiert und vor dem Aufbau des Baums (mit tsearch) mit NULL initialisiert werden. Die Adresse des Wurzelknotens ist für die Algorithmen zur Verwaltung des Binärbaums der Ausgangspunkt, wobei der Wert NULL einen leeren Baum kennzeichnet. Darauf kommen wir später zurück.

Für die Bibliotheksfunktionen wiederum ist der Aufbau der im Baum zu verwaltenden Informationen unbekannt. Deshalb wird key in der Form

```
char *key;
```

deklariert. Bei der Anwendung der Funktionen sollte immer auf eine geeignete Typangleichung geachtet werden.

Die Behandlung der Funktionen tsearch, tfind und tdelete erfolgt im Zusammenhang mit der dynamischen Speicherplatzverwaltung im Abschn. 3.3.7. Im Abschn. 3.5.2. zeigen wir die Anwendung der Bibliotheksfunktionen zur Arbeit mit Binärbäumen.

# 3. Programmierumgebung

Nachdem der komplette Sprachumfang vorgestellt wurde, soll in den folgenden Abschnitten auf verschiedene Aspekte der Anwendung von C eingegangen werden. Eine wichtige Rolle spielt hierbei die Standardbibliothek, die in gleicher Form in allen Systemumgebungen realisierbar ist. Diese Bibliothek unterstützt den Transport von C-Programmen unabhängig vom konkreten Betriebssystem und der zugrunde liegenden Hardware. Die natürlichste und wohl auch häufigste C-Programmierumgebung stellt das Betriebssystem UNIX dar. Die einheitlichen Schnittstellen zum Systemkern unterstützen die Portabilität von in C geschriebener Software. Sie sind somit für viele Programmierer relevant. Weiterhin werden die Möglichkeiten des Präprozessors und die Konventionen des Aufrufs von C-Programmen behandelt.

## 3.1. Argumentübergabe an die main-Funktion

Dieser Abschnitt erläutert, wie man die Arbeit eines Programms beim Aufruf steuern kann. Im weiteren gehen wir davon aus, dass der Aufruf eines C-Programms in der Form

<$par_0$> <$par_1$> <$par_2$> ... <$par_n$>

erfolgt. Dabei bezeichnet <$par_0$> den Filenamen des ausführbaren Programms, im folgenden auch Kommando genannt. Die weiteren Parameter sind Flags, die der Auswahl oder Präzisierung von Programmfunktionen dienen, oder andere Parameter, z.B. Filenamen. Unter einem Flag soll ein einzelner Buchstabe verstanden werden, dem das Zeichen - vorangestellt ist. Erlaubt sei eine beliebige Reihenfolge sowie die Zusammenfassung mehrerer Flags. Ausserdem kann ein Flag eine Zeichenfolge fordern, die direkt mit oder ohne Leerzeichen folgen muss.

Einheitliche Regeln zur Gestaltung der Kommandosyntax unterstützen die Nutzerfreundlichkeit eines Systems. Um die Einhaltung dieser Regeln zu unterstützen, existiert in der Standardbibliothek die Funktion getopt, auf die am Ende des Abschnitts eingegangen wird.

An die main-Funktion werden zwei Argumente übergeben. Das erste gibt die Anzahl der Aufrufparameter an, während als zweites Argument die Adresse eines Feldes übergeben wird. Jedes Feldelement enthält die Anfangsadresse eines Zeichenfeldes, das die Zeichenfolge eines einzelnen Aufrufparameters darstellt (Bild 3.1). Diese Konventionen für den Programmaufruf gelten in den meisten Programmierumgebungen, die C unterstützen. Aus sprachlicher Sicht sind sie jedoch nicht zwingend.

Es ist üblich, die entsprechenden formalen Parameter von main mit argc bzw. argv zu bezeichnen:

```
main(argc, argv)
    int argc;
    char *argv[];        /* oder:  char **argv; */
{

}
```

Die Deklaration von argv spiegelt wider, dass die Adresse eines Feldes von Zeigern auf Zeichen erwartet wird. Die beiden Formen **char** *argv[] und **char** **argv sind identisch.

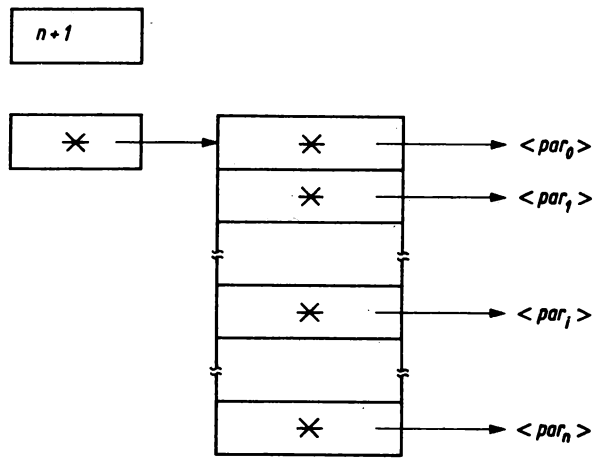



Bild 3.1. Argumente an die main-Funktion

Bezogen auf eine Kommandozeile zum Aufruf des C-Compilers

```
cc -o wurzel wurzel.c
```

erhält argc den Wert 4, da als erster Parameter der Programmname bereitgestellt wird. Weiterhin zeigt argv auf ein Feld mit 5 Zeigern. Dabei enthalten die einzelnen Feldelemente die Adressen der Zeichenketten "cc", "-o", "wurzel" und "wurzel.c". Der Wert des letzten Elements ist NULL und spezifiziert damit das Ende der Parameterliste:

```
argv [0]   -->   "cc"
argv [1]   -->   "-o"
argv [2]   -->   "wurzel"
argv [3]   -->   "wurzel.c"
argv [4]   ==    NULL
```

Verdeutlichen wir den Mechanismus der Parameterübergabe am Beispiel des Kommandos cat. Das Programm liest die Standardeingabe und schreibt in die Standardausgabe. Dabei soll mit dem Flag n die Numerierung der Zeilen und mit dem Flag v das "Sichtbarmachen" nicht druckbarer Zeichen gefordert werden können.

Bezogen auf dieses Beispiel sind folgende Programmaufrufe zulässig:

```
cat
cat -n              /* Zeilennummern ausgeben */
cat -v              /* Umwandlung nichtdruckbarer Zeichen */
cat -n -v           /* beide Programmfunktionen */
cat -v -n
cat -nv
cat -vn
```

Die Argumentauswertung erfolgt in der main-Funktion.

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

#define  MAX 512

short vanz = Ø;

main(argc, argv)                    /* cat (Version 1) */
    int argc;
    char *argv[];
{
    char zf[MAX];
    long nummer = Ø;
    register char *opt, **argp;
    short nanz = Ø;

    argp = argv;                                          /*1*/
    while(--argc > Ø && (*++argp)[Ø] == '-')              /*2*/
        for(opt = argp[Ø]+1; *opt != '\Ø'; opt++)         /*3*/
            switch(*opt) {
                case 'n' :
                    nanz++;
                    break;
                case 'v' :
                    vanz++;
                    break;
                default :
                    fprintf(stderr, "%s: bad argument\n", argv[Ø]);
                    exit(1);
            }
    while(getline(zf) != NULL) {
        if(nanz)
            printf("%6ld  ", ++nummer);
        putline(zf);
    }
}
```

Die durch Kommentare gekennzeichneten Konstruktionen können als Modell für die Auswertung von Aufrufparametern betrachtet werden. Der Zeiger argv wird aus Effektivitätsgründen in eine Registervariable argp umgespeichert (/*1*/). Anschliessend wird geprüft, ob weitere Aufrufparameter auszuwerten sind, wobei die Dekrementierung von argc vor dem Test erfolgt, um die Parameterauswertung im Fall argc gleich 1 abzubrechen (/*2*/). Das ist der Fall, wenn nur der Kommandoname angegeben war bzw. wenn alle Aufrufparameter verarbeitet sind. Der Ausdruck

```
(*++argp)[Ø] == '-'
```

prüft, ob das erste Zeichen des Parameters '-' ist. Ist das der Fall, so werden die nachfolgenden Zeichen als Flags interpretiert. Wichtig ist die Präfixinkrementierung von argp, da auf diese Weise der Kommandoname übergangen wird. Aufrufparameter, die keine Flags enthalten, werden ohne Mitteilung ignoriert. Die **for**-Anweisung untersucht den Rest des aktuellen Aufrufparameters hinter dem Zeichen - und ermöglicht dadurch insbesondere die Angabe mehrerer Flags in einem Parameter (/*3*/). Die **switch**-Konstruktion muss nur noch das konkrete Flagzeichen vermerken bzw. eine Fehlermitteilung bei ungültigen Flags ausgeben. Der Rest der main-Funktion steuert die eigentliche Aufgabe des Programms und verwendet zu dem Zweck die Funktionen getline und putline.

```
char *getline(z)            /* Einlesen einer Zeile */
    register char *z;
{
    register int c;
    register char *pz;

    pz = z;
    while((c = getchar()) != EOF)
        if((*pz++ = c) == '\n')
            break;
    if(c == EOF && pz == z)
        return(NULL);
    *pz = '\Ø';
    return(z);
}

void putline(z)             /* Ausgabe einer Zeile und Sichtbarmachen */
    register char *z;
{
    register int c;

    while(c = *z++)
        if(vanz && isprint(c) == Ø && isspace(c) == Ø)
            printf("\\%#o", c);
        else
            putchar(c);
}
```

In der Standardbibliothek ist die Funktion getopt (s. Anhang B) definiert, die die Auswertung der Aufrufparameter erleichtern und vor allem die Durchsetzung weitgehend einheitlicher Regeln der Kommandosyntax unterstützen soll.

```
int getopt(argc, argv, optstring) 1)
    int argc;
    char *argv[];
    char *optstring;

extern char *optarg;
extern int optind;
extern int opterr;
```

Die Funktion wird über argc, argv und optstring sowie die externen Variablen optarg, optind und opterr gesteuert. Die Parameter argc und argv besitzen die gleiche Bedeutung wie die gleichnamigen Parameter der main-Funktion. Die Zeichenkette optstring muss die definierten Flags enthalten. Folgt einem Flag in optstring ein Doppelpunkt (:), so wird in der Kommandozeile nach dem Flag eine Zeichenfolge erwartet. In diesem Fall wird die Adresse dieser Zeichenfolge in der externen Variablen optarg gespeichert. Die Funktion liefert als Ergebnis das nächste Flag bzw. das Zeichen '?' bei ungültigen Flags. Im letzteren Fall gibt getopt ausserdem eine Fehlernachricht aus, die man durch Setzen der externen Variablen opterr auf Ø unterdrücken kann. Sind keine Flags mehr vorhanden, wird EOF zurückgegeben.

Bei Verwendung von getopt zur Auswertung der Aufrufparameter ergeben sich für die main-Funktion des Kommandos cat folgende Änderungen:

---

1) Für jede der hier vorgestellten Standard- und Systemfunktionen geben wir den Kopf der Funktionsdefinition, die einzuschliessenden Files sowie die notwendigen Deklarationen für externe Variablen an.

```
#include <stdio.h>
#include <ctype.h>
#define  MAX 512
short vanz = Ø;

main(argc, argv)                          /* cat (Version 1) */
    int argc;
    char *argv[];
{
    char zf[MAX];
    long nummer = Ø;
    register int opt;
    short nanz = Ø;

    while((opt = getopt(argc, argv, "nv")) != EOF)
        switch(opt) {
            ...
            default:
                exit(1);
        }
    ...
}
```

Es folgt eine mögliche Implementation der Funktion getopt:

```
#include  <stdio.h>

char *optarg;
int optind = 1, opterr = 1;

int getopt(argc, argv, optstring)       /* Argumentauswertung  */
    int argc;
    char *argv[], *optstring;
{
    char *opt;
    static int inopt = Ø;

    if(optind >= argc || *argv[optind] != '-' && inopt == Ø)     /*1*/
        return(EOF);
    opt = argv[optind] + 1 + inopt;                              /*2*/
    if(*opt == '-') {                                            /*3*/
        optind++;
        return(EOF);
    }
    if(*opt == '\Ø')                                             /*4*/
        opt = " ";
    while(*optstring++ != *opt)                                  /*5*/
        if(*optstring == '\Ø') {
            if(opterr)
                fprintf(stderr, "illegal option flag: %c\n", *opt);
            return('?');
        }
    if(*optstring == ':') {                                      /*6*/
        inopt = Ø;
        optind++;
        if(opt[1] != '\Ø')
            optarg = &opt[1];                                    /*7*/
        else
            optarg = argv[optind++];                             /*8*/
    } else if(opt[1] != '\Ø')
        inopt++;
    else {
        inopt = Ø;
        optind++;
    }
    return(*opt);
}
```

Zunächst wird untersucht, ob noch ein weiterer Aufrufparameter vorhanden ist und ob es sich dabei um ein Flag handelt (/*1*/). Die **static**-Variable inopt wird über den Funktionsaufruf hinaus als Index in der aktuellen Flagzeichenkette benutzt. Demzufolge zeigt der Wert Ø für inopt an, dass eine neue Flagzeichenkette beginnen und mit dem Zeichen '-' eingeleitet werden muss. Folgt kein neues Flag, so liefert getopt als Funktionswert EOF.

Die Variable opt adressiert das nächste Flagzeichen innerhalb des aktuellen Aufrufparameters (/*2*/).

Als logisches Flagendekennzeichen ist das Flagzeichen '-' definiert (/*3*/).

Der Aufrufparameter "-" wird als ein leeres Flag betrachtet. Erlaubt ist dieses Flag, wenn in optstring ein Leerzeichen enthalten ist (/*4*/).

Die **while**-Schleife sucht nach einem gültigen Flag und gibt im Fehlerfall standardmässig eine Mitteilung aus (/*5*/).

Ist durch das Zeichen ':' in optstring festgelegt, dass nach dem entsprechenden Flag eine Zeichenfolge stehen muss (/*6*/), so wird der Zeiger optarg gesetzt. Dabei kann die Zeichenfolge dem Flag direkt (/*7*/) oder durch Leerzeichen getrennt folgen (/*8*/).

Der Aufrufparameterindex optind und der Index inopt werden für den nächsten Aufruf von getopt entsprechend modifiziert.

**Aufgabe 3.1.** Modifizieren Sie das Kommando cat so, dass nach dem Flag v ein Zeichen angegeben werden muss. Dieses Zeichen soll dann anstelle des Backslash \ die Oktalzahl für nicht druckbare Zeichen hervorheben. Benutzen Sie dabei die Funktion getopt.

## 3.2. Präprozessor

Vor der eigentlichen Compilierung erfolgt die Verarbeitung der Quelltextfiles durch einen Präprozessor, der die Files nach Zeilen durchsucht, die mit dem Zeichen # beginnen. Diese Zeilen werden vom Präprozessor ausgewertet und entsprechende Aktionen durchgeführt. Dem Zeichen # dürfen Leer- und Tabulatorzeichen folgen. Mittels Präprozessorsteueranweisungen ist es u.a. möglich,

- symbolische Konstanten und Makros zu definieren sowie die entsprechenden Ersetzungen im Quelltext vorzunehmen
- Files einzufügen
- eine bedingte Compilierung zu steuern.

Der C-Präprozessor unterstützt mit seinen funktionellen Möglichkeiten die Entwicklung grösserer Programmsysteme durch ein Programmiererkollektiv. Ausserdem werden geeignete Mittel bereitgestellt, um portable Software zu schreiben und damit den Transport auf unterschiedliche Zielrechner zu vereinfachen.

### 3.2.1. Definition symbolischer Konstanten und Makros

In vielen bisherigen Beispielen wurden bereits **#define**-Steueranweisungen verwendet, um symbolische Konstanten zu definieren, z.B.

```
#define  EOF (-1)
#define  MAX 512
```

Eine Steuerzeile der Form

```
#define  <bezeichner>   <zeichenfolge>
```

bewirkt, dass im restlichen Eingabefile jedesmal <bezeichner> durch <zeichenfolge> ersetzt wird. Eine Ersetzung findet nur dann nicht statt, wenn <bezeichner> innerhalb von Zeichenkonstanten oder Zeichenkettenkonstanten auftritt. Beispielsweise transformiert der Präprozessor einen Eingabetext

```
#define  LAENGE  8Ø
...
   if(x > LAENGE)
...
   printf("DIE LAENGE BETRAEGT %d", LAENGE);
...
```

in die Zeilen

```
...
   if(x > 8Ø)
...
   printf("DIE LAENGE BETRAEGT %d", 8Ø);
...
```

Wie man sieht, werden die Steuerzeilen selbst nicht in die Ausgabe übernommen.

Im übrigen ist <bezeichner> eine Folge von Buchstaben und Ziffern. Das erste Zeichen muss ein Buchstabe sein, wobei das Unterstreichungszeichen _ zu den Buchstaben gezählt wird. Es gelten also die gleichen Regeln zur Namensbildung wie für Variablenbezeichner.

In diesem Zusammenhang sei nochmals an eine in der C-Programmierpraxis übliche Konvention erinnert. Während Variablenbezeichner gewöhnlich aus Kleinbuchstaben bestehen, werden für symbolische Konstanten vorzugsweise Grossbuchstaben verwendet. Diese Verfahrensweise erleichtert die Unterscheidung zwischen Konstanten und Variablen und trägt somit zur Verbesserung der Lesbarkeit von C-Programmen bei.

Eine Besonderheit besteht darin, dass der Ersetzungstext selbst wiederum nach definierten symbolischen Konstanten durchsucht wird:

```
#define  MAX  1ØØ
#define  N    MAX/2
...
   x = MAX;
   y = N;
...
```

Die Präprozessorausgabe für dieses Beispiel sieht folgendermassen aus:

```
...
   x = 1ØØ;
   y = 1ØØ/2;
...
```

da N als eine Zeichenfolge MAX/2 definiert wurde.

Für den Ersetzungstext <zeichenfolge> existieren praktisch keine Einschränkungen. Normalerweise erstreckt er sich bis zum Ende der Zeile, es sind auch Fortsetzungszeilen erlaubt. Sie werden durch \ als letztes Zeichen der vorhergehenden Zeile gekennzeichnet:

```
#define  TEXT  "Mein erstes C-Pro\
gramm!\n"
...
   printf(TEXT);
...
```

Der Präprozessor bietet dem Compiler in diesem Fall folgenden Text zur weiteren Verarbeitung an:

```
...
   printf("Mein erstes C-Programm!\n");
...
```

Die Gültigkeit einer Konstantendefinition erstreckt sich bis zum Ende des Files, es sei denn, <bezeichner> wird mittels **#define** neu definiert oder die Definition wird durch eine Steuerzeile der Form

**#undef** <bezeichner>

aufgehoben.

Man kann beim Aufruf des Präprozessors mittels des cc-Kommandos verlangen, dass symbolische Konstanten definiert werden. Jeder nach einem D-Flag angegebene Bezeichner wird immer dann, wenn er im Eingabefile vorkommt, durch das Zeichen 1 ersetzt. Somit hat beispielsweise der Aufruf

cc -D<bezeichner$_1$> -D<bezeichner$_2$> <quell_filename>

die gleiche Wirkung wie die Definition von <bezeichner$_1$> und <bezeichner$_2$> am Anfang des Quellfiles <quell_filename>:

```
#define <bezeichner1>  1
#define <bezeichner2>  1
...
```

Die Definition eines nach dem D-Flag angegebenen Bezeichners kann mit dem Flag U (analog zu **#undef**) wieder aufgehoben werden. So hat der Compileraufruf

cc -D<bezeichner$_1$> <file$_1$> -U<bezeichner$_1$> <file$_2$>

zur Folge, dass die symbolische Konstante im File <file$_1$> definiert ist, im File <file$_2$> hingegen nicht. Das Flag U kann auch mehrfach angegeben werden.

Unter einer Makrodefinition versteht man eine Präprozessorsteueranweisung der Form

**#define** <makro_bezeichner>(<par$_1$>,..., <par$_n$>) <zeichenfolge>

Jedes folgende Auftreten eines Makroaufrufes

<makro_bezeichner>(<argument$_1$>,...,<argument$_n$>)

führt zur Substitution durch <zeichenfolge>, wobei für jeden Parameter

<par$_i$> im Ersetzungstext das entsprechende Aufrufargument <argument$_i$> eingesetzt wird. Die Argumente stellen in syntaktischer Hinsicht Zeichenfolgen dar, die Parameter jedoch sind Bezeichner. Beim Aufruf muss die Argumentzahl mit der Anzahl der definierten Parameter übereinstimmen. Ausserdem wird gefordert, dass die öffnende runde Klammer in der Makrodefinition unmittelbar auf <makro_bezeichner> folgt. Hinsichtlich Fortsetzungszeilen, Gültigkeitsbereich usw. gelten für Makros die gleichen Regeln wie für symbolische Konstanten.

```
#define abs(P)  (P < Ø ? -(P) : P)
```

Diese Definition stellt beispielsweise einen Makro abs zur Verfügung, der den absoluten Betrag des Parameters P berechnet. Betrachten wir nun folgende Makroaufrufe:

```
x = abs(y);
x = abs(y-z);
```

Sie werden ersetzt durch

```
x = (y<Ø ? -(y) : y);
x = (y-z < Ø ? -(y-z) : y-z);
```

Das zweite Beispiel zeigt, dass bei der Makrodefinition im Ersetzungstext <zeichenfolge> sorgfältig auf die Klammernsetzung geachtet werden muss, um die erforderliche Reihenfolge der Ausdrucksauswertung zu gewährleisten.

An dieser Stelle weisen wir darauf hin, dass die Verwendung von Makros anstelle von Funktionen auch Konsequenzen hinsichtlich der Kodeeffizienz nach sich zieht. Einerseits erfordern Makros keinen Funktionsaufruf, d.h., es wird die sonst zur Vor- und Nachbereitung eines Funktionsaufrufes notwendige Befehlsabarbeitung eingespart. Andererseits wird an jeder Stelle im Programm, wo ein Makroaufruf vorkommt, der bei der Makroexpansion entstehende Befehlskode eingefügt. Die Konsequenz ist offensichtlich: Makros führen i.allg. zur Verbesserung des Laufzeitverhaltens bei gleichzeitiger Erhöhung des Speicherplatzbedarfes.

Ausserdem sollte man Argumentausdrücke vermeiden, die Nebeneffekte hervorrufen. Bezogen auf den abs-Makro ergeben sich z.B. Probleme bei einem Aufruf

```
y = abs(*px++);
```

**Aufgabe 3.2.** Warum ist dieser Makroaufruf nicht korrekt?

**Aufgabe 3.3.** Definieren Sie einen Makro quadrat(x), der die zweite Potenz von x berechnet. Überprüfen Sie die Richtigkeit!

### 3.2.2. Einfügen von Files

Eine Präprozessoranweisung

```
#include <<filename>>
```

bewirkt den Einschluss des Files <filename>. Die Suche nach <filename> erfolgt zunächst in einer Folge vordefinierter Verzeichnisse. Existieren

keine vordefinierten Verzeichnisse oder kann <filename> dort nicht gefunden werden, so wird die Suche in einem Standardverzeichnis fortgesetzt. Die Vorgabe eines oder mehrerer Verzeichnisse ist beim Aufruf des Präprozessors möglich:

cc -I<verzeichnis$_1$> -I<verzeichnis$_2$> ... <quellfilename>

Die Reihenfolge der I-Angaben legt dabei die Reihenfolge fest, in der die so vorgegebenen Verzeichnisse nach <filename> durchsucht werden. Man kann jedoch auch verlangen, dass die Suche nach <filename> in dem Verzeichnis beginnt, in dem sich auch das Eingabefile <quellfilename> befindet:

**#include** "<filename>"

Der Präprozessor behandelt die eingeschlossenen Files in der gleichen Weise wie das ursprüngliche File, d.h., er wertet Steuerzeilen aus, führt die Textsubstitution durch usw. Erlaubt sind insbesondere auch Schachtelungen von **#include**-Anweisungen.

Wie in verschiedenen Beispielen angedeutet, wird der Mechanismus des Einfügens von Files benutzt, um Konstantendefinitionen, Deklarationen, Funktionen und Makros der Standardbibliothek verfügbar zu machen. Eine wichtige Rolle spielt diese Möglichkeit bei der Entwicklung grosser Programme, insbesondere dann, wenn sie aus einer Vielzahl einzelner Quelltextfiles bestehen. Um zu gewährleisten, dass sich alle Quelltextfiles auf die gleichen Objekte beziehen, werden symbolische Konstanten, Makros und global benötigte Deklarationen in sog. Headerfiles zusammengefasst. In den einzelnen Quellfiles muss dann nur noch die entsprechende **#include**-Anweisung enthalten sein. Ausserdem ist dafür zu sorgen, dass bei einer Änderung des Headerfiles alle betroffenen Quellfiles neu übersetzt werden. Wir kommen auf diese Methode im Zusammenhang mit der Diskussion eines komplexen Beispiels (s. Abschn. 3.5.4.) zurück.

### 3.2.3. Bedingte Compilierung

Die Mittel zur Steuerung der bedingten Compilierung können für verschiedene Zwecke sinnvoll genutzt werden, insbesondere zur Unterstützung der Portabilität von Programmen auf dem Quelltextniveau. Das Grundprinzip besteht darin, dass abhängig von einer bestimmten Bedingung die darauffolgenden Zeilen des Eingabefiles durch den Präprozessor normal verarbeitet oder einfach übergangen werden. Ein einfaches Beispiel soll die Vorgehensweise verdeutlichen:

```
...
#ifdef A
    printf("A ist definiert!");
#endif
...
```

Bei der Auswertung der **#ifdef**-Steuerzeile untersucht der Präprozessor, ob der Bezeichner A vorher mittels **#define** definiert wurde. Wenn dies der Fall ist, dann wird die printf-Anweisung an den Compiler weitergereicht. Ist A für den Präprozessor nicht definiert, so erfolgt ein Übergehen dieser Anweisung. In dem Fall könnte jedoch eine bestimmte Aktion erforderlich sein:

```
...
#ifdef A
    printf("A ist definiert!");
#else
    printf("A ist nicht definiert!");
#endif
...
```

Wenn der Bezeichner A definiert ist, so erfolgt die Ausgabe von

```
printf("A ist definiert!");
```

Ansonsten übergeht der Präprozessor diese Anweisung und gibt statt dessen

```
printf("A ist nicht definiert!");
```

aus. Den gleichen Effekt erreicht man durch folgende Konstruktionen:

```
...
#ifndef A
    printf("A ist nicht definiert!");
#else
    printf("A ist definiert!");
#endif
...
```

Natürlich können zwischen den Steueranweisungen **#ifdef** oder **#ifndef** und **#else** bzw. zwischen **#else** und **#endif** beliebig viele Zeilen stehen.

Es existiert noch eine dritte Möglichkeit zur Steuerung der bedingten Compilierung, bei der geprüft wird, welchen Wert ein Konstantenausdruck (vgl. Abschn. 2.1.8.) besitzt. Erlaubt sind dabei alle Formen von Konstantenausdrücken ausser **sizeof**-Ausdrücken und **enum**-Konstanten. Ist der Wert des Konstantenausdrucks ungleich Ø, so werden alle Zeilen zwischen **#if** und **#else** vom Präprozessor verarbeitet, bei einem Wert gleich Ø werden diese Zeilen übergangen.

```
#if <konst_ausdruck>
...
#else
...
#endif
```

In <konst_ausdruck> kann auch ein Ausdruck der Form

**defined** <bezeichner>

oder

**defined**(<bezeichner>)

vorkommen, wobei der Ausdruck den Wert 1 bzw. Ø besitzt, je nachdem, ob <bezeichner> für den Präprozessor definiert ist oder nicht. 1)

Die Konstruktionen zur Steuerung der bedingten Compilierung können beliebig geschachtelt sein.

Während der Programmentwicklung ist es oft notwendig, an kritischen Stellen im Quelltext sog. Testpunkte einzubauen. Im ausgetesteten Programm

---

1) Diese Möglichkeit zur Bildung von Konstantenausdrücken ist neu in den Präprossezor aufgenommen worden /50/.

soll der überflüssige Kode dann nicht mehr enthalten sein. Mit Hilfe der Präprozessorsteuerung können solche Teile bei Bedarf einfach ein- oder ausgeschlossen werden.

```
f()
{
    ...
#ifdef DEBUG
    dumptab();
#endif
    ...
}

#ifdef DEBUG
dumptab()
{

}
#endif
```

Ist DEBUG nicht definiert, so werden vom Compiler weder der Ruf der Funktion dumptab noch die Funktionsdefinition selbst übersetzt.

Um die Testausgaben der Funktion dumptab einzuschliessen, muss entweder die Zeile

```
#define DEBUG
```

vorher im Quellfile enthalten sein, oder der Bezeichner DEBUG beim Aufruf des Compilers wie folgt angegeben werden:

```
cc -DDEBUG ...
```

Der Compileraufruf

```
cc -DDEBUG file1.c -UDEBUG file2.c
```

bewirkt, dass DEBUG in file1.c definiert ist, in file2.c jedoch nicht.

**Aufgabe 3.4.** Zerlegen Sie das Programm cat aus Abschn. 3.1. in 3 Files und ein Headerfile. Fügen Sie am Beginn jeder Funktion Testdruckanweisungen ein, die mittels bedingter Übersetzung aktiviert und deaktiviert werden können.

### 3.2.4. Zeilennumerierung

Verschiedene Programme erzeugen selbst wieder C-Programme. Hierbei besteht ein Problem darin, wie man von einer Fehlernachricht des C-Compilers bei der Übersetzung dieser Programme auf die Fehlerursache im Ursprungseingabefile schliessen kann. Es ist deshalb möglicherweise erforderlich, die Zeilennumerierung gezielt zu beeinflussen. Durch

```
#line <konstante> "<filename>"
```

kann dem Compiler mitgeteilt werden, dass die Nummer der nächsten Quelltextzeile durch <konstante> definiert wird und <filename> das aktuelle Eingabefile bezeichnet. Wird <filename> weggelassen, so bleibt die bisherige Filenamenbezeichnung bestehen.

## 3.3. Standardbibliothek

Die Ein- und Ausgabe, die Zeichenkettenverarbeitung und andere Standardfunktionen sind nicht Bestandteil der Sprache C selbst. In der Standardbibliothek sind jedoch eine Reihe nützlicher Funktionen und Makros definiert, die zu jedem C-Compiler in einer bestimmten Betriebssystemumgebung bereitgestellt werden. Im Anhang B ist eine vollständige Übersicht über den in /50/ definierten Standard angegeben. Benutzt man diese Funktionen bei der Programmentwicklung in C, so ist ein hoher Grad an Portabilität der entwickelten Softwareprodukte gewährleistet.

Einen wesentlichen Teil der Standardbibliothek bilden die Funktionen der Ein- und Ausgabe. Um diese Funktionen nutzen zu können, muss im Quelltextfile am Anfang die Zeile

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

enthalten sein. Das File stdio.h enthält Makros und Definitionen, die von den E/A-Funktionen benötigt werden. Die spitzen Klammern < und > geben an, dass sich dieses File in einem Standardverzeichnis (im UNIX das Verzeichnis /usr/include) befindet. In den folgenden Abschnitten werden die wichtigsten Ein- und Ausgabemöglichkeiten und die Funktionen zur dynamischen Speicherplatzzuordnug im UNIX bzw. in UNIX-kompatiblen Systemen anhand von Beispielen erläutert. Beim Studium ist ein ergänzendes Lesen der Funktionsbeschreibungen in Anhang B empfehlenswert.

### 3.3.1. Zeichenweise Ein- und Ausgabe

Die einfachste Eingabemöglichkeit ist mit dem zeichenweisen Lesen der Standardeingabe mittels getchar gegeben:

```
int getchar()
```

Bei jedem Aufruf wird ein Zeichen übergeben und bei Fileende EOF. Der Wert von EOF ist im File stdio.h als (-1) definiert, bei Tests sollte aber immer die symbolische Konstante verwendet werden.

Die Standardeingabe ist normalerweise dem Nutzerterminal zugeordnet. Im UNIX kann diese Standardzuordnung mit Hilfe des Steuerzeichens < verändert werden. Diesen Vorgang nennt man **Fileumlenkung**. Durch die Kommandozeile

```
<kommando> <<infile>
```

wird der Eingabestrom vor der Kommandoausführung vom Terminal auf das File <infile> umgelenkt. Die Fileumlenkung wird für die Dauer der Ausführung von <kommando> vorgenommen. Sie geschieht unabhängig vom gerufenen Programm, das von dieser Manipulation selbst nichts merkt, da es die Zeichenfolge <<infile> nicht als Aufrufparameter erhält.

In analoger Weise erfolgt durch putchar die Ausgabe eines Zeichens in die Standardausgabe:

```
int putchar(c)
    int c;
```

Die Standardausgabe ist normalerweise ebenfalls dem Terminal zugeordnet

und kann durch

<kommando> ><outfile>

in das File <outfile> umgelenkt werden.

Die Standardausgabe eines Kommandos kann mit Hilfe des Steuerzeichens | mit der Standardeingabe eines anderen Kommandos verbunden werden:

$<kommando_1>$ | $<kommando_2>$ | ... | $<kommando_n>$

Die von $<kommando_i>$ erzeugten Ausgabedaten dienen dabei als Eingabedaten für $<kommando_{i+1}>$. Eine solche Verbindung wird Pipeline oder kurz **Pipe** genannt /8/,/43/,/32/,/33/. Die beteiligten Programme brauchen für diese Verbindung wiederum nicht besonders präpariert zu sein. Durch die Leistungen der Betriebssystemumgebung kann man mit diesen Mitteln schon recht sinnvolle Programme schreiben. In der Tat gibt es sehr viele Programme, die nur über einen Eingabestrom und einen Ausgabestrom verfügen. Diese Programme bezeichnet man als **Filter** /31/.

Als Beispiel betrachten wir die Aufgabe, ein Kommando umlaut zu entwickeln, das einen Text mit Umlauten auf einen Drucker ausgibt, bei dem diese Zeichen nicht im Zeichenvorrat enthalten sind. Die Umlaute seien im Text durch Setzen der im ASCII-Zeichensatz freien höchsten Bitposition gekennzeichnet. Auf dem Drucker kann man einen Umlaut durch Überdruck des entsprechenden Buchstaben (a, o, u bzw. A, O, U) mit dem Anführungszeichen (") darstellen. Bei Grossbuchstaben muss das Anführungszeichen etwas höher gedruckt werden.

```
#include  <stdio.h>

#define  Hoch putchar(0x1b); putchar(0x5b); putchar(0x30);          \
              putchar(0x30); putchar(0x31); putchar(0x75);

#define  Tief putchar(0x1b); putchar(0x5b); putchar(0x30);          \
              putchar(0x30); putchar(0x31); putchar(0x65);

#define  K_umlaut(c) putchar(c); putchar('\b'); putchar('"'); break;

#define  G_umlaut(c) putchar(c); putchar('\b'); Hoch;               \
                                 putchar('"'); Tief; break;

main()                  /* Ausgabe von Umlauten durch Überdruck */
{
    register int c;

    while((c = getchar()) != EOF) {
        switch(c) {
            case 'a'+0x80:
            case 'o'+0x80:
            case 'u'+0x80:
                K_umlaut(c);
            case 'A'+0x80:
            case 'O'+0x80:
            case 'U'+0x80:
                G_umlaut(c);
            default:
                putchar(c);
                break;
        }
    }
}
```

Der einfache Überdruck in den Makros K_umlaut und G_umlaut ist leicht mit dem Steuerzeichen Backspace \b realisierbar. Das Hochstellen bzw. Tiefstellen der Druckwalze wird bei dem hier verwendeten Typenraddrucker mit der Steuerfolge Øx1b5b3Ø3Ø3175 bzw. Øx1b5b3Ø3Ø3165 realisiert. Diese geräteabhängigen Steuerfolgen sind in den Makros Hoch bzw. Tief untergebracht, um eine eventuelle Anpassung an andere Druckertypen zu erleichtern.

Angenommen, in den Files text1, text2 und text3 sind Texte mit Umlauten enthalten. Das Drucken dieser Files kann dann mit der Kommandozeile

```
cat text1 text2 text3 | umlaut '>/dev/lpr
```

erfolgen. Dabei liest cat nacheinander die Files text1, text2 sowie text3 und gibt sie in das Standardausgabefile aus. Die Standardausgabe von cat ist mit der Standardeingabe von umlaut verbunden. Nach der Behandlung der speziell markierten Umlaute schreibt umlaut die resultierenden Ausgabedaten wiederum in das Standardausgabefile. Im UNIX ist es möglich, physische Geräte über Spezialfilenamen (z.B. /dev/lpr für den Drucker) wie gewöhnliche Files zu behandeln.

Übrigens sind getchar und putchar ebenfalls als Makros implementiert, um den Aufwand für Funktionsaufrufe zu reduzieren (vgl. Abschn. 3.4.3.).

**Aufgabe 3.5.** Wie muss obige Kommandozeile verändert werden, um den in den Files text1, text2 und text3 enthaltenen Text vor der Transformation der Umlaute mittels des Formatierungsprogramms format im Abschn. 2.9.1. zu formatieren und auszugeben?

### 3.3.2. Formatgesteuerte Ausgabe

Die Funktion printf wurde in den vorangehenden Abschnitten bereits verwendet. Eine vollständige Beschreibung ist im Anhang B gegeben. Hier sollen anhand von Beispielen die wichtigsten der vielfältigen Formatierungsmöglichkeiten erläutert werden. Wichtig ist zunächst, dass printf ebenfalls in die Standardausgabe ausgibt. Aufrufe von printf und putchar können somit gemischt auftreten.

```
int printf(format [, arg ] ...)
    char *format;
```

Entsprechend der Zeichenkette format verarbeitet printf die weiteren Argumente und gibt das Ergebnis in die Standardausgabe aus. In format können sowohl Zeichen enthalten sein, die lediglich kopiert werden, als auch Konvertierungsvorschriften, die angeben, wie das zugehörige Argument zu konvertieren und zu formatieren ist. Die Konvertierungsvorschriften legen dabei fest, wieviel weitere Argumente an printf übergeben werden.

Eine Konvertierungsvorschrift wird mit dem Zeichen % eingeleitet und besitzt die allgemeine Form

```
%[<flags>][<feldbreite>][.[<genauigkeit>]][l]<konv_zeichen>
```

Immer muss also ein Konvertierungszeichen <konv_zeichen> (d, o, u, x, X, f, e, E, g, G, c, s oder %) vorhanden sein. Fehlt ein solches Zeichen, so ist das Ergebnis undefiniert. Das Konvertierungszeichen % steht nur für sich selbst und benötigt kein Argument. Der Funktionsaufruf

```
printf("%4.2f %%", 3.25);
```

bewirkt z.B. die Ausgabe der Zeichenkette "3.25 %" in die Standardausgabe.

Das Zeichen l unmittelbar vor dem Konvertierungszeichen zeigt für die Konvertierungszeichen d, o, u, x oder X an, dass das zugehörige Argument vom Typ **long int** ist. Vor anderen Konvertierungszeichen wird der Buchstabe l ignoriert.

Bei einem Funktionsaufruf

```
printf("%ld", 111111);
```

ist die Angabe des Buchstabens l unbedingt erforderlich, da z.B. auf 16-Bit-Rechnern die Konstante 111111 vom Typ **long int** ist. Fehlt l, so würde nur ein **int**-Argument (2 Byte) verarbeitet werden und demzufolge ein unerwartetes Ergebnis entstehen. Folgen weitere Konvertierungsvorschriften in dieser Formatzeichenkette, so sind auch alle weiteren Ergebnisse unbrauchbar, da die Argumentverarbeitung aus dem "Takt" geraten ist. Es gehört zum guten Programmierstil, diese Notation auch dann zu verwenden, wenn **long**- und **int**-Werte den gleichen Wertebereich besitzen (32-Bit-Rechner).

Mit den Angaben <flags>, <feldbreite> und <genauigkeit> wird die Art der Darstellung im Ausgabebereich, wie Links- bzw. Rechtsausrichtung, minimale Länge des Ausgabebereichs und die Genauigkeit, festgelegt. In unserem Beispiel "%4.2f %%" ist die Feldbreite 4, d.h., es werden mindestens 4 Stellen für die Darstellung bereitgestellt. Erfordert das Argument mehr Stellen, so erfolgt die Erweiterung des Bereichs, andernfalls wird nach rechts ausgerichtet und links mit Leerzeichen aufgefüllt. Die Genauigkeit 2 gibt beim Konvertierungszeichen f die Stellen nach dem Dezimalpunkt an. Weitere Beispiele sollen die Wirkung dieser Angaben auf die verschiedenen Konvertierungszeichen zeigen.

Betrachten wir zunächst die Konvertierung der Zeichenkettenkonstanten "Programmiersprache". Zur Verdeutlichung des sich ergebenden Ausgabebereichs dient dabei der senkrechte Strich.

```
%s        |Programmiersprache|
%6s       |Programmiersprache|
%.8s      |Programm|
%25s      |        Programmiersprache|
%-25s     |Programmiersprache        |
%25.8s    |                Programm|
%-25.8s   |Programm                  |
```

Mit der Konvertierungsvorschrift %s wird die vollständige Zeichenkette bis zum Zeichen \0 ausgegeben. Die Feldbreitenangabe 6 hat hier keine Wirkung, da die Zeichenkette länger ist. Die Angabe der Genauigkeit mit 8 hat hingegen ein Abschneiden der Zeichenkette zur Folge. Der Wert 25 für die Feldbreite vergrössert den Ausgabebereich, wobei nach rechts ausgerichtet und mit Leerzeichen aufgefüllt wird. Das Flag - führt zur Linksausrichtung.

Im Bild 3.2 ist die Wirkung einiger Konvertierungsvorschriften für Integerwerte am Beispiel des **int**-Wertes 7661 dargestellt. Innerhalb einer Zeile variieren dabei die Konvertierungszeichen, innerhalb einer Spalte die Angaben, die die Darstellung im Ausgabebereich beeinflussen.

Die Konvertierungszeichen o, x und X interpretieren ihre Argumente immer als **unsigned**-Werte.

| | <kz>=d | <kz>=o | <kz>=x | <kz>=X |
|---|---|---|---|---|
| %<kz> | \|7661\| | \|16755\| | \|1ded\| | \|1DED\| |
| %2<kz> | \|7661\| | \|16755\| | \|1ded\| | \|1DED\| |
| %6<kz> | \| 7661\| | \| 16755\| | \| 1ded\| | \| 1DED\| |
| %-6<kz> | \|7661 \| | \|16755 \| | \|1ded \| | \|1DED \| |
| %.7<kz> | \|ØØØ7661\| | \|ØØ16755\| | \|ØØØ1ded\| | \|ØØØ1DED\| |
| %#<kz> | \|7661\| | \|Ø16755\| | \|Øx1ded\| | \|ØX1DED\| |

Bild 3.2. Integerkonvertierung

Eine neue Variante der Konvertierungsvorschrift ist in der 5.Zeile der Tabelle mit der Genauigkeit 7 gegeben. Diese Angabe bewirkt, dass mindestens 7 Ziffern ausgegeben werden. Ist der Wert kleiner, so werden führende Nullen aufgefüllt.[1]

Das Flag # bewirkt eine Ausgabe von Oktal- und Hexadezimalzahlen in der üblichen C-Syntax.

Für das Konvertierungszeichen d sind noch die Flags + und blank (Leerzeichen) interessant. Betrachten wir deshalb die Konvertierung eines zweiten negativen Arguments -7661.

| | |
|---|---|
| %+6d\|%+6d | \| +7661\| -7661\| |
| % 6d\|% 6d | \| 7661\| -7661\| |
| %+d\|%+d | \|+7661\|-7661\| |
| % d\|% d | \| 7661\|-7661\| |

Das Flag + erzwingt also die Ausgabe eines Vorzeichens, und bei Angabe eines Leerzeichens (blank) wird zumindest der Platz dafür berücksichtigt.

Abschliessend geben wir noch einige Konvertierungsbeispiele für Realzahlen an. Hierfür existieren die Konvertierungszeichen f, e, E, g und G, die in ihrer Bedeutung im Bild 3.3 erläutert sind. In den Zeilen der Tabelle sind verschiedene Konvertierungsvorschriften angegeben, deren Wirkung auf drei Realwerte gezeigt wird.

| | 3.24 | -23.Ø | Ø.ØØØØØ23 |
|---|---|---|---|
| %f | \|3.24ØØØØ\| | \|-23.ØØØØØØ\| | \|Ø.ØØØØØ2\| |
| %e | \|3.24ØØØØe+ØØ\| | \|-2.3ØØØØØe+Ø1\| | \|2.3ØØØØØe-Ø6\| |
| %g | \|3.24\| | \|-23\| | \|2.3e-Ø6\| |
| %1Ø.1f | \| 3.2\| | \| -23.Ø\| | \| Ø.Ø\| |
| %+5.ØE | \|+3E+ØØ\| | \|-2E+Ø1\| | \|+2E-Ø6\| |

Bild 3.3. Konvertierung von Realwerten

---

1) Die Situation ist insofern etwas heikel, weil ältere Versionen hierfür eine führende Null vor der Feldbreite benutzen (%Ø7d).

Wichtig ist hier, dass die Genauigkeit die Anzahl der Stellen nach dem Dezimalpunkt bzw. bei g und G die Anzahl der signifikanten Ziffern angibt. Der Standardwert ist 6. Beim Konvertierungszeichen g werden die Ziffern nach dem Komma nur ausgegeben, wenn sie verschieden von Null sind. Die Grossbuchstaben E und G führen nur zu einer anderen Darstellung des Exponentenkennzeichens.

Die Konvertierungsfunktionen sprintf bzw. fprintf arbeiten nach den gleichen Konvertierungsregeln, wobei die Ausgabe in ein Zeichenfeld bzw. in ein File erfolgt (s. Anhang B).

### 3.3.3. Formatgesteuerte Eingabe

Die Eingabefunktion scanf liest die Standardeingabe, konvertiert die gelesenen Zeichenfolgen und speichert die Werte in die dafür vorgesehenen Variablen.

```
int scanf(format [, pointer ] ...)
    char *format;
```

Als erstes Argument erwartet scanf eine Zeichenkette, die Konvertierungsvorschriften der allgemeinen Form

```
%[*][<feldbreite>][l|h]<konv_zeichen>
```

enthält. Eine Konvertierungsvorschrift beschreibt, wie ein Eingabefeld interpretiert und konvertiert werden soll. Im allgemeinen fordert jede Konvertierungsvorschrift die Übergabe eines Arguments pointer. Dieses Argument legt die Speicheradresse fest, wohin der konvertierte Eingabewert zu speichern ist. Folgt dem % als nächstes das Zeichen *, so wird zwar eine Konvertierung durchgeführt, die Wertzuweisung aber unterdrückt. (Demzufolge darf man für eine solche Konvertierungsvorschrift kein Argument angeben.) Mit der Angabe des Feldbreitenwertes <feldbreite> kann die maximale Länge des zugehörigen Eingabefeldes begrenzt werden. Unter einem Eingabefeld ist dabei eine Zeichenfolge zu verstehen, die keine Trennzeichen (Leerzeichen, Tabulator, Newline) enthält.

Als Konvertierungszeichen können %, n, d, u, o, x, i, e, f, g, s, c und [ angegeben werden, wobei % bedeutet, dass genau dieses Zeichen als nächstes in der Eingabe erwartet wird. (Ein Argument pointer wird für die Konvertierungsvorschrift nicht benötigt.) Die Buchstaben l oder h vor den Konvertierungszeichen d, u, o und x geben an, dass das zugehörige Argument pointer ein Zeiger auf **long**- bzw. **short**-Objekte ist. Der Buchstabe l muss auch vor f, e und g verwendet werden, um das Argument als **double**-Zeiger zu kennzeichnen.

Die Ausführung von scanf wird normalerweise beendet, wenn alle Konvertierungsvorschriften abgearbeitet worden sind. Tritt bei einer Konvertierung eine Fehlerbedingung auf, d.h., passt ein Zeichen in der Standardeingabe nicht zu der entsprechenden Konvertierungsvorschrift, so erfolgt an dieser Stelle der Abbruch der Verarbeitung. Als Funktionswert liefert scanf die Anzahl der erfolgreich durchgeführten Konvertierungen, mit denen Wertzuweisungen an Variable verbunden sind. Die Funktion gibt den Wert EOF genau dann zurück, wenn die Fileendebedingung bereits vor der ersten Konvertierung aufgetreten ist.

Ausser Konvertierungsvorschriften kann die Zeichenkette format auch Trennzeichen enthalten, die einfach ignoriert werden, sowie beliebige

andere Zeichen (ausser %), die als nächste Zeichen in der Eingabe vorkommen müssen.

Die Wirkung einiger Konvertierungsvorschriften soll bei einer in der Standardeingabe gegebenen Zeichenfolge

```
|229 1777  Øxff 3.14   4.987e-2|
```

verdeutlicht werden. Die senkrechten Striche kennzeichnen Anfang und Ende der Eingabe. Ausserdem seien folgende Variablendefinitionen gegeben:

```
int fval, n, m;
short j;
float x;
double y;
char z, f[2Ø];
```

Betrachten wir zunächst den Aufruf

```
fval = scanf("%d%hd%i%f%lf", &n, &j, &m, &x, &y);
```

Im Ergebnis dieser Anweisung erhalten die Variablen n, j bzw. m die Dezimalzahlen 229, 1777 bzw. 255 (entspricht der hexadezimalen C-Schreibweise Øxff) zugewiesen, x erhält den Wert 3.14 sowie y den Wert Ø.Ø4987. Als Funktionswert liefert scanf den Wert 5, da alle fünf Konvertierungsvorschriften erfolgreich realisiert werden konnten.

Dagegen bewirkt der Aufruf

```
fval = scanf("%2d%d%s%*s%*f%5f%c%hd", &n, &m, f, &x, &z, &j);
```

folgende Wertzuweisungen:

- die Variable n erhält den Dezimalwert 22 und m den Dezimalwert 9
- dem Feld f wird die Zeichenkette "1777" zugewiesen (f[Ø]='1', f[1]='7', f[2]='7', f[3]='7', f[4]='\Ø')
- die Eingabebereiche mit den Zeichenfolgen Øxff und 3.14 werden übergangen
- die **float**-Variable x erhält den Wert 4.987,
- z das Zeichen 'e' und
- j den Wert -2
- der Variablen fval wird 6 zugewiesen.

Nun wird folgender scanf-Aufruf auf diese Zeichenfolge in der Eingabe angewendet:

```
fval = scanf("%6d%o%c%2s", &n, &m, &f[Ø], &f[1]);
```

Das ergibt:

- die **int**-Variablen n bzw. m erhalten die Dezimalwerte 229 bzw. 511 (entspricht oktal Ø1777)
- dem Feldelement f[Ø] wird ' ' zugewiesen, da bei dem Konvertierungszeichen c das sonst übliche Übergehen von Trennzeichen unterdrückt wird
- die Werte von f[1], f[2] bzw. f[3] sind 'Ø', 'x' bzw. '\Ø'
- die Variable fval erhält den Funktionswert 4.

Der nächste Aufruf von scanf setzt die Verarbeitung der Eingabe an dieser Stelle fort:

```
fval = scanf("%x%*c%c%le%s", &n, &z, &y, f);
```

Das Ergebnis lautet:

- die Variable n erhält den Dezimalwert 255 (hexadezimal Øxff)
- das Zeichen z besitzt den Zeichenwert '3' (das vorangehende Leerzeichen wurde mittels %*c übergangen!)
- die **double**-Variable y hat den Wert Ø.14
- dem Zeichenfeld f wird die Zeichenkette "4.987e-2" und
- fval der Wert 4 zugeordnet.

Den Abschluss der Beispiele bildet ein einfaches Tischrechnerprogramm, das für Eingaben der Form

```
<realzahl> <op> <realzahl>
```

jeweils die Summe (bei <op> gleich +), die Differenz (-), das Produkt (*) oder den Quotienten (/) berechnet.

```
#include <stdio.h>

main()                                  /* einfacher Tischrechner */
{
    double operand1, operand2;
    char op[2];

    while(scanf("%lf %1s %lf", &operand1, op, &operand2) != EOF) {
        switch(op[Ø]) {
            case '+':
                operand1 += operand2;
                break;
            case '-':
                operand1 -= operand2;
                break;
            case '*':
                operand1 *= operand2;
                break;
            case '/':
                operand1 /= operand2;
                break;
            default :
                printf("illegal operator\n");
                continue;
        }
        printf(" = %g\n",operand1);
    }
}
```

Neben der Konvertierungsfunktion scanf gibt es die Funktionen fscanf bzw. sscanf, die ein angegebenes File bzw. ein Zeichenfeld lesen.

### 3.3.4. Filezugriff

Bisher wurde der Filezugriff nur über die bereits von der Betriebssystemumgebung zur Verfügung gestellten Standardfiles realisiert. Hier soll nun gezeigt werden, dass vom Programm aus auf benannte Files zugegriffen

werden kann. Wie allgemein üblich, muss ein File vor seiner Verarbeitung geöffnet werden. Für diesen Zweck gibt es die Bibliotheksfunktion fopen.

```
FILE *fopen(file_name, type)
    char *file_name;
    char *type;
```

Als Funktionsargumente sind beim Aufruf von fopen der Filename file_name und der Zugriffsmodus type als Zeichenketten anzugeben. Dabei legt type fest, wie auf das File zugegriffen werden soll. Mögliche Zugriffsmodi sind Lesen ("r"), Schreiben ("w"), Fortschreiben ("a") sowie Lesen und Schreiben ("r+", "w+", "a+").

Die Funktion fopen liefert einen Zeiger auf eine Strukturinstanz des Typs FILE, einen sog. **Filepointer**. Dieser Filepointer wird bei den weiteren Bezugnahmen auf das File verwendet. Der Aufbau der Struktur FILE ist im File stdio.h enthalten.

Versucht man, ein existierendes File für Schreiben ("w" oder "w+") zu öffnen, so geht der ursprüngliche Inhalt des Files verloren. Nach dem Öffnen ist das alte File bereits zerstört. Dagegen führt das Öffnen eines noch nicht existierenden Files für Schreiben oder Fortschreiben ("w", "w+", "a", "a+") dazu, dass dieses File erzeugt wird. Tritt jedoch eine Fehlerbedingung auf, z.B. wenn ein nicht existierendes File gelesen werden soll ("r", "r+"), so liefert fopen den Zeigerwert NULL.

Die bisher benutzten Standardfiles werden bereits vor der Programmausführung geöffnet. Ihre Filepointer sind

```
stdin:   Standardeingabe
stdout:  Standardausgabe
stderr:  Standardfehlerausgabe.
```

Die einfachste Möglichkeit, geöffnete Files zu lesen bzw. zu schreiben, stellen die Standardfunktionen getc bzw. putc dar.

```
int getc(file_p)
    FILE *file_p;
```

Die Funktion getc liefert das nächste Zeichen aus dem File file_p, wobei file_p der Filepointer ist, den fopen lieferte. Bei Auftreten eines Fehlers bzw. der Fileendebedingung gibt getc EOF zurück.

Für die Ausgabe existiert die Funktion putc:

```
int putc(c, file_p)
    int c;
    FILE *file_p;
```

Die Funktion schreibt das Zeichen c in das File file_p und liefert als Ergebnis den Wert von c oder EOF im Fehlerfall.

Jetzt können auch die Makros getchar und putchar angegeben werden:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc(c, stdout)
```

Die Filepointer stdin, stdout und stderr können wie andere Zeiger vom Typ FILE * verwendet werden, sie sind jedoch als Konstanten zu betrachten. Oft wird z.B.

```
fprintf(stderr, "error in ...");
```

verwendet, um Fehlernachrichten nicht mit der Standardausgabe zu vermischen.

Mit der Funktion fclose können Files explizit geschlossen werden:

```
int fclose(file_p)
    FILE *file_p;
```

Implizit werden alle mit fopen geöffneten Files beim Beenden der Programmausführung mittels exit (s. Abschn. 3.3.8.) geschlossen.

### 3.3.5. Zeilenweise Ein- und Ausgabe

Zur Unterstützung der zeilenweisen Verarbeitung von Files gibt es für die Eingabe die Funktion fgets.

```
char *fgets(s, n, file_p)
    char *s;
    int n;
    FILE *file_p;
```

Aus dem File file_p wird eine Zeile (einschliesslich Newline) in das Zeichenfeld s übertragen. Am Ende wird noch das Zeichen \Ø angefügt. Die Funktion fgets liest maximal n-1 Zeichen und liefert als Funktionswert den Zeiger auf s oder NULL bei Erreichen des Fileendes.

Die analoge Funktion für die Ausgabe ist mit der Funktion fputs gegeben.

```
int fputs(s, file_p)
    char *s;
    FILE *file_p;
```

Alle Zeichen des Zeichenfeldes s bis zum Auftreten des Zeichens \Ø werden übertragen. Das Newlinezeichen sollte also bereits im Zeichenfeld enthalten sein.

Als Beispiel für die Anwendung verschiedener E/A-Funktionen wird eine weitere Version des Kommandos cat vorgestellt. Sie ermöglicht das Verketten mehrer Eingabefiles zu einem Ausgabefile. Als Aufrufparameter kann man beliebig viele Filenamen angegeben. Diese Files werden von cat nacheinander gelesen und in die Standardausgabe (stdout) ausgegeben. Sind beim Aufruf keine Filenamen angegeben, so wird die Standardeingabe (stdin) gelesen. Das Lesen des Standardeingabefiles kann man auch durch die Angabe des speziellen Aufrufparameters - fordern. Somit ergibt sich folgende Syntax für den Aufruf dieser Version von cat:

```
cat [-] [<file1> ...]
```

Die main-Funktion des Programms ist verantwortlich für die Auswertung der Aufrufparameter, das Öffnen und Schliessen der Eingabefiles sowie für die Ausgabe eventueller Fehlermitteilungen in die Standardfehlerausgabe (stderr).

```
#include  <stdio.h>

#define  MAX  512

main(argc, argv)                    /* cat (Version 2) */
    char *argv[];
{
    register char **argp;
    register FILE *file;

    if(argc == 1)
        copy(stdin);
    else {
        argp = argv;
        while(--argc > Ø)
            if((*++argp)[Ø] == '-' && (*argp)[1] == '\Ø')
                copy(stdin);
            else if((file = fopen(*argp, "r")) != NULL) {
                copy(file);
                fclose(file);
            }
            else
                fprintf(stderr, "%s: can't open %s\n", *argv, *argp);
    }
}
```

Die eigentliche Kopierfunktion lässt sich ganz einfach mittels fgets und fputs realisieren.

```
void copy(file)                     /* Kopieren von file nach stdout */
    FILE *file;
{
    char zf[MAX];

    while((fgets(zf, MAX, file)) != NULL)
        fputs(zf, stdout);
}
```

### 3.3.6. Dynamische Speicherverwaltung

Abhängig von der konkreten Situation ist es oft sinnvoll, während der Laufzeit dynamisch Speicherplatz anzufordern. Da das Zeigerkonzept einen freizügigen Umgang mit Zeigern erlaubt, ist dieses Problem vollständig in Bibliotheksfunktionen lösbar. Das Betriebssystem muss dafür natürlich die elementaren Mittel zur Ausdehnung des Programmbereichs innerhalb der möglichen Grenzen bereitstellen. Diese Grenzen schwanken je nach Rechnertyp sehr stark, von wenigen Kilobytes bis zu mehreren Gigabytes.

Die Funktion malloc

```
#include <malloc.h>

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

liefert einen Zeiger auf einen Speicherbereich der Länge size oder NULL, wenn der geforderte Bereich nicht bereitgestellt werden kann. Obwohl ein Zeiger auf **char** zurückgegeben wird, so genügt der Zeiger doch den Ausrichtungsanforderungen aller Datentypen.

Nicht mehr benötigter Speicherplatz kann mit der Funktion free freigegeben werden.

```
#include <malloc.h>

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

Dabei muss ptr einen vorher mit malloc angeforderten Bereich adressieren. Falsche Zeigerwerte führen zu undefinierten Ergebnissen. Auch das Einhalten der Grenzen des mit malloc bereitgestellten Speicherbereichs liegt in der Verantwortung des Programmierers.

Die Freibereiche werden in einer zyklischen Liste verwaltet, so dass zurückgegebene Bereiche leicht eingekettet und benachbarte Speicherbereiche zusammengelegt werden können.

### 3.3.7. Beispiel: Verwalten von Binärbäumen

Viele Bibliotheksfunktionen benutzen die Möglichkeiten der dynamischen Speicherverwaltung, so auch die Funktionen tsearch und tdelete. Die im Abschn. 2.9.3. angegebene Funktion twalk kann zum Durchmustern von binären Bäumen verwendet werden. In diesem Abschnitt gehen wir ausführlicher auf die Funktionen tsearch, tfind und tdelete ein.

```
#include <search.h>

char *tsearch(key, rootp, compar)
    char *key;
    char **rootp;
    int (*compar)();

char *tfind(key, rootp, compar)
    char *key;
    char **rootp;
    int (*compar)();

char *tdelete(key, rootp, compar)
    char *key;
    char **rootp;
    int (*compar)();
```

Mit den Funktionen tsearch und tfind kann eine solche Baumstruktur aufgebaut bzw. nach der Information *key gesucht werden. Die Funktion tdelete entfernt eine Information aus dem Baum. Die Ordnungsrelation eines konkreten Binärbaums wird durch den Anwender mit der Funktion *compar festgelegt (s. Anhang B).

In den folgenden Implementationen von tsearch, tfind und tdelete kommen wir auf das Problem der dynamischen Speicherplatzverwaltung zurück.

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

static enum flag {
    SEARCH,
    FIND
} tfindflag;

struct node {
    char *key;
    struct node *left;
    struct node *right;
};
```

```
char *tsearch(key, rootp, compar) /* Aufruf von _tsearch: SEARCH */
    char *key;
    char **rootp;
    int (*compar)();
{
    struct node *np, *_tsearch();

    tfindflag = SEARCH;
    return((np = _tsearch(key,rootp,compar))==NULL ? NULL : np->key);
}

char *tfind(key, rootp, compar)   /* Aufruf von _tsearch: FIND */
    char *key;
    char **rootp;
    int (*compar)();
{
    struct node *np, *_tsearch();

    tfindflag = FIND;
    return((np = _tsearch(key,rootp,compar))==NULL ? NULL : np->key);
}
```

Die Funktion tfind ist der Funktion tsearch sehr ähnlich. Sie rufen beide die rekursive Funktion _tsearch auf, die die eigentliche Arbeit erledigt. Der einzige Unterschied besteht darin, dass vor dem Aufruf in der Variablen tfindflag vermerkt wird, ob eine Information in den Binärbaum eingefügt werden soll (SEARCH) oder nicht (FIND).

```
static struct node *_tsearch(key, rootp, compar)      /* Suchen */
    char *key;
    char **rootp;
    int  (*compar)();
{
    register struct node **p;
    int compv;

    if(*(p = (struct node **)rootp) == NULL) {
        if(tfindflag == FIND ||
            (*p = (struct node *)malloc(sizeof(struct node)))==NULL)
            return(NULL);
        (*p)->key = key;
        (*p)->left = (*p)->right = NULL;
    } else if((compv = (*compar)(key, (*p)->key)) < 0) {
        return(_tsearch(key, (char **)&(*p)->left, compar));
    } else if(compv > 0) {
        return(_tsearch(key, (char **)&(*p)->right, compar));
    }
    return(*p);
}
```

Bei jedem Durchlauf der Funktion _tsearch wird jeweils ein Knoten untersucht.

Wird dabei ein leerer Unterbaum getroffen (*rootp == NULL), so ist die Information *key nicht im Baum enthalten. Für tfind ist damit die Arbeit beendet. Erfolgte der ursprüngliche Aufruf jedoch von tsearch, so muss für die Information *key ein neuer Knoten eingerichtet werden. Die Bereitstellung des dafür notwendigen Speicherplatzes erfolgt durch die Funktion malloc. Ist kein Speicher mehr verfügbar, liefert malloc den Zeigerwert NULL.

Für jeden nichtleeren Unterbaum wird die zum Wurzelknoten gehörige Information *(*p)->key mit der gesuchten Information *key verglichen. Ist *key im Sinne der Ordnungsrelation "kleiner" bzw. "grösser", so muss nach

links bzw. rechts absteigend weitergesucht werden. Bei Gleichheit ist die Suche beendet, die Funktion _tsearch gibt den Zeiger des aktuellen Wurzelknotens und tsearch oder tfind den Zeiger auf die gesuchte Information zurück.

Abschliessend zeigen wir die Funktion tdelete, mit der eine Information aus dem Baum entfernt und die zugehörige Knoteninstanz freigegeben wird. Zu diesem Zweck erfolgt der Aufruf von free. Die Funktion tdelete ist etwas komplexer, da der Knoten nicht nur gefunden und beseitigt, sondern der Baum auch geeignet "repariert" werden muss.

```
char *tdelete(key, rootp, compar)       /* Entfernen von Knoten */
    char *key, **rootp;
    int  (*compar)();
{
    register struct node **p, *parent, *np;
    int compv;
    char *mparent();

    p = (struct node **)rootp;
    parent = NULL;
    for(;;) {                                                    /*1*/
        if(*p == NULL)
            return(NULL);                                        /*2*/
        if((compv = (*compar)(key, (*p)->key)) > Ø) {
            parent = *p;
            p = &(*p)->right;
            continue;
        } else if(compv < Ø) {
            parent = *p;
            p = &(*p)->left;
            continue;
        } else
            break;                                               /*3*/
    }
    if((*p)->left == NULL && (*p)->right == NULL)                /*4*/
        return(mparent(parent, p, NULL));
    else if((*p)->left == NULL)                                  /*5*/
        return(mparent(parent, p, (*p)->right));
    else if((*p)->right == NULL)                                 /*6*/
        return(mparent(parent, p, (*p)->left));
    else {                                                       /*7*/
        np = (*p)->right;
        while(np->left != NULL)
            np = np->left;
        np->left = (*p)->left;
        return(mparent(parent, p, (*p)->right));
    }
}
/* Reparieren der Baumstruktur und Freigeben des Knotens */

static char *mparent(parent, p, pn)
    struct node *parent, **p, *pn;
{
    if(parent == NULL) {                                         /*8*/
        free((char *)*p);
        *p = pn;
        return((char *)*p);
    }
    if(parent->left == *p)
        parent->left = pn;
    else
        parent->right = pn;
    free((char *)*p);
    return((char *)parent);
}
```

In der **for**-Schleife erfolgt die Suche nach dem durch key festgelegten Knoten (/*1*/). Die Schleife wird verlassen, wenn tdelete auf einen leeren Knoten trifft, d.h. *key nicht im Baum enthalten ist (/*2*/) oder wenn der Knoten gefunden wurde (/*3*/).

Besitzt der Knoten keine Nachfolger, muss lediglich im Vorgängerknoten der entsprechende Zeiger auf NULL gesetzt und der Knoten selbst beseitigt werden (/*4*/).

Ist nur der rechte bzw. linke Nachfolgerknoten vorhanden, so wird er mit dem Vorgängerknoten verbunden (/*5*/, /*6*/).

Existieren beide Nachfolgerknoten, so erfolgt im rechten Unterbaum links absteigend die Suche nach einem leeren Knoten, der mit dem linken Unterbaum verknüpft wird (/*7*/).

Die Funktion mparent realisiert die Verknüpfung mit dem Vorgängerknoten und das Freigeben einer Knoteninstanz. Ein Sonderfall liegt vor, wenn mittels tdelete der Wurzelknoten **rootp beseitigt werden soll (/*8*/). In diesem Fall ist es erforderlich, den beim Anwender definierten Zeiger auf den Wurzelknoten zu modifizieren. Aus diesem Grund muss an tdelete die Adresse des Zeigers auf den Wurzelknoten des Baums übergeben werden.

### 3.3.8. Fehlerbehandlung

Bibliotheksfunktionen melden über ihren Funktionswert auch eine während der Abarbeitung aufgetretene Fehlerbedingung. Ausserdem wird in der externen **int**-Variablen errno eine Fehlernummer abgespeichert, die den Fehler genauer beschreibt. Diese Fehlernummer kann von der aufrufenden Funktion zur differenzierten Fehlerbehandlung benutzt werden. Die verschiedenen Fehlernummern und ihre Bedeutung sind im File errno.h definiert. Vorteilhaft ist es, die Funktion perror für Fehlermitteilungen zu benutzen.

```
void perror(s)
    char *s;

extern int errno, sys_nerr;
extern char *sys_errlist[];
```

Die Fehlermitteilung s wird in das Standardfile stderr ausgegeben und daran anschliessend eine durch errno festgelegte Systemnachricht. Dabei ist errno als Index im Feld sys_errlist zu betrachten, sys_nerr enthält den Index der letzten in sys_errlist verfügbaren Fehlernachricht. Die Funktion perror sollte nur gerufen werden, wenn tatsächlich ein Fehler aufgetreten ist, da errno im Fehlerfall zwar gesetzt, aber nie gelöscht wird.

Sind schwerwiegende Fehler aufgetreten, so ist oft ein unmittelbarer Programmabbruch erforderlich. Das kann mit der Funktion exit erfolgen.

```
void exit(status)
    int status;
```

Als Argument erwartet exit einen Wert zwischen -127 und +127, mit dem eine Fehlerwichtung erfolgen kann. Der Wert Ø sollte bei ordnungsgemässem Programmende verwendet werden. Die Funktion exit sorgt auch für das Schliessen aller mit fopen geöffneten Files. Ist das nicht erwünscht, kann die Funktion _exit gerufen werden.

```
void _exit(status)
    int status;
```

## 3.4. Systemnahe Programmierung

In diesem Abschnitt wird die Systemschnittstelle des Betriebssystems UNIX aus der Sicht des C-Programmierers behandelt. Dafür gibt es vor allem zwei Gründe: erstens, da diese Schnittstelle für die meisten C-Programmierer relevant ist, und zweitens, um die konkrete Implementation einer E/A-Funktion der Standardbibliothek zu zeigen. Die Systemschnittstelle stellt sich dem Programmierer als ein Satz von **Systemfunktionen** dar, mit denen er die Dienste des Betriebssystems in Anspruch nehmen kann. Neben den Systemfunktionen für das E/A-System gehen wir anhand von Beispielen auch auf Grundlagen der Prozesssteuerung und Signalbehandlung ein. Es wird im weiteren nicht unterschieden, ob es sich um Systemfunktionen handelt, die direkt auf Systemrufe (system calls) abgebildet werden (z.B. open, creat), oder um solche, die zusätzliche Dienste bereitstellen (z.B. execl, execv). Die Kenntnisse über die recht einfache Handhabung der Systemfunktionen helfen, Zusammenhänge klarzustellen. Der Programmierer sollte sich jedoch so lange wie möglich auf die Nutzung der Standardbibliothek beschränken.

### 3.4.1. Zuordnen und Freigeben von Files

Sämtliche Ein- und Ausgabeoperationen erfolgen im Betriebssystem UNIX über eine einheitliche Schnittstelle. So stellen sich dem Programmierer auch periphere Geräte, wie z.B. das Nutzerterminal oder der Drucker, wie Files dar, aus denen gelesen oder in die geschrieben werden kann. Wir können uns also auf die Behandlung von Files beschränken, da das Betriebssystem ein peripheres Gerät als Spezialfile behandelt.

Bevor man ein File lesen bzw. in ein File schreiben kann, muss man diese Absicht dem Betriebssystem mitteilen, d.h., das File muss geöffnet werden. Hierfür existieren die Systemfunktionen open und creat.

```
#include <fcntl.h>

int open(filename, oflag [, mode])
    char *filename;
    int oflag, mode;


#include <sys/types.h>
#include <sys/stat.h>

int creat(filename, mode)
    char *filename;
    int mode;
```

Mit open kann die Verbindung zu bereits existierenden Files hergestellt werden. Dabei wird als erstes Argument der Filename in Form einer Zeichenkette erwartet. Die möglichen Werte für oflag sind im File fcntl.h definiert. 1)

```
#define  O_RDONLY  Ø    /* reading only */
#define  O_WRONLY  1    /* writing only */
#define  O_RDWR    2    /* reading and writing */
```

---

1) Neben den hier angegebenen sind laut /50/ weitere Werte für oflag erlaubt. Nur in diesem Zusammenhang ist der Parameter mode von Bedeutung.

Die Funktion creat ermöglicht das Erzeugen und Öffnen nicht existierender Files. Hierbei gibt mode die Zugriffsrechte für das File an. Die Rechte zum Lesen (r), Schreiben (w) und Ausführen (x) für den Eigentümer, die Gruppe und für andere Nutzer sind in insgesamt 9 Bits verschlüsselt (s. Bild 3.4).

| Eigentümer | | | Gruppe | | | andere | | |
|---|---|---|---|---|---|---|---|---|
| r | w | x | r | w | x | r | w | x |

Bild 3.4. Zugriffsrechte für ein File

Der Schutzmodus lässt sich mit den im File sys/stat.h definierten Konstanten darstellen.

```
#define S_IRUSR Ø4ØØ    /* read by owner */
#define S_IWUSR Ø2ØØ    /* write by owner */
#define S_IXUSR Ø1ØØ    /* execute (search if a directory) by owner */
#define S_IRGRP ØØ4Ø    /* read by group */
#define S_IWGRP ØØ2Ø    /* write by group */
#define S_IXGRP ØØ1Ø    /* execute (search) by group */
#define S_IROTH ØØØ4    /* read by others */
#define S_IWOTH ØØØ2    /* write by others */
#define S_IXOTH ØØØ1    /* execute (search) by others */
```

So gibt z.B.

```
S_IRUSR | S_IWUSR | S_IRGRP
```

an, dass der Eigentümer lesen (S_IRUSR) und schreiben (S_IWUSR) und die zur Gruppe gehörenden Nutzer lesen (S_IRGRP) dürfen, aber alle anderen keinen Zugriff auf das File haben. Wird creat für ein bereits existierendes File gerufen, ist das kein Fehler. Der Bytezähler wird auf Null gesetzt, die Zugriffsrechte werden jedoch nicht verändert. Die Systemfunktion creat bewirkt immer ein Öffnen für Schreiben. Konnte die Verbindung zum angegebenen File hergestellt werden, so liefern die Systemfunktionen creat und open eine positive Integerzahl, den sog. **Filedeskriptor**. Der Filedeskriptor wird im weiteren bei der Bezugnahme auf das gewünschte File verwendet. Erhält ein Nutzerprogramm die Steuerung, so sind die Filedeskriptoren Ø, 1 und 2 bereits zugeordnet, und zwar standardmässig dem Nutzerterminal. Diese Filedeskriptoren entsprechen den Filepointern stdin, stdout und stderr.

Im File stdio.h ist der Strukturtyp _iobuf deklariert, in dem Informationen über die Pufferverwaltung, den Zustand des Files und der zugehörige Filedeskriptor enthalten sind.

```
struct _iobuf  {
    char *_ptr;          /* next character position */
    int _cnt;            /* number of characters left */
    char *_base;         /* location of buffer */
    char _flag;          /* mode of file access */
    char _file;          /* file descriptor */
};
```

Die Bibliotheksfunktionen und Makros für die Ein- und Ausgabe verwalten ein Feld _iob, dessen Elemente Instanzen des Strukturtyps _iobuf sind. Der im Abschn. 3.3.4. eingeführte Filepointer stellt nun genau einen Zeiger auf ein solches Feldelement dar. Die Initialisierung von _iob verdeutlicht diesen Zusammenhang. Der vollständige Aufbau der Struktur _iobuf ist im Abschn. 3.4.3. angegeben. Hier soll nur gezeigt werden, dass neben anderen Informationen als letztes Element der Filedeskriptor enthalten ist.

```
#include <stdio.h>

char _sibuf[BUFSIZ];

struct _iobuf _iob[_NFILE] = {
    { _sibuf, Ø, _sibuf, _IOREAD, Ø };          /* Filedeskriptor Ø */
    { NULL, Ø, NULL, _IOWRT, 1 };               /* Filedeskriptor 1 */
    { NULL, Ø, NULL, _IOWRT+_IONBF, 2 };        /* Filedeskriptor 2 */
};

#define stdin (&_iob[Ø])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

Die Konstante _NFILE entspricht der Anzahl der Files, die gleichzeitig geöffnet sein können (i.allg. 2Ø). Die Verbindung zu nicht mehr benötigten Files sollte explizit aufgelöst werden. Für diesen Zweck steht die Systemfunktion close zur Verfügung.

```
int close(fd)
      int fd;
```

Die Funktion **close** schliesst das File und macht den Filedeskriptor fd wieder verfügbar. Implizit werden alle geöffneten Files beim Beenden des Programms durch das System geschlossen.

Die Systemfunktion unlink kann verwendet werden, um ein File aus dem Filesystem zu beseitigen.

```
int unlink(filename)
    char *filename;
```

Ist filename zum Zeitpunkt des Aufrufs von unlink geöffnet, so wird lediglich die Absicht vermerkt, dass dieses File beim Schliessen beseitigt werden soll. Diese Eigenschaft kann man beim Öffnen temporär benötigter Files ausnutzen.

```
int fd;
...
fd = open("tempfile", O_RDWR);
unlink("tempfile");
```

Unmittelbar nach dem Öffnen von tempfile erfolgt der Aufruf von unlink für dieses File. Jetzt kann das Programm ohne weiteren Aufwand an beliebigen Stellen mit exit verlassen werden, da mit dem impliziten Schliessen auch das File "tempfile" beseitigt wird.

### 3.4.2. Lesen, Schreiben und Positionieren

Nach dem Öffnen soll ein File natürlich auch gelesen bzw. geschrieben werden. Zunächst betrachten wir die rein sequentielle Arbeitsweise, die

aber für die meisten Aufgaben völlig ausreicht.

```
int read(fd, buffer, n)
    int fd;
    char *buffer;
    unsigned n;

int write(fd, buffer, n)
    int fd;
    char *buffer;
    unsigned n;
```

Bei beiden Funktionen stellt der Parameter fd den durch open bzw. creat bereitgestellten Filedeskriptor dar und buffer ist die Adresse eines Feldes, aus dem bzw. in das n Bytes übertragen werden. Der Bezugspunkt ist dabei immer die aktuelle Byteposition im File, die vom System verwaltet wird und die angibt, ab welcher Position im File die nächste Ein- oder Ausgabeoperation erfolgen soll.

Die Systemfunktionen read und write liefern als Funktionswert die Anzahl der tatsächlich übertragenen Bytes. Bei read bedeutet der Wert Null das Ende des Files. Bei write liegt ein Fehler vor, wenn der Funktionswert nicht mit der angegebenen Bytezahl n übereinstimmt.

Mit der Systemfunktion lseek kann die Byteposition für die nächste Ein- oder Ausgabeoperation verändert werden.

```
#include <unistd.h>

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

Die Byteposition des Files mit dem Filedeskriptor fd wird auf die Position offset relativ zu origin gesetzt. Für origin können die im File unistd.h definierten Werte angegeben werden.

```
#define SEEK_SET  0     /* next position: file begin + offset */
#define SEEK_CUR  1     /* next position: current position + offset */
#define SEEK_END  2     /* next position: file end + offset */
```

Somit ergibt sich die Position für die nächste Ein- oder Ausgabeoperation aus dem Wert von offset bezogen auf den Anfang des Files (SEEK_SET), die aktuelle Position (SEEK_CUR) bzw. das Fileende (SEEK_END).

Als Beispiel für den direkten Filezugriff geben wir die Funktionen pushf und popf zur einfachen Kellerspeicherverwaltung an.

```
int popf(fd)                /* Lesen eines int-Wertes aus dem Keller */
    int fd;
{
    int val;

    if(lseek (fd, (long)(-sizeof (int)), SEEK_CUR) == -1) {
        fprintf(stderr, "empty stack\n");
        exit(1);
    }
    read(fd, &val, sizeof (int));
    lseek(fd, (long)(-sizeof (int)), SEEK_CUR);
    return(val);
}
```

```
void pushf(fd, val) /* Speichern eines int-Wertes in den Keller */
    int fd, val;
{
    write(fd, &val, sizeof (int));
}
```

In der Funktion pushf ist natürlich kein explizites Positionieren erforderlich, da dies durch die Systemfunktion write realisiert wird. Bei popf muss hingegen zweimal positioniert werden. Die Systemfunktion lseek liefert normalerweise die Position im File und im Fehlerfall -1. Das ist im Beispiel der Fall, wenn im "File-Kellerspeicher" kein Element mehr vorhanden ist.

### 3.4.3. Beispiel: Implementation von fopen

Man kann die Funktionen der Standardbibliothek problemlos ohne Kenntnis der konkreten Implementation benutzen. Das File stdio.h muss eingeschlossen und die Arbeit mit Files über Filepointer realisiert werden. Diese Filepointer zeigen auf Strukturen vom Typ FILE. Das reicht aus. Es ist jedoch hilfreich, einmal hinter die Kulissen zu sehen. Man betrachte dazu das File stdio.h (Auszug):

```
#define BUFSIZ 512
#define _NFILE 20

extern struct _iobuf  {
    char *_ptr;
    int _cnt;
    char *_base;
    char _flag;
    char _file;
} _iob[_NFILE];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])

#define _IOREAD    01          /* Lesen */
#define _IOWRT     02          /* Schreiben */
#define _IONBF     04          /* ungepuffert */
#define _IOMYBUF   010         /* Puffer angelegt */
#define _IOEOF     020         /* Fileende */
#define _IOERR     040         /* Fehler */
#define _IOSTRG    0100
#define _IORW      0200

#define NULL 0
#define EOF  (-1)
#define FILE struct _iobuf;

#define getc(p)  ( (!((p)->_flag & _IOWRT) && (--(p)->_cnt >= 0)) ? \
                   (*(p)->_ptr++) & 0377  :                        \
                   _filbuf(p)                                      \
                 )

#define putc(x,p)                                                  \
                 ( (!((p)->_flag & _IOREAD) && (--(p)->_cnt >= 0)) ? \
                   ((int)(*(p)->_ptr++ = (unsigned)(x)))  :        \
                   _flsbuf((unsigned)(x), p)                       \
                 )
```

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

#define putchar(x) putc(x, stdout)

#define feof(p)    (((p)->_flag & _IOEOF) != Ø)

#define ferror(p)  (((p)->_flag & _IOERR) != Ø)

#define fileno(p)  p->_file

extern FILE *fopen();
extern FILE *freopen();
extern FILE *fdopen();
extern FILE *popen();
extern long ftell();
extern char *fgets();
```

Wie man sieht, beginnen die Bezeichner, die der Anwender normalerweise nicht zu kennen braucht, mit dem Unterstreichungszeichen. Auf diese Weise werden Namenskonflikte vermieden.

Interessant sind in diesem Zusammenhang auch die Makros getc und putc. Die Systembelastung, die bei der zeichenweisen Verarbeitung durch die Funktionsrufe entstehen würde, wird bei gepufferter Verarbeitung reduziert. Das bei getc dem Puffer entnommene Zeichen wird mit Ø377 maskiert, um eine maschinenabhängige Vorzeichenausdehnung zu verhindern.

Damit kann der Quelltext der Bibliotheksfunktion fopen angegeben werden.

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <errno.h>
#include  <unistd.h>
#include  <fcntl.h>

FILE *fopen(file_name, type)      /* fopen: Öffnen von Files */
    char *file_name;
    char *type;
{
    FILE *_findiop();
    FILE *_endopen();

    return(_endopen(file_name, type, _findiop()));
}
```

Die eigentliche Arbeit erledigt _endopen. In fopen selbst wird durch die Funktion _findiop ein freier Eintrag in dem Feld _iob ermittelt und als Argument an _endopen übergeben.

```
FILE *_findiop()                  /* Suchen eines freien Eintrags */
{
    extern FILE *_lastbuf;
    register FILE *iop;

    for(iop = _iob; iop->_flag & (_IOREAD | _IOWRT | _IORW); iop++) {
        if(iop >= _lastbuf)
            return(NULL);
    }
    return(iop);
}
```

```
FILE *_endopen(file_name, type, iop)  /* Initialisieren _iob-Entry */
    char *file_name, *type;
    register FILE *iop;
{
    extern int errno;
    register int rw, f;

    if(iop == NULL)
        return(NULL);
    rw = type[1] == '+';
    switch(*type)  {
        case 'w':
            f = create(file_name, rw);
            break;
        case 'a':
            if((f = open(file_name, rw ? O_RDWR : O_WRONLY)) < Ø)
                if(errno == ENOENT)
                    f = create(file_name, rw);
            lseek(f, ØL, SEEK_END);
            break;
        case 'r':
            f = open(file_name, rw ? O_RDWR : O_RDONLY);
            break;
        default:
            return(NULL);
    }
    if(f < Ø)
        return(NULL);
    iop->_cnt = Ø;
    iop->_file = f;
    if(rw)
        iop->_flag |= _IORW;
    else if(*type == 'r')
        iop->_flag |= _IOREAD;
    else
        iop->_flag |= _IOWRT;
    return(iop);
}
```

Für die Zugriffsmodi "w" (Schreiben) und "w+" (Schreiben und Lesen) wird die Funktion create verwendet, um ein nicht existierendes File zu erzeugen oder ein bereits vorhandenes File zu löschen. Der Aufruf von create erfolgt auch für die Modi "a" (Fortschreiben) und "a+" (Fortschreiben und Lesen), wenn das betreffende File noch nicht existiert. Neben dem Öffnen und eventuellen Erzeugen des Files wird die Struktur _iobuf initialisiert, jedoch noch kein Puffer zugeordnet.

Die Funktion create ist für den Anwender von fopen nicht sichtbar, deshalb kann es zu keinen Namenskonflikten kommen.

```
static int create(file_name, rw)      /* Erzeugen eines Files */
    register char *file_name;
    int rw;
{
    register int f;

    f = creat(file_name,  S_IRUSR | S_IWUSR
                        | S_IRGRP | S_IWGRP
                        | S_IROTH | S_IWOTH );
    if(rw && f >= Ø)  {
        close(f);
        f = open(file_name, O_RDWR);
    }
    return(f);
}
```

### 3.4.4. Verzeichnisfiles

Ein **Verzeichnisfile** (directory) besteht aus einer Menge von Verzeichniseinträgen. Verzeichnisfiles können wie andere Files gelesen, jedoch nicht geschrieben werden. Um die Konsistenz von Verzeichnisfiles zu sichern, muss das Schreiben speziellen Systemkomponenten vorbehalten bleiben. Jeder Eintrag enthält den Namen eines Files sowie die Nummer des dazugehörigen Inode-Eintrags (inode), der das File beschreibt. Zu den darin enthaltenen Informationen gehören u.a.

- der Filetyp und die Zugriffsrechte
- die Grösse des Files in Bytes
- Datum und Uhrzeit der letzten Zustandsänderung, der letzten Modifikation und des letzten Zugriffs
- die physischen Blockadressen des Files.

Der Aufbau eines Verzeichniseintrags ist in der Struktur direct deklariert, die im File sys/dir.h enthalten ist.

```
#define DIRSIZ 14                   /* Länge des Filenamens (max.) */
struct direct  {
    ino_t d_ino;                    /* Inode-Nummer */
    char d_name[DIRSIZ];            /* Filename */
};
```

Als Beispiel folgt die Funktion next_entry, die für ein geöffnetes Verzeichnisfile jeweils den nächsten gültigen Eintrag bereitstellt.

```
#include <sys/types.h>
#include <sys/dir.h>
#include <string.h>

int next_entry(fd, fname)     /* nächsten Verzeichniseintrag lesen */
    int fd;
    char *fname;
{
    struct direct dirbuf;
    register struct direct *pdir = &dirbuf;

    while(sizeof(dirbuf) == read(fd, (char *) pdir, sizeof(dirbuf)) {
        if(pdir->d_ino == (ino_t) 0)   /* leerer Eintrag */
            continue;
        if(   strcmp(pdir->d_name, ".") != 0
           && strcmp(pdir->d_name, "..") != 0)  {
            strcpy(fname, pdir->d_name);
            return(0);
        }
    }
    return(-1);
}
```

Leere Verzeichniseinträge entstehen durch das Löschen von Files mittels unlink, wobei lediglich die Inode-Nummer d_ino des zugehörigen Eintrags auf Null gesetzt wird. In jedem Verzeichnisfile existieren zwei Einträge mit den Filenamen "." bzw. "..", die auf das Verzeichnisfile selbst bzw. auf das übergeordnete Verzeichnisfile verweisen. Diese Einträge werden in next_entry übergangen.

Das File sys/types.h enthält eine Reihe wichtiger Typdefinitionen, u.a. die Definition des Typs einer Inode-Nummer:

```
typedef unsigned int ino_t;
```

Über die Systemfunktionen stat bzw. fstat kann man sich die Inode-Informationen eines Files bereitstellen lassen. Die Bezugnahme auf den entsprechenden Inode-Eintrag erfolgt bei stat über den Namen des Files. Bei einem geöffneten File kann fstat verwendet werden, wobei der Filedeskriptor anzugeben ist.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(filename, stat_p)
    char *fname;
    struct stat *stat_p;

int fstat(fd, stat_p)
    int fd;
    struct stat *stat_p;
```

Im File sys/stat.h ist die Strukturdeklaration von stat enthalten.

```
struct stat  {
    dev_t st_dev;          /* Gerät */
    ino_t st_ino;          /* Inode-Nummer */
    short st_mode;         /* Statusbits */
    short st_nlink;
    short st_uid;          /* Nutzernummer */
    short st_gid;          /* Gruppennummer */
    dev_t st_rdev;
    off_t st_size;         /* Filegrösse in Bytes */
    time_t st_atime        /* Zeitpunkt des letzten Zugriffs */
    time_t st_mtime        /* Zeitpunkt der letzten Modifikation */
    time_t st_ctime;       /* Zeitpunkt der letzten Zustandsänderung */
};
```

Um die Auswertung der Statusbits zu erleichtern, sind in diesem File symbolische Konstanten definiert, z.B.

```
#define    S_IFMT    Ø17ØØØØ     /* Filetyp */
#define    S_IFDIR   ØØ4ØØØØ     /* Verzeichnis */
#define    S_IFCHR   ØØ2ØØØØ     /* Spezialfile (Zeichenprotokoll) */
#define    S_IFBLK   ØØ6ØØØØ     /* Spezialfile (Blockprotokoll) */
#define    S_IFREG   Ø1ØØØØØ     /* reguläres File */
```

Die Deklarationen für die Typbezeichner dev_t, off_t und time_t sind im File sys/types.h enthalten:

```
typedef int dev_t;
typedef long off_t;
typedef long time_t;
```

Die Auswertung von Inode-Informationen wird im Abschn. 3.5.1. angewendet.

### 3.4.5. Grundelemente der Prozeßsteuerung

Unter einem Prozess sei hier die Ausführung eines Programms in seiner Umgebung verstanden. Zur Umgebung eines Prozesses gehören dessen aktuelle Speicherbelegung, d.h. der ausführbare Programmtext und die zugehörigen Daten (Programmumgebung) sowie weitere Informationen, wie der Zustand geöffneter Files (Systemumgebung). Die wichtigsten Systemfunktionen zur Pro-

zesssteuerung sind fork, execl, execv, wait und exit. Die bereits vorgestellte Funktion exit bewirkt die Beendigung eines Prozesses und die Rückgabe von Statusinformationen an den Elternprozess (parent process).

Die Systemfunktionen execl und execv werden benutzt, um die aktuelle Programmumgebung durch ein neues Programm zu überlagern. Die Systemumgebung des laufenden Prozesses bleibt dabei erhalten, es wird kein neuer Prozess eingerichtet.

```
int execl(filename, arg0, arg1, ..., argn, (char *)0)
    char *filename, *arg0, *arg1, ..., *argn;

int execv(filename, argv)
    char *filename, *argv[];
```

Dabei gibt die Zeichenkette filename den Namen des auszuführenden Files an. Die weiteren Argumente von execl sind ebenfalls Zeichenketten und entsprechen den einzelnen Parametern $par_0$, $par_1$, ..., $par_n$ beim Aufruf eines C-Programms (vgl. Abschn. 3.1.). Abgeschlossen wird die Argumentliste mit dem Zeigerwert NULL. Beispielsweise bewirkt

```
execl("/bin/cat", "cat", "-n", "-v", (char *)0);
```

das Laden und Ausführen des Kommandos cat. Die Abarbeitung des rufenden Programms wird damit beendet. Es erhält die Steuerung nur dann zurück, wenn execl nicht erfolgreich durchgeführt werden konnte.

Die Systemfunktion execv unterscheidet sich von execl nur hinsichtlich der zu übergebenden Argumente. Sie wird dann verwendet, wenn die Anzahl der Argumente im voraus nicht bekannt ist. Die Bezeichnung argv deutet an, dass die zu übergebenden Argumentwerte genauso aufbereitet werden müssen, wie es die main-Funktion des neuen Programms erwartet. Demzufolge ist argv die Adresse eines Feldes von Zeichenzeigern, wobei jedes Feldelement die Adresse einer Zeichenkette darstellen muss:

```
static char *argv[] = {
                          "cat",
                          "-n",
                          "-v",
                          (char *)0
                      };
...
execv("/bin/cat", argv);
```

Möchte man ein anderes Programm ausführen, seine eigene Arbeit jedoch nicht beenden, so muss mittels fork ein neuer Prozess, der sog. Kindprozess (child process), eingerichtet werden.

```
int fork()
```

Tatsächlich dupliziert fork lediglich den aktuellen Prozess. Beide Prozesse können dann separat weiterlaufen, wobei im Kindprozess gewöhnlich mit execl oder execv ein anderes Programm ausgeführt wird. Der Elternprozess kann mit wait auf das Ende des Kindprozesses warten und dessen exit-Status auswerten.

```
int wait(status_adr)
    int *status_adr;

int wait((int *)0)
```

Ihre Identität können die Prozesse am Funktionswert von fork erkennen, wie folgende Anweisungen verdeutlichen:

```
int status;
int pid;

if((pid = fork()) == Ø) {
    ...                         /* Kindprozess */
    execl(...)
    exit(127);
}
wait(&status);                  /* Elternprozess */
...                             /* pid = Prozessnummer des Kindprozesses */
```

Der Elternprozess erhält als Ergebnis der fork-Funktion einen positiven Integerwert, die Prozessnummer des Kindprozesses. Der Kindprozess ist am Funktionswert Ø zu erkennen. Durch execl erfolgt die Überlagerung der Programmumgebung des Kindprozesses. Demzufolge wird der nachstehende Aufruf von exit nur dann durchgeführt, wenn execl nicht erfolgreich war, d.h. das neue Programm nicht geladen werden konnte. In einem solchen Fall wird der Kindprozess mit dem exit-Status 127 beendet.

Die einfachste Möglichkeit, ein anderes Programm auszuführen und auf die Beendigung zu warten, bietet die Standardbibliotheksfunktion system.

```
#include <stdio.h>

int system(cmd_line)
    char *cmd_line;
```

Als Argument erwartet system eine Zeichenkette, deren Aufbau einer am Terminal eingegebenen Kommandozeile entspricht. Zum Beispiel bewirkt

```
system("cc -o wurzel wurzel.c");
```

die Compilierung des Programms wurzel.c durch den C-Compiler.

Um die Anwendung der vorgestellten Systemfunktionen _exit, execl, fork und wait an einem Beispiel zu zeigen, folgt eine vereinfachte Version von system:

```
int system(cmd_line)                /* Ausführen eines Kommandos */
    char *cmd_line;
{
    int status, pid, w;

    if((pid = fork()) == Ø) {
        execl("/bin/sh", "sh", "-c", cmd_line, (char *)Ø);
        _exit(127);
    }
    while((w = wait(&status)) != pid && w != -1)
        ;
    if(w == -1)
        status = -1;
    return(status);
}
```

Mit execl wird hier der Kommandointerpreter (**shell**) aufgerufen, wobei -c angibt, dass die Kommandozeile cmd_line auszuführen ist. Die Funktion wait liefert als Wert die Nummer des beendeten Kindprozesses. Das ist notwendig, da der Elternprozess mehrere Kindprozesse erzeugen kann. In bestimmten Situationen (s. Abschn. 3.4.6.) liefert wait den Funktionswert -1.

### 3.4.6. Signalbehandlung

Oft ist es notwendig, dass Programme auf bestimmte asynchrone Ereignisse reagieren können. Das dabei wohl interessanteste Ereignis ist ein vom Nutzer am Terminal ausgelöstes Interrupt-Signal. Die Standardreaktion bewirkt den Abbruch des laufenden Programms. Manchmal sind vor dem Programmabbruch jedoch gewisse Sicherungsmassnahmen erforderlich, z.B. das Löschen temporärer Files. Bei Dialogprogrammen möchte man meist nur eine Teilaktion abbrechen und in einen interaktiven Modus zurückkehren. In kritischen Programmteilen kann auch das Ignorieren solcher Ereignisse erforderlich sein.

Zur Steuerung der Signalbehandlung gibt es die Systemfunktion signal.

```
#include <signal.h>

int (*signal(sig, func))()
    int sig;
    int (*func)();
```

Beim Aufruf dieser Funktion klassifiziert das erste Argument den Signaltyp. Als zweites Argument gibt man die Adresse einer Funktion an, zu der bei Eintreten des angegebenen Signals verzweigt werden soll. Die speziellen Argumentwerte SIG_IGN bzw. SIG_DFL bewirken das Ignorieren des betreffenden Signals bzw. den Abbruch des Programms (Standardreaktion). Beide symbolischen Konstanten sind im File signal.h definiert.

Als Funktionswert liefert signal die Adresse der zum Zeitpunkt des Aufrufs für den angegebenen Signaltyp aktiven Behandlungsroutine.

Das File signal.h enthält auch die Konstantendefinitionen für die einzelnen Signaltypen:

```
#define SIGHUP  1           /* hangup */
#define SIGINT  2           /* interrupt */
#define SIGQUIT 3           /* quit */
#define SIGILL  4           /* illegal instruction */
#define SIGTRAP 5           /* trap */
#define SIGABRT 6           /* process abort */
    ...
```

Möchte man z.B. ein Interrupt-Signal ignorieren, so schreibt man einfach:

```
signal(SIGINT, SIG_IGN);
```

Um für das Interrupt-Signal die Standardreaktion wieder einzuschalten, ist folgender Aufruf von signal erforderlich:

```
signal(SIGINT, SIG_DFL);
```

Danach kann das Programm vom Nutzer also wieder durch Betätigen der Interrupt-Taste abgebrochen werden.

Wenn man Signale selbst behandeln möchte, so muss man als zweites Argument die Adresse einer vom Nutzer bereitzustellenden Funktion angeben:

```
int int_handler();
...
signal(SIGINT, int_handler);
```

Diese Funktion erhält beim Eintreten des Signals die Steuerung, wobei als Argument der Signaltyp übergeben wird.

```
#include <signal.h>

void int_handler(sig_nr)          /* Interruptbehandlungsfunktion */
    int sig_nr;
{
    extern int critical;
    extern int interrupt;

    signal(sig_nr, int_handler);
    ...
    if(critical != Ø) {
        interrupt = 1;
        return;
    }
    ...
    exit(1);
}
```

Bei den meisten Signaltypen wird nach dem Auftreten automatisch wieder die Standardreaktion in Kraft gesetzt. Wünscht man das nicht, so muss man die Funktion signal erneut aufrufen. Wird die Behandlungsfunktion über **return** verlassen, so erfolgt eine Weiterarbeit an der unterbrochenen Stelle. Im obigen Beispiel wurde angedeutet, dass z.B. in kritischen Programmabschnitten (critical != Ø) lediglich eine externe Variable gesetzt und das Programm normal fortgesetzt wird. Das Signal könnte verzögert behandelt werden. Voraussetzung dafür ist, dass der Programmierer die kritischen Programmteile durch Setzen bzw. Löschen der Variablen critical "klammert" und an geeigneter Stelle die Variable interrupt testet.

Einige Systemfunktionen können ihre Arbeit jedoch nicht problemlos fortsetzen und liefern den Funktionswert -1.

Im Abschn. 3.5.5. wird in einem Beispiel die Möglichkeit gezeigt, wie man aus der Behandlungsroutine an eine vorher definierte Stelle verzweigen kann.

Abschliessend stellen wir die vollständige Implementierung der Funktion system vor.

```
#include <signal.h>

int system(cmd_line)              /* Ausführen eines Kommandos */
    char *cmd_line;
{
    int status;
    int pid;
    int w;
    register int (*istat)();
    register int (*qstat)();

    if((pid = fork()) == Ø) {
        execl("/bin/sh", "sh", "-c", cmd_line, (char *)Ø);
        _exit(127);
    }
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while((w = wait(&status)) != pid && w != -1)
        ;
    if(w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}
```

Interessant ist hier die Vereinbarung der Variablen istat und qstat, die als Zeiger auf Funktionen vom Typ int definiert sind. Da es sich um Zeiger handelt, können sie durchaus in Registern untergebracht werden.

Die system-Funktion im Absohn. 3.4.5. kann zu Problemen führen, wenn in der Zeit des Wartens (wait) auf die Beendigung des Kindprozesses ein Interrupt-Signal auftritt. Um eine von der system-Funktion unkontrollierbare Weiterarbeit des Elternprozesses zu verhindern, erfolgt das Ignorieren von Interruptsignalen. Die vorhergehende Signalbehandlung wird gerettet und nach der Beendigung des Kindprozesses wieder eingeschaltet.

## 3.5. Programmbeispiele

Im letzten Abschnitt werden noch einmal Beispielprogramme vorgestellt, die verschiedene Aspekte der Programmierung in der Sprache C verdeutlichen. Wenngleich die Analyse von Quellprogrammen im Prozess des Erlernens einer Sprache kein Ersatz für eigene praktische Programmierübungen sein kann, so ist das Studieren "echter" C-Programme trotzdem sehr wertvoll. Es trägt sowohl zum besseren Verständnis der Sprache und zur Beherrschung der sprachlichen Mittel bei und fördert zudem die Entwicklung eines "Programmierstils".

Das Programm cat wurde bereits in verschiedenen Varianten vorgestellt. Hier geben wir eine abschliessende Version an, die weitgehend dem gleichnamigen UNIX-Kommando entspricht. Ausserdem soll anhand verschiedener Problemstellungen gezeigt werden, wie man auf recht einfache Weise Basiswerkzeuge in der Sprache C entwickeln kann.

### 3.5.1. Verketten von Files

Wir betrachten ein Programm cat, das mehrere Files nacheinander liest und in die Standardausgabe ausgibt, wobei wahlweise vor jede Ausgabezeile eine Zeilennummer gesetzt wird (-n) bzw. alle nichtdruckbaren Zeichen als Oktaldarstellung der Form \nnn erscheinen (-v). Ausserdem kann die ungepufferte Ausgabe gefordert werden (-u). Damit ergibt sich folgende Kommandosyntax:

```
cat [-n][-v][-u] [-] [<file1> ...]
```

Es können mehrere Flags zusammengefasst werden, z.B. bedeutet

```
cat -nv
```

dass sowohl die Zeilennumerierung erfolgen soll als auch die Umwandlung nichtdruckbarer Zeichen (vgl. Abschn. 3.1.). Ist kein Eingabefile angegeben, so wird von der Standardeingabe gelesen. Das gleiche bewirkt der Aufrufparameter - (ohne nachfolgenden Buchstaben).

Die main-Funktion des Programms cat ist hauptsächlich für die Auswertung der Aufrufargumente sowie für das Öffnen bzw. Schliessen der Files verantwortlich.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>

char **argp;
char **argØ;
char bufout[BUFSIZ];
short nanz, vanz, uanz, status;
short stin = Ø;
struct stat fs;
dev_t geraet = Ø;
ino_t inode = Ø;

main(argc, argv)                        /* cat (Version 3) */
    char **argv;
{
    register int opt;
    register FILE *file;
    extern int optind;

    argØ = argv;
    while((opt = getopt(argc, argv, "vnu ")) != EOF)
            switch(opt) {
                case 'n':
                    nanz++;
                    break;
                case 'v':
                    vanz++;
                    break;
                case 'u':
                    uanz++;
                    break;
                case ' ':
                    stin++;
                    break;
                default :
                    exit(1);
            }
    if(uanz)
        setbuf(stdout, (char *) NULL);
    else
        setbuf(stdout, bufout);
    setstat();
    argc -= optind;
    if(argc == Ø)
        copy(stdin);
    else {
        argp = &argv[optind - 1];
        if(stin)
            copy(stdin);
        while(argc-- > Ø)
            if((*++argp)[Ø] == '-' && (*argp)[1] == '\Ø')
                copy(stdin);
            else if((file = fopen(*argp, "r")) != NULL) {
                copy(file);
                fclose(file);
            }
            else
                fprintf(stderr, "%s - cannot open: %s\n",
                                                *argØ, *argp);
    }
    exit(Ø);
}
```

Der Aufruf der Bibliotheksfunktion setbuf mit dem Zeigerwert NULL (als Pufferadresse) hat zur Folge, dass die Ausgabe ungepuffert erfolgt. Ansonsten wird bufout als Ausgabepuffer benutzt. Die eigentliche Arbeit erledigt die Funktion copy.

```
#define  MAX 512

void copy(file)                    /* Kopieren von file nach stdout */
    FILE *file;
{
    register char *pz;
    long nummer = Ø;
    char zf[MAX];
    register int nlanz;

    if(cmpstat(file))
        return;
    nlanz = 1;
    while((fgets(zf, MAX, file)) != NULL) {
        if(nanz && nlanz)
            printf("%6ld  ", ++nummer);
        nlanz = strchr(zf, '\n') == NULL ? Ø : 1;
        for(pz = zf; *pz; pz++)
            if(vanz && !isprint(*pz) && !isspace(*pz))
                printf("\\%.3o", *pz);
            else
                putchar(*pz);
    }
}
```

Das Problem besteht offensichtlich darin, die Zeilennummer jeweils an der richtigen Stelle einzufügen. Zu diesem Zweck wird die Variable nlanz definiert. Sie erhält den Wert 1 immer dann, wenn in der Eingabe ein Newline aufgetreten ist, und zeigt somit an, dass als nächstes die aktuelle Zeilennummer ausgegeben werden muss.

Kommen wir nun zu den Funktionen setstat bzw. cmpstat, die durch die main-Funktion bzw. von copy aufgerufen werden. Betrachten wir deshalb folgenden Aufruf von cat, bei dem ein File sowohl Eingabe- als auch Ausgabefile ist:

```
cat  file_a  file_b  >file_a
```

Der Fileumlenkungsoperator > bewirkt das Zerstören des aktuellen Inhalts von file_a. Da die Fileumlenkung durch den UNIX-Kommandointerpreter, d.h. vor der Übergabe der Steuerung an das Kommando cat erfolgt, ist dieses File zum Zeitpunkt des Öffnens durch fopen in der main-Funktion bereits leer. Demzufolge werden nur die Daten des Files file_b verarbeitet. Noch problematischer ist jedoch die folgende Situation:

```
cat  file_a  file_b  >file_b
```

In diesem Fall wird file_b zwar vor dem Aufruf von cat gelöscht, andererseits enthält file_b die sich aus der Verarbeitung von file_a ergebenden Daten, wenn es für die Eingabe geöffnet wird. Dadurch entsteht die Gefahr eines unendlichen Lese-Schreib-Zyklus, der cat folgendermassen begegnet. Zunächst wird für jedes Eingabefile setstat gerufen, um die Geräte- und die Inode-Nummer festzustellen. Die Funktion cmpstat vergleicht diese beiden Werte mit den entsprechenden Werten des Ausgabefiles. Stimmen sowohl die Geräte- als auch die Inode-Nummer überein, so handelt es sich um das

gleiche File. Es erfolgt die Ausgabe einer Fehlernachricht und das Übergehen dieses Eingabefiles.

```
void setstat()                  /* Setzen des Ausgabefilestatus */
{
    fstat(fileno(stdout), &fs);
    status = fs.st_mode & S_IFMT;
    if(status != S_IFCHR && status != S_IFBLK) {
        geraet = fs.st_dev;
        inode = fs.st_ino;
    }
}

int cmpstat(file)               /* Statusvergleich: Eingabe/Ausgabe */
    FILE *file;
{
    fstat(fileno(file), &fs);
    if(fs.st_dev == geraet && fs.st_ino == inode) {
        fprintf(stderr, "%s - input is output: %s\n",
                        *argØ, file == stdin ? "-" : *argp);
        return(1);
    }
    return(Ø);
}
```

### 3.5.2. Verwalten einer Datenbasis in binären Bäumen

In diesem Abschnitt behandeln wir eine Anwendung der bereits vorgestellten Funktionen tsearch, tdelete, twalk und tfind. Betrachten wir dazu folgende Aufgabenstellung. In einer Datenbasis sollen Informationen über Autoren und Titel von Büchern verwaltet werden. Dabei seien folgende Operationen in der Datenbasis erlaubt:

- Einfügen einer Information (E)
- Streichen einer Information (S)
- sortierte Ausgabe der Datenbasis in die Standardausgabe (A)
- Beenden der Arbeit (B).

Die Auswahl der einzelnen Operationen erfolgt dialoggesteuert mit den Kommandos E, S, A bzw. B.

Angenommen, der Name des Programms ist buchdb und der Aufruf des Programms wie folgt festgelegt:

```
buchdb  <file_name>
```

Dabei gibt <file_name> den Namen eines Files an, das die Daten enthält. Nach dem Start des Programms werden diese Daten gelesen und zum Aufbau des Binärbaums benutzt. Wir nehmen an, dass genügend Hauptspeicherplatz vorhanden ist, um die Datenbasis vollständig aufzunehmen.

Betrachten wir zunächst die main-Funktion, in der die Dialogsteuerung vorgenommen wird und der Aufruf der vom Nutzer angeforderten Funktionen erfolgt.

```
#include  <stdio.h>
#include  <search.h>
#include  <malloc.h>
#include  <string.h>

struct data {
    char *name;
    char *titel;
};

FILE *fp;
char *filename;

main(argc, argv)                          /* Dialogsteuerung */
    int argc;
    char *argv[];
{
    char *rootp;
    int save_db();
    int tpr();
    char c;

    if(argc != 2) {
        fprintf(stderr, "Aufruf: buchdb <file_name>\n");
        exit(1);
    }
    filename = argv[1];
    init_db(&rootp);
    for(;;) {
        fprintf(stderr,
        "\nEinfuegen(E) | Streichen(S) | Ausgeben(A) | Beenden(B):");
        scanf("%1s", &c);
        switch(c) {
            case 'B':
                if((fp = fopen(filename, "w")) == NULL) {
                    fprintf(stderr, "kein Schreiben moeglich\n");
                    exit(1);
                }
                twalk(rootp, save_db);
                fclose(fp);
                exit(0);
            case 'E':
                insert(&rootp);
                break;
            case 'S':
                delete(&rootp);
                break;
            case 'A':
                twalk(rootp, tpr);
                break;
        }
    }
}
```

Die **include**-Files enthalten die für die Benutzung der Bibliotheksfunktionen erforderlichen Deklarationen. Die Komponenten der Struktur data stellen Zeiger auf den Autor und den Titel eines Buchs dar.

Da die Ausgabe einer sortierten Liste (Operation A) in das Standardausgabefile stdout erfolgt, werden alle Eingabeanforderungen in das Standardausgabefile stderr ausgegeben. Das ermöglicht dem Nutzer beim Aufruf von buchdb die Umlenkung der Standardausgabe auf den Drucker oder in ein File.

Nach der Argumentauswertung erfolgt die Initialisierung der Datenbasis in der Funktion init_db.

```
void init_db(p)                 /* Initialisierung der Datenbasis */
    char **p;
{
    char n1[50], n2[150];
    FILE *fp;

    *p = NULL;
    if((fp = fopen(filename, "r")) != NULL) {
        while(fscanf(fp, "%s%s", n1, n2) == 2)
            makenode(p, n1, n2);
        fclose(fp);
    }
}
```

Existiert das beim Programmaufruf angegebene File noch nicht, so wird lediglich der Zeiger auf die Wurzel des binären Baums mit NULL initialisiert. Ansonsten werden die in dem File enthaltenen Daten gelesen und im Binärbaum gespeichert. An dieser Stelle ist anzumerken, dass die Daten im File nicht sortiert vorliegen sollten, da in dem Fall der Algorithmus uneffektiv ist. Das muss beim Abspeichern des Baumes in das File berücksichtigt werden.

Die Funktion makenode ordnet den erforderlichen Speicherplatz zu und benutzt tsearch, um ein Datenelement im Baum zu installieren.

```
void makenode(rp, n1, n2)       /* Installieren eines Datenelements */
    char **rp;
    char *n1, *n2;
{
    struct data *pd;
    char *pc1, *pc2;
    int pdcmp();

    if((pc1 = malloc(strlen(n1) + 1)) == NULL
    || (pc2 = malloc(strlen(n2) + 1)) == NULL
    || (pd = (struct data *)malloc(sizeof(struct data))) == NULL) {
        fprintf(stderr, "kein Speicher mehr verfuegbar\n");
        exit(1);
    }
    pd->name = strcpy(pc1, n1);
    pd->titel = strcpy(pc2, n2);
    tsearch((char *)pd, rp, pdcmp);
}
```

Wenn der Nutzer die sortierte Ausgabe der Datenbasis verlangt (A), dann erfolgt der Aufruf der Ausgabefunktion tpr durch twalk. Die Ausgabe des zum aktuellen Knoten p gehörigen Autors und Buchtitels wird immer dann vorgenommen, wenn p ein Blatt ist (leaf) bzw. wenn der Knoten nach dem Durchmustern des linken Unterbaumes (postorder) getroffen wird.

```
void tpr(p, visit, level)       /* sortierte Ausgabe */
    struct data **p;
    VISIT visit;
    int level;
{
    if(visit == postorder || visit == leaf)
        printf("%-20.20s %-50.50s\n", (*p)->name, (*p)->titel);
}
```

Bei Beendigung der Arbeit (B) ruft twalk die Funktion save_db auf, um die Informationen in ein File zu speichern. Da die Knoten sowie alle Blätter bereits beim ersten Durchmustern (preorder) verarbeitet werden, wird die gleiche Ordnung beim nächsten Einlesen mit init_db wieder hergestellt.

```
void save_db(p, visit, level)      /* unsortierte Ausgabe */
    struct data **p;
    VISIT visit;
    int level;
{
    if(visit==preorder || visit==leaf)
        fprintf(fp, "%s\t%s\n", (*p)->name, (*p)->titel);
}
```

Um eine Information in die Datenbasis einzufügen (E), wird der Autorname und der Buchtitel vom Terminal angefordert und mittels scanf gelesen.

```
void insert(rp)                    /* Aufbau einer Information */
    char **rp;
{
    char n1[50], n2[150];

    fprintf(stderr, "\nName:"); scanf("%s", n1);
    fprintf(stderr, "Titel:"); scanf("%s", n2);
    makenode(rp, n1, n2);
}
```

Die Funktion delete zum Beseitigen einer Information (S) ist etwas aufwendiger. Vom Nutzer werden Name und Titel der gewünschten Information erfragt. Da die Ordnungsrelation durch den vollständigen Vergleich von Name und Titel (Funktion pdcmp) definiert ist, wählen wir für das Suchen bzw. Löschen folgende Vereinfachung. Als Suchbegriff können Autornamen und Buchtitel verkürzt eingegeben werden. Das erfordert eine modifizierte Vergleichsfunktion (pdncmp). Zur Sicherheit wird der Suchbegriff vor dem Löschen zur Bestätigung vollständig ausgegeben. Die veränderte Vergleichsfunktion kann problemlos verwendet werden, da die Ordnungsrelation des Baums durch die Suchoperationen nicht verletzt werden kann.

```
char *delete(rootp)                /* Loeschen einer Information */
    char **rootp;
{
    char n1[50], n2[150];
    int pdcmp(), pdncmp();
    struct data d;
    struct data *pk;
    char c;

    fprintf(stderr, "\nName:");
    scanf("%s", n1);
    d.name = n1;
    fprintf(stderr, "\nTitel:");
    scanf("%s", n2);
    d.titel = n2;
    if((pk=(struct data *)tfind((char *)&d, rootp, pdncmp)) == NULL)
        fprintf(stderr, "nicht gefunden:%s %s\n", n1, n2);
    else {
        fprintf(stderr, "%s %s ****(y/n):", pk->name, pk->titel);
        scanf("%1s", &c);
        if(c == 'y') {
            free(pk->name);
            free(pk->titel);
            tdelete((char *)pk, rootp, pdcmp);
            free((char *)pk);
        }
    }
}
```

```
int pdcmp(p1, p2)          /* vollständiger Vergleich (Autor/Titel) */
    struct data *p1, *p2;
{
    int i;

    if((i = strcmp(p1->name, p2->name)) != Ø)
        return(i);
    return(strcmp(p1->titel, p2->titel));
}

int pdncmp(p1, p2)         /* verkürzter Vergleich (Autor/Titel) */
    struct data *p1, *p2;
{
    int i;

    if((i = strncmp(p1->name, p2->name, strlen(p1->name))) != Ø)
        return(i);
    return(strncmp(p1->titel, p2->titel, strlen(p1->titel)));
}
```

### 3.5.3. Verwalten von Ressourcen in doppelt verketteten Listen

Listenstrukturen lassen sich oft vorteilhaft zur Verwaltung bestimmter Ressourcen, wie z.B. Hauptspeicherbereiche, anwenden. Als Beispiel betrachten wir ein vereinfachtes Problem der Platzkartenverteilung, für das eine doppelt verkettete Liste (s. Bild 3.5) verwendet werden kann. Eine kontinuierliche Folge von Plätzen soll in einem Listenelement enthalten sein. Zwei wichtige Operationen für die Arbeit mit Listen sind das Einfügen bzw. Entfernen von Listenelementen aus der Kette.

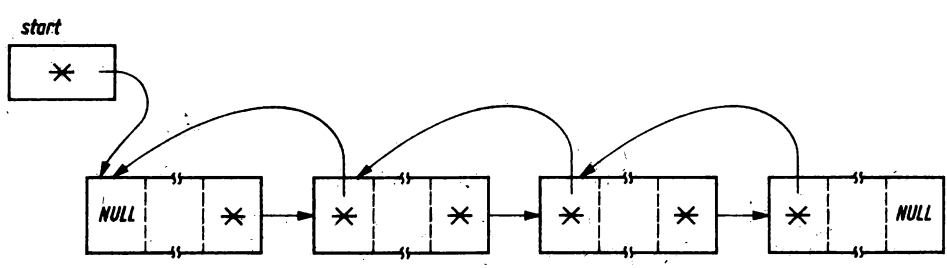



Bild 3.5. Doppelt verkettete Liste

Betrachten wir die Deklaration eines Listenelements, so stellen wir grosse Ähnlichkeit zur weiter vorn vereinbarten Knoteninstanz eines Binärbaums fest. Die Datenstrukturen Binärbaum und Liste werden also im wesentlichen durch die Regeln charakterisiert, nach denen die Datenelemente verknüpft sind.

```
struct listelem {
    int anz;
    int start;
    struct listelem *next;
    struct listelem *last;
};
```

Zu Beginn ist das Problem der Platzkartenverteilung einfach. Es existiert nur ein Listenelement, in dem start die erste freie Platznummer und anz die Anzahl der verfügbaren Plätze angibt. Das Reservieren von Karten besteht demzufolge einfach in der Korrektur der Werte start und anz, wenn die Karten mit start beginnend verteilt werden. Wir wollen jedoch auch das Problem der Rückgabe von Plätzen behandeln.

Die folgende main-Funktion initialisiert das erste Listenelement und führt den Dialog. Die Reservierung oder Rückgabe von Plätzen erfolgt, indem die Funktionen reserviere oder storniere aufgerufen werden.

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

#define  MAX 2ØØØ

struct listelem {
    int anz;
    int start;
    struct listelem *next;
    struct listelem *last;
};
struct listelem *start;

main()                                  /* Platzreservierung */
{
    register struct listelem *ls;
    int c, val;

    start = ls = (struct listelem *)malloc(sizeof(struct listelem));
    ls->anz = MAX;
    ls->start = 1;
    ls->last = ls->next = NULL;
    for(;;) {
        printf("\nReservieren oder Stornieren? (R/S):");
        if((c = getchar()) == EOF)
            exit(Ø);
        printf("\nAnzahl?:");
        scanf("%d", &val);
        if(c=='R') {
            if((c = reserviere(val)) == Ø)
                printf("Das ist leider nicht moeglich!\n");
            else
                printf("Sie belegen die Plaetze von %d bis %d\n",
                        c, c + val -1);
        } else if(c=='S') {
            printf("\nWelches ist die erste Platznummer?:");
            scanf("%d", &c);
            storniere(c, val);
        }
    }
}
```

Die Funktion reserviere sucht in der Liste nach einem Listenelement, in dem die geforderte Anzahl von Plätzen vollständig enthalten ist. Für ein solches Element werden die Werte anz und start korrigiert und damit die Plätze reserviert. Danach erfolgt die Kontrolle, ob das Listenelement

keine Plätze mehr enthält. Ist das der Fall, so muss dieses Element aus der Liste entfernt werden, indem das vorhergehende Listenelement mit dem nachfolgenden verkettet wird. Der Platz für das leere Listenelement wird anschliessend freigegeben. Ist die geforderte Anzahl von Plätzen nicht in einem Element enthalten, so liefert die Funktion reserviere den Wert Ø.

```
int reserviere(anzahl)              /* Reservieren von Plätzen */
    int anzahl;
{
    register struct listelem *p, *q;
    int rval;

    for(p = start; p != NULL; p = p->next) {
        if(p->anz >= anzahl) {
            p->anz -= anzahl;
            rval = p->start;
            p->start += anzahl;
            if(p->anz == Ø) {
                p->last->next = p->next;
                p->next->last = p->last;
                free((char *)p);
            }
            return(rval);
        }
    }
    return(Ø);
}
```

Die Funktion storniere nimmt eine kontinuierliche Anzahl zurückgegebener Plätze wieder in die Liste auf. Im einfachsten Fall können durch Korrektur der Werte start und anz alle Plätze in ein vorhandenes Listenelement aufgenommen werden. Ist das nicht möglich, so muss ein neues Element eingerichtet und in die Liste eingekettet werden.

```
void storniere(beg, anzahl)         /* Freigeben von Plätzen */
    int beg;
    int anzahl;
{
    register struct listelem *p, *lastp;

    for(p = start, lastp = NULL; p != NULL; lastp = p, p = p->next) {
        if(beg == p->start + p->anz) {                          /*1*/
            p->anz += anzahl;
            if(p->anz + p->anz == p->next->start) {             /*2*/
                p->anz += p->next->anz;
                p->next = p->next->next;
                free((char *)p->next->last);
                p->next->last = p;
            }
            return;
        } else if(beg + anzahl == p->start) {                   /*3*/
            p->start = beg;
            p->anz += anzahl;
            return;
        } else if(beg < p->start)  {                            /*4*/
            newelement(anzahl, beg, p, p->last);
            return;
        } else {                                                /*5*/
            continue;
        }
    }
    newelement(anzahl, beg, NULL, lastp);                       /*6*/
}
```

Die **for**-Schleife steuert die Untersuchung der einzelnen Listenelemente.

Zunächst wird untersucht, ob die zurückgegebenen Plätze mit dem Ende des aktuellen Listenelements verknüpft werden können (/*1*/). In diesem Fall braucht man nur die Platzanzahl anz zu korrigieren. Es muss weiterhin geprüft werden, ob sich daraus eine Verknüpfungsmöglichkeit mit dem Folgeelement ergibt (/*2*/). Ist die Verknüpfung möglich, so erfolgt die Korrektur der Platzanzahl und der Verkettung sowie die Freigabe eines Listenelements.

Der zweite Fall berücksichtigt die Möglichkeit, die Plätze mit dem Anfang des Listenelements zu verknüpfen (/*3*/). Die Werte von beg und anz werden korrigiert. Eine Verknüpfungsmöglichkeit mit dem Vorgänger kann nicht mehr bestehen.

Ist beides nicht möglich, so muss ein neues Listenelement eingerichtet (/*4*/) oder das nächste Element untersucht (/*5*/) werden.

Die Funktion newelement zum Erzeugen eines neuen Listenelements wird auch beim Erreichen des Listenendes aufgerufen (/*6*/).

```
void newelement(anzahl, beg, p, lastp) /* neues Listenelement */
    int anzahl;
    int beg;
    struct listelem *p;
    struct listelem *lastp;
{
    register struct listelem *new;

    if((new = (struct listelem *)malloc(sizeof(*new))) == NULL) {
        printf("malloc error\n");
        exit(1);
    }
    new->anz = anzahl;
    new->start = beg;
    new->next = p;
    new->last = lastp;
    if(p != NULL)
        p->last = new;
    if(lastp == NULL)
        start = new;
    else
        lastp->next = new;
}
```

Um ein neues Listenelement einrichten zu können, fordert newelement über malloc Speicherplatz an. Danach werden die Informationen beg und anz eingetragen und die erforderliche Verkettung hergestellt. Unter Umständen erfolgt die Korrektur des Zeigers auf den Listenanfang.

### 3.5.4. Ein erweiterter Tischrechner

Das folgende Tischrechnerprogramm bietet nochmals die Gelegenheit, auf die Verwendung von Strukturen und die Nutzung rekursiver Programmierung einzugehen.

Die wesentliche Erweiterung gegenüber der im Abschn. 3.3.3. vorgestellten Tischrechnerversion besteht in der Möglichkeit, Funktionen zu definieren und aufzurufen. Zunächst muss die äussere Schnittstelle festgelegt werden. Besonders einfach wird die syntaktische Analyse der Eingabe, wenn man die umgekehrte polnische Notation verwendet. Weiterhin soll jeder

Operand und Operator mit dem ersten Zeichen einer Zeile beginnen, d.h. allein auf einer Zeile stehen.

Folgende Eingabezeilen seien möglich:

- Wenn das erste Zeichen eine Ziffer ist, dann wird die gesamte Zeile als Realzahl interpretiert.
- Die zulässigen Operatoren sind +, -, *, /, = und ^, wobei ^ für das unäre Minus steht. Durch = wird das Lesen und Anzeigen des obersten Kellerspeichereintrags bewirkt.
- Ist das erste Zeichen #, so wird durch die folgenden 4 Zeichen ein Funktionsname definiert.
- Bei einem ersten Zeichen ¤ spezifiziert die folgende Ziffer einen formalen Funktionsparameter.
- Alle anderen Zeichen werden als Funktionsaufruf interpretiert, wobei die aktuellen Funktionsparameter im Kellerspeicher stehen müssen. Dorthin wird auch das Funktionsergebnis abgelegt.

Die strengen Syntaxforderungen begrenzen den erforderlichen Aufwand für die syntaktische Analyse, aber natürlich auch die Nutzerfreundlichkeit. Hier soll das genügen.

Funktionen bzw. arithmetische Ausdrücke lassen sich sehr einfach in Ausdrucksbäumen speichern und auswerten. Ein Knoten des Baumes definiert einen Operator. Die Blätter sind Konstanten oder definieren einen formalen Funktionsparamter.

Verdeutlichen wir das am Beispiel einer Zinsberechnung nach der Formel

```
zins(x) = 3.25 * x / 100
```

Die Funktion zins kann durch folgende Eingaben definiert werden:

```
3.25
¤1
*
100.
/
#zins
```

Diese Funktion soll nun angewendet werden, um die Aufgabe

```
zins(1000) + zins(100)
```

zu lösen:

```
1000.
zins
100.
zins
+
=
35.75
```

Der Wert 35.75 stellt das Ergebnis dieser Aufgabe dar.

Die Eingabesyntax lässt sich auf einfache Weise in der main-Funktion realisieren, die sich in einem File trØ.c befinden soll. Die Verarbeitung der erkannten syntaktischen Einheiten wird in spezielle Funktionen verlagert.

File trØ.c:

```
#include "defs.h"

#define MAX 2Ø

main()                                  /* Programmierbarer Tischrechner */
{
    char s[MAX];
    char *ps;

    while(ps = fgets(s, MAX, stdin)) {
        switch (*ps) {
            case 'ø':
                argument(++ps);
                break;
            case '+':
            case '-':
            case '*':
            case '/':
            case '^':
                operator(ps);
                break;
            case '#':
                funcdef(++ps);
                break;
            case '=':
                printf("%f\n", eval(pop(), NULL));
                break;
            default:
                if(*ps >= 'Ø' && *ps <= '9')
                    konstante(ps);
                else
                    funccall(ps);
                break;
        }
    }
}
```

Wenn man auch noch nicht exakt wissen muss, wie die in main verwendeten Funktionen argument, operator, funcdef, eval, konstante, funccall und pop arbeiten, so sollte man sich jedoch schon vorher über die erforderlichen Datenstrukturen und Algorithmen Gedanken machen.

Die wichtigste Datenstruktur ist node, die einen Knoten im Binärbaum beschreibt. Die Strukturkomponente n_typ legt den Knotentyp (KONSTANTE, ARGUMENT oder OPERATOR) fest. Für jede eingegebene Zeile kann ein solcher Knoten gebildet werden, wobei Blattknoten (KONSTANTE, ARGUMENT) im Kellerspeicher abgelegt werden. Beim Knotentyp OPERATOR wird der Kellerspeicher gelesen, die gelesenen Einträge mit dem OPERATOR-Knoten verknüpft und das Ergebnis wieder gekellert.

Mit Hilfe der Struktur fnode, die den Funktionsnamen f_name, die Anzahl der Argumente f_args und den Zeiger auf die Wurzel des Ausdrucksbaums f_root enthält, erfolgt die Definition einer Tischrechnerfunktion.

Betrachten wir die konkreten Definitionen und Deklarationen im File defs.h.

File defs.h:

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

#define NFUNC   1Ø      /* maximal 1Ø Funktionen */
#define LEFT     Ø
#define RIGHT    1
#define KONSTANTE  Ø
#define OPERATOR   1
#define ARGUMENT   2
#define MAXARGS   1Ø

#define TFREE(args, pn) if(args == NULL) free((char *)pn)

struct node {
    char n_typ;
    char n_op;
    union {
        double u_val;
        struct node *u_p[2];
    } n_u;
};
struct fnode {
    char f_name[4];
    short f_nargs;
    struct node *f_root;
};

extern struct fnode pfunc[];
extern short funcanz;
extern void argument(), operator();
extern void funcdef(), konstante(), funccall();
extern double eval();
extern struct node *pop();
extern int maxarg;
```

Es ist jetzt einfach, alle benötigten Funktionen (sie befinden sich im File tr1.c) aufzuschreiben.

File tr1.c:

```
#include "defs.h"

struct fnode *pfunc[NFUNC];
short funcanz = Ø;
int maxarg = Ø;

void argument(ps)                   /* Argumentknoten einrichten */
    char *ps;
{
    int number;
    register struct node *pn;

    number = atoi(ps);
    if(number < 1 || number > MAXARGS) {
        fprintf(stderr,"argnumber fault: %d\n", number);
        return;
    }
    pn = (struct node *) malloc(sizeof(*pn));
    pn->n_typ = ARGUMENT;
    pn->n_op = number;
    push(pn);
    maxarg = number > maxarg ? number : maxarg;
}
```

Der Aufruf der Funktion argument erfolgt, wenn in einem arithmetischen Ausdruck ein formaler Parameter ($i) angegeben wurde. Die Integerzahl i wird konvertiert und auf Gültigkeit geprüft, anschliessend ein Blattknoten vom Typ ARGUMENT gebildet und in den Kellerspeicher geschrieben. Ausserdem erfolgt das Setzen der externen Variablen maxarg, die bei der zugehörigen Funktionsdefinition benötigt wird.

```
void konstante(ps)                  /* Konstantenknoten einrichten */
    char *ps;
{
    register struct node *pn;
    double val;

    if(sscanf(ps, "%lf", &val) == Ø) {
        fprintf(stderr, "%s: no float constant\n", ps);
        return;
    }
    pn = (struct node *) malloc(sizeof(*pn));
    pn->n_typ = KONSTANTE;
    pn->n_u.u_val = val;
    push(pn);
}
```

Die Funktion konstante konvertiert die eingegebene Realkonstante, die sich in der Zeichenkette ps befindet, mittels sscanf. Gegebenenfalls erfolgt eine Fehlermitteilung, und der Nutzer kann die Eingabe wiederholen. Für die Konstante richtet die Funktion eine Datenstruktur node ein, deren Speicherplatz dynamisch mittels malloc angefordert wird. Wenn nicht mehr genügend Speicherplatz vorhanden ist, dann liefert die Bibliotheksfunktion malloc den Zeigerwert NULL. (In seriösen Programmen muss dieser Fall natürlich berücksichtigt werden. Auf Grund der gewählten Werte von NFUNC bzw. MAXSTACK (s. File tr2.c) wird eine derartige Situation jedoch kaum auftreten.) Nach dem Eintragen des Knotentyps (KONSTANTE) und des Wertes der Konstante wird die Adresse des Blattknotens mittels push in den Kellerspeicher abgelegt.

```
void operator(ps)                   /* Operatorknoten einrichten */
    char *ps;
{
    register struct node *pn;

    pn = (struct node *) malloc(sizeof(*pn));
    switch(*ps) {
        case '+':
        case '-':
        case '*':
        case '/':
            pn->n_typ = OPERATOR;
            pn->n_op = *ps;
            pn->n_u.u_p[RIGHT] = pop();
            pn->n_u.u_p[LEFT] = pop();
            push(pn);
            break;
        case '^':
            pn->n_typ = OPERATOR;     /* unäres Minus */
            pn->n_op = '^';
            pn->n_u.u_p[RIGHT] = NULL;
            pn->n_u.u_p[LEFT] = pop();
            push(pn);
            break;
    }
}
```

Die Funktion operator richtet einen Knoten mit dem Typ OPERATOR ein. Je nachdem, ob es sich um einen unären oder binären Operator handelt, werden eine oder zwei Knotenadressen aus dem Kellerspeicher gelesen (pop) und mit dem Operatorknoten verbunden. Auf diese Weise erfolgt der schrittweise Aufbau eines Ausdrucksbaums.

```
void funcdef(ps)          /* Definition einer Tischrechnerfunktion */
    char *ps;
{
    register int i;
    register struct fnode *pf;

    for(i = Ø; i < funcanz; i++) {
        if(strncmp(pfunc[i].f_name, ps, 4))
            continue;
        cleartree(pfunc[i].f_root);
        break;
    }
    if(i >= NFUNC) {
        fprintf(stderr, "too many functions\n");
        return;
    }
    if(i < funcanz)
        pf = &pfunc[i];
    else
        pf = &pfunc[funcanz++];
    strncpy(pf->f_name, ps, 4);
    pf->f_nargs = maxarg;
    maxarg = Ø;
    pf->f_root = pop();
}
```

Die Funktion funcdef realisiert die Definition einer Tischrechnerfunktion (#name). Sie überprüft zunächst, ob schon eine gleichnamige Funktion existiert. Ist das der Fall, so wird die alte Funktion überschrieben und der zugehörige Ausdrucksbaum mit cleartree freigegeben. Ist ein freier Eintrag vorhanden, so bildet funcdef die zugehörige Datenstruktur fnode und trägt den Funktionsnamen, die Anzahl der verwendeten symbolischen Parameter und einen Zeiger auf die Wurzel des Ausdrucksbaums ein.

```
void cleartree(pn)        /* Freigeben eines Teilbaums */
    register struct node *pn;
{
    register struct node *pl = NULL;
    register struct node *pr = NULL;

    if(pn->n_typ == OPERATOR) {
        pl = pn->n_u.u_p[LEFT];
        pr = pn->n_u.u_p[RIGHT];
    }
    if(pl) {
        cleartree(pl);
        free((char *)pl);
    }
    if(pr) {
        cleartree(pr);
        free((char *)pr);
    }
    if(pn)
        free((char *)pn);
}
```

Die Aufgabe der Funktion cleartree ist schon bekannt. Der Binärbaum wird nach links absteigend untersucht. Blattknoten können direkt beseitigt werden. Ist der linke Unterbaum vollständig beseitigt, so erfolgt die Untersuchung des zugehörigen rechten Unterbaumes. Ist auch dieser freigegeben, so kann der jeweilige Ausgangsknoten selbst beseitigt werden.

Wer noch Probleme mit der rekursiven Programmierung hat, sollte sich einen Binärbaum aufzeichnen und die Arbeitsweise der Funktion cleartree verfolgen. Ein anschaulicheres Beispiel für eine rekursive Funktion wird man kaum finden.

```
void funccall(ps)                    /* Aufruf einer Funktion */
    char *ps;
{
    register struct node *pn;
    register int i;
    register int j;
    double args[MAXARGS];

    for(i = Ø; i < funcanz; i++) {
        if(strncmp(pfunc[i].f_name, ps, 4))
            continue;
        for(j = pfunc[i].f_nargs; j > Ø;) {
            pn = pop();
            if(pn->n_typ != KONSTANTE) {
                fprintf(stderr, "illegal argument\n");
                return;
            }
            args[--j]  = pn->n_u.u_val;
        }
        pn = (struct node *) malloc(sizeof(*pn));
        pn->n_typ = KONSTANTE;
        pn->n_u.u_val = eval(pfunc[i].f_root, args);
        push(pn);
        return;
    }
    fprintf(stderr, "func: %s not defined\n", ps);
}
```

Die Funktion funccall sucht zunächst nach dem Funktionsnamen. Wird die zugehörige Datenstruktur fnode gefunden, so können auch die benötigten Argumente aus dem Kellerspeicher gelesen werden. Der Wurzelknoten des Ausdrucksbaums und die zugehörige Argumentliste werden an die Funktion eval übergeben, die das Ergebnis ermittelt und bereitstellt. Den so gebildeten Knoten legt funccall in den Kellerspeicher ab. Das ist durchaus sinnvoll, da auf diese Weise das Ergebnis des Funktionsaufrufs gleich weiter verwendet werden kann.

Als nächstes betrachten wir die Funktion eval. Was die Durchmusterung des Binärbaums anbetrifft, so arbeitet eval in der gleichen Weise wie die Funktion cleartree. Handelt es sich um einen Blattknoten (KONSTANTE), wird als Ergebnis der zugehörige Konstantenwert geliefert. Bei einem Operatorknoten (OPERATOR) erfolgt vor der Rückgabe des Ergebniswertes erst die erforderliche Berechnung des Teilbaums. Die zugehörigen Operandenwerte werden wiederum mittels eval ermittelt. Erfolgte der ursprüngliche Aufruf von eval nicht durch die Funktion funccall, sondern direkt aus main (Zeiger auf die Argumentliste ist NULL), so müssen die Knoten des Ausdrucksbaums freigegeben werden. Für diesen Zweck wird der Makro TFREE verwendet.

```
double eval(pn, args)              /* Berechnen eines Ausdrucksbaums */
    register struct node *pn;
    double args[];
{
    double val1, val2;
    int op;

    switch(pn->n_typ) {
        case KONSTANTE:
            val1 = pn->n_u.u_val;
            TFREE(args, pn);
            return(val1);
        case ARGUMENT:
            return(args[pn->n_op - 1]);
        case OPERATOR:
            val1 = eval(pn->n_u.u_p[LEFT], args);
            val2 = pn->n_u.u_p[RIGHT] ?
                   eval(pn->n_u.u_p[RIGHT], args) : Ø;
            op = pn->n_op;
            TFREE(args, pn);
            switch(op) {
                case '+':
                    return(val1 + val2);
                case '-':
                    return(val1 - val2);
                case '*':
                    return(val1 * val2);
                case '/':
                    if(val2 == Ø) {
                        fprintf(stderr, "division by zero\n");
                        return(Ø);
                    }
                    return(val1 / val2);
                case '^':
                    return(-val1);
            }
    }
}
```

Jetzt fehlt nur noch die Verwaltung des Kellerspeichers.

File tr2.c:

```
#include "defs.h"

#define MAXSTACK 2ØØ

static struct node *pns[MAXSTACK];
static int istack = Ø;

struct node *pop()                 /* Lesen eines Stack-Elements */
{
    if(--istack >= Ø)
        return(pns[istack]);
    fprintf(stderr,"fatal error: stack empty \n");
    exit(1);
}
void push(pn)                      /* Speichern eines Stack-Elements */
    register struct node *pn;
{
    if(istack >= MAXSTACK) {
        fprintf(stderr, "fatal error: stack owerflow\n");
        exit(1);
    }
    pns[istack++] = pn;
}
```

Es ist sinnvoll, die Variablen pns und istack als **static**-Variable zu vereinbaren, da die anderen Funktionen nur über die Funktion push und pop mit dem Kellerspeicher arbeiten sollen. Die Mittel der separaten Übersetzung und das Einschränken der Sichtbarkeit bieten die Möglichkeit einer höheren Datenabstraktion. Der Programmierer legt beim Programmentwurf als Schnittstelle nach aussen die Funktionen für den Zugriff auf die Datenobjekte (hier push und pop) fest. Die Implementation der Datenobjekte selbst ist diesen Funktionen vorbehalten. Somit ist es z.B. möglich, die auf den Hauptspeicher orientierte Kellerspeicherverwaltung problemlos auf die im Abschn. 3.4.2. angegebene Verwaltung eines "File-Kellerspeichers" umzustellen.

### 3.5.5. Interaktives Ändern von Binärfiles

Das folgende Programm bietet die Möglichkeit, auf einfache Weise Files mit beliebigen Binärinformationen zu modifizieren. Da es sich um ein interaktives Programm handelt, werden Interruptsignale behandelt.

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

jmp_buf jmpbuf;

main(argc, argv)                /* Interaktives Ändern von Binärfiles */
    int argc;
    char *argv[];
{
    int fd;
    int onintr();

    if(argc != 2) {
        printf("usage: zap file\n");
        exit(1);
    }
    if((fd = open(argv[1], 2)) < 0) {
        printf("can't open %s\n", argv[1]);
        exit(1);
    }
    setjmp(jmpbuf);
    signal(SIGINT, onintr);
    for(;;)
        switch(getchar()) {
            case 'd':
            case 'D':
                zap(fd, 'd');
                break;
            case 'r':
            case 'R':
                zap(fd, 'r');
                break;
            case 'e':
            case 'E':
                exit(0);
            case '\n':
            case ' ':
                break;
            default:
                printf("format error\n");
                break;
        }
}
```

Die main-Funktion wertet die Aufrufparameter aus und öffnet das zu bearbeitende File. Ausserdem befindet sich hier die zentrale Schleife, in der jeweils ein Zeichen von der Standardeingabe gelesen wird. Über dieses Zeichen erfolgt die Einleitung einer konkreten Aktion, wobei der Anwender des Programms unter folgenden Aktionen auswählen kann:

- Ausdrucken (d bzw. D)
- Ersetzen (r bzw. R)
- Beenden der Arbeit (e bzw. E).

Die eigentlichen Operationen Ausdrucken bzw. Ersetzen werden in der Funktion zap realisiert. Das soll uns jedoch zunächst nicht interessieren.

Unterbricht der Anwender eine laufende Aktion, dann soll am Beginn der zentralen Schleife in der main-Funktion fortgesetzt, d.h. eine neue Eingabe vom Nutzer angefordert werden. Zur Unterstützung einer derartigen Interruptbehandlung existieren in der Standardbibliothek die Funktionen setjmp und longjmp sowie das File setjmp.h.

```
#include <setjmp.h>

int setjmp(env)
    jmp_buf env;

void longjmp(env, val)
    jmp_buf env;
    int val;
```

Mit setjmp werden in jmpbuf alle erforderlichen Informationen gespeichert, um mit longjmp aus beliebig tieferen Funktionsniveaus direkt zurückkehren zu können. Das geschieht in unserem Beispiel in der Interruptbehandlungsroutine onintr.

```
void onintr()                    /* Interruptbehandlung */
{
    longjmp(jmpbuf);
}
```

Die Systemfunktion signal wird absichtlich nach setjmp aufgerufen, da nach dem Auftreten eines Interruptsignals die Interruptbehandlung neu festgelegt werden muss.

Die Funktion zap erwartet für die Operationen Ausdrucken bzw. Ersetzen, dass der Anwender zwei Hexadezimalzahlen eingibt, die die Anfangs- und Endeposition im File für die entsprechende Aktion festlegen. Mit lseek wird die Anfangsposition eingestellt und anschliessend bis zur Endeposition gelesen. Entsprechend der geforderten Aktion wird entweder nur hexadezimal ausgegeben oder neben der Ausgabe auch die Möglichkeit der Korrektur angeboten. Gibt der Nutzer bei der Korrektur keine gültigen Hexadezimalziffern ein, so wird das betreffende Zeichen nicht überschrieben. Bei einer Korrekturanforderung ist das Rücksetzen des Files auf die Ausgangsposition mittels lseek erforderlich.

Möchte der Anwender die laufende Aktion vorzeitig beenden, so muss er die Interrupt-Taste des Terminals Betätigen und kann danach eine neue Aktion auswählen.

```
void zap(fd, c)                    /* Anzeigen und Ändern */
    int fd;
    int c;
{
    int pos1;
    int pos2;
    int j;
    register int i;
    register int anz;
    char buf[24];

    if(scanf("%x%x", &pos1, &pos2) != 2) {
        printf("format error\n");
        return;
    }
    pos2 -= pos1;
    pos2++;
    lseek(fd, (long) pos1, 0);
    for( ;
       pos2 > 0 && (anz = read(fd, buf, pos2 > 24 ? 24 : pos2)) > 0;
       pos2 -= anz) {
        if(c == 'r') {
            lseek(fd, (long) -anz, 1);
            for(i = 0; i < anz; i++) {
                printf("%.4x %.2x", pos1, buf[i]);
                fflush(stdout);
                pos1++;
                if(scanf("%x", &j) == 1)
                    buf[i] = j;
                else
                    getchar();
            }
            write(fd, buf, anz);
        } else {
            printf("%.4x  ", pos1);
            pos1 += anz;
            for(i = 0; i < anz; i++)
                printf("%.2x%s", buf[i], (i+1)%4 ? "" : " ");
            putchar('\n');
        }
    }
}
```

# Anhang A. Syntaxübersicht

Die Zusammenfassung der Syntax erfolgt in einer erweiterten Backus-Naur-Notation. Dabei werden Terminalsymbole im Fettdruck dargestellt. Eine Syntaxregel hat die Form

```
Nichtterminalsymbol:
   syntaktische Konstruktion
```

In den syntaktischen Konstruktionen werden die Metasymbole |, [], {} verwendet:

x|y Auswahl einer der syntaktischen Konstruktionen x oder y (Alternative)

[x] wahlweises Auftreten der syntaktischen Konstruktion x (Option)

{x} Die Konstruktion x kann einmal, mehrmals oder überhaupt nicht auftreten (Wiederholung).

C-Programme können im freien Format notiert werden. Leerzeichen, Tabulator- und Newlinezeichen sowie Kommentare werden ignoriert, abgesehen davon, dass sie zur Trennung direkt benachbarter Bezeichner, Konstanten oder Schlüsselwörter dienen.

```
modul:
       externe_vereinbarung
     | externe_vereinbarung modul

externe_vereinbarung:
       funktions_definition
     | daten_definition
     | externe_deklaration
     | typ_definition

funktions_definition:
       funktions_kopf  funktions_körper

funktions_kopf:
       [static] [typ] funktions_deklarator [par_deklarations_liste]

funktions_deklarator:
       funktions_name
     | (funktions_deklarator)
     | *funktions_deklarator
     | (*funktions_deklarator) ()
     | (*funktions_deklarator) [[konst_ausdruck]]
```

```
funktions_name:
        bezeichner([parameter_liste])


parameter_liste:
        bezeichner
      | bezeichner,parameter_liste


par_deklarations_liste:
        par_deklaration
      | par_deklarations_liste  par_deklaration


par_deklaration:
        register [typ]  deklarator_liste;
      | typ deklarator_liste;
      | typ_definition


daten_definition:
        static [typ]  init_deklarator_liste;
      | typ init_deklarator_liste;


externe_deklaration:
        extern [typ]  deklarator_liste;
      | extern funktions_deklaration;


typ_definition:
        typedef [typ] deklarator_liste;


funktions_körper:
        block


block:
        {[vereinbarungs_liste] [anweisungs_liste] }


vereinbarungs_liste:
        interne_vereinbarung
      | interne_vereinbarung  vereinbarungs_liste


interne_vereinbarung:
        funktions_deklaration
      | interne_daten_definition
      | externe_deklaration
      | typ_definition
```

```
funktions_deklaration:
       [typ] funktions_bezeichner;


funktions_bezeichner:
       bezeichner()
     | (funktions_bezeichner)
     | *funktions_bezeichner
     | (*funktions_bezeichner)()
     | (*funktions_bezeichner)[]


interne_daten_definition:
       speicherklasse [typ] init_deklarator_liste;
     | typ  init_deklarator_liste;


speicherklasse:
       auto
     | register
     | static


typ:
       integer_typ
     | unsigned
     | unsigned integer_typ
     | float
     | long float
     | double
     | struct  strukt_deklarator
     | union   strukt_deklarator
     | enum  enum_deklarator
     | typ_bezeichner
     | void


integer_typ:
       char
     | int
     | short
     | long
     | short int
     | long int


strukt_deklarator:
       [strukt_typ_bezeichner] {element_deklarations_liste[;]}
     | strukt_typ_bezeichner


strukt_typ_bezeichner:
       bezeichner
```

```
element_deklarations_liste:
       element_deklaration
     | element_deklaration  element_deklarations_liste


element_deklaration:
       typ  element_deklarator_liste;


element_deklarator_liste:
       element_deklarator
     | element_deklarator,element_deklarator_liste


element_deklarator:
       deklarator
     | [deklarator] : konst_ausdruck


enum_deklarator:
       [enum_typ_bezeichner] {enumerator_liste[,]}
     | enum_typ_bezeichner


enum_typ_bezeichner:
       bezeichner


enumerator_liste:
       enumerator
     | enumerator,enumerator_liste


enumerator:
       enum_bezeichner [=konst_ausdruck]


enum_bezeichner:
       bezeichner


typ_bezeichner:
       strukt_typ_bezeichner
     | enum_typ_bezeichner
     | bezeichner


init_deklarator_liste:
       init_deklarator
     | init_deklarator,init_deklarator_liste
```

```
init_delarator:
        deklarator
      | deklarator = init_wert
      | deklarator = {init_wert_liste[,]}


init_wert_liste:
        init_wert
      | init_wert,init_wert_liste


init_wert:
        ausdruck
      | {init_wert_liste[,]}


deklarator_liste:
        deklarator
      | deklarator,deklarator_liste


deklarator:
        bezeichner
      | (deklarator)
      | *deklarator
      | (*deklarator)()
      | (*deklarator)[]
      | feld_deklarator


feld_deklarator:
        bezeichner[konst_ausdruck]
      | feld_deklarator[konst_ausdruck]


anweisungs_liste:
        anweisung
      | anweisung  anweisungs_liste


anweisung:
        block
      | marke: anweisung
      | if(ausdruck) anweisung [else anweisung]
      | while(ausdruck) anweisung
      | do anweisung while(ausdruck);
      | for([ausdruck];[ausdruck];[ausdruck]) anweisung
      | switch_anweisung
      | break;
      | continue;
      | return[ausdruck];
      | goto marke;
      | ausdruck;
      | ;
```

```
switch_anweisung:
        switch ausdruck {case_liste[default_zweig][case_liste]}
      | switch ausdruck {default_zweig case_liste}


case_liste:
        case_zweig
      | case_zweig case_liste


case_zweig:
        case konst_ausdruck : [anweisungs_liste]


default_zweig:
        default : [anweisungs_liste]


marke:
        bezeichner


ausdruck:
        einfacher_ausdruck
      | konst_ausdruck
      | !ausdruck
      | ~ausdruck
      | inc_op lvalue
      | lvalue inc_op
      | -ausdruck
      | (typ_spezifikation) ausdruck
      | &lvalue
      | *ausdruck
      | ausdruck mul_op ausdruck
      | ausdruck add_op ausdruck
      | ausdruck shift_op ausdruck
      | ausdruck rel_op ausdruck
      | ausdruck bit_op ausdruck
      | ausdruck log_op ausdruck
      | ausdruck ? ausdruck : ausdruck
      | lvalue ass_op ausdruck
      | ausdruck , ausdruck
      | (ausdruck)


einfacher_ausdruck:
        bezeichner
      | konstante
      | (ausdruck)
      | einfacher_ausdruck([argument_liste])
      | einfacher_ausdruck[ausdruck]
      | einfacher_ausdruck.bezeichner
      | einfacher_ausdruck->bezeichner
```

```
lvalue:
        bezeichner
      | einfacher_ausdruck[ausdruck]
      | lvalue.bezeichner
      | einfacher_ausdruck->bezeichner
      | *ausdruck
      | (lvalue)


argument_liste:
        ausdruck
      | ausdruck,argument_liste


konst_ausdruck:
        integer_konstante
      | long_integer_konstante
      | zeichen_konstante
      | (konst_ausdruck)
      | ~konst_ausdruck
      | -konst_ausdruck
      | sizeof ausdruck
      | sizeof(typ_spezifikation)
      | enum_konstante
      | konst_ausdruck mul_op konst_ausdruck
      | konst_ausdruck add_op konst_ausdruck
      | konst_ausdruck shift_op konst_ausdruck
      | konst_ausdruck rel_op konst_ausdruck
      | konst_ausdruck bit_op konst_ausdruck
      | konst_ausdruck ? konst_ausdruck : konst_ausdruck


enum_konstante:
        enum_bezeichner


typ_spezifikation:
        typ  abstrakter_deklarator


abstrakter_deklarator:
        [abstrakter_deklarator]
      | (abstrakter_deklarator)
      | *abstrakter_deklarator
      | abstrakter_deklarator()
      | abstrakter_deklarator[[konst_ausdruck]]


ass_op:
        = | mul_op= | add_op= | shift_op= | bit_op=


inc_op:
        ++ | --
```

```
mul_op:
        * | / | %


add_op:
        + | -


shift_op:
        << | >>


rel_op:
        < | <= | > | >= | == | !=


bit_op:
        & | ^ | |


log_op:
        && | ||


bezeichner:
        buchstabe{buchstabe|ziffer}


konstante:
        integer_konstante
      | long_integer_konstante
      | gleitkomma_konstante
      | zeichen_konstante
      | zeichenketten_konstante


integer_konstante:
        dezimal_konstante
      | oktal_konstante
      | hexadezimal_konstante


dezimal_konstante:
        nicht_null_ziffer[dezimalzahl]


oktal_konstante:
        Øoktalziffer{oktalziffer}


hexadezimal_konstante:
        Øxhexadezimalziffer{hexadezimalziffer}
      | ØXhexadezimalziffer{hexadezimalziffer}
```

```
long_integer_konstante:
        integer_konstanteL
      | integer_konstantel


gleitkomma_konstante:
        dezimalzahl.dezimalzahl[exponent]
      | dezimalzahl.[exponent]
      | dezimalzahlexponent
      | .dezimalzahl[exponent]


exponent:
        e[+|-] dezimalzahl
      | E[+|-] dezimalzahl


zeichen_konstante:
        'buchstabe'
      | 'ziffer'
      | 'sonderzeichen'
      | 'nichtgrafisches_zeichen'
      | 'anführungszeichen'


zeichenketten_konstante:
        "{zeichen}"


zeichen:
        buchstabe
      | ziffer
      | sonderzeichen
      | apostroph
      | nichtgrafisches_zeichen


dezimalzahl:
        ziffer{ziffer}


hexadezimalziffer:
        ziffer|a|b|c|d|e|f|A|B|C|D|E|F


ziffer:
        Ø|nicht_null_ziffer


nicht_null_ziffer:
        1|2|3|4|5|6|7|8|9
```

```
oktalziffer:
       Ø|1|2|3|4|5|6|7


buchstabe:
       a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
     | A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_


sonderzeichen:
       !|?|(|)|[|]|{|}|+|-|=|<|>|/|||.|,|:|;|*|&|#|%|¤|^|~|ß


nichtgrafisches_zeichen:
       \n|\t|\b|\r|\f|\\|\'|\Ø|\"|\oktalziffer[oktalziffer[oktalziffer]]


apostroph:
       '


anf‡hrungszeichen:
       "
```

# Anhang B. Standardbibliothek

Dieser Anhang enthält eine nahezu vollständige Übersicht über die Funktionen und Makros der Standardbibliothek. Dabei beziehen wir uns auf den im Teil 3 von /50/ definierten Umfang. Wenn auch einige existierende Compilersysteme im Detail davon abweichen, sollte diese Standardempfehlung als Richtlinie für zukünftige Compilerentwicklungen dienen. Da die Funktionen der Standardbibliothek zum grössten Teil selbst in C implementiert wurden, sind Erweiterungen und Portierungen aus anderen Compilersystemen recht einfach möglich.

Die Beschreibung der Funktionen und Makros erfolgt in gekürzter Form. Insbesondere bei Funktionen, die aus unserer Sicht selten benötigt werden bzw. die dem System sehr nahe sind (z.B. Funktionen zur Passwort-Verarbeitung), geben wir nur die äussere Schnittstelle an.

**abort**

Erzeugen eines anormalen Programmabbruchs

```
int abort()
```

Die Funktion **abort** bewirkt zunächst das Schliessen aller Files und dann das Senden des Signals SIGABRT an den Prozess. Dadurch werden implementationsabhängige Routinen für den anormalen Programmabbruch aktiviert, die z.B. ein File mit einem Speicherabzug erzeugen.

Das Signal SIGABRT kann nicht ignoriert werden.

**abs**

Absoluter Betrag einer Integerzahl

```
int abs(i)
    int i;
```

Die Funktion **abs** liefert den Absolutbetrag seines Integerarguments i.

**assert**

Programminstrumentierung

```
#include <assert.h>

void assert(expression)
    int expression;
```

Der Makro **assert** kann zur Erzeugung von Testnachrichten benutzt werden. Wenn das Argument expression den Wert Null besitzt, wird folgende Nachricht im Standardfile stderr erzeugt:

"Assertion failed: expression, file <xyz>, line <nnn>"

Für <xyz> wird der Name des Quellfiles und für <nnn> die aktuelle Zeilennummer eingesetzt.

Eine Definition des Bezeichners NDEBUG (entweder beim Aufruf des Compilers mit dem Aufrufparameter -DNDEBUG oder durch Einfügen einer Präprozessorzeile "**#define** NDEBUG" vor der Zeile "**#include** <assert.h>") hat zur Folge, dass der Kode für assert nicht generiert wird.

**Bessel - jØ, j1, jn, yØ, y1, yn**

Bessel-Funktionen

```
#include  <math.h>

double jØ(x)
    double x;

double j1(x)
    double x;

double jn(n, x)
    int n;
    double x;

double yØ(x)
    double x;

double y1(x)
    double x;

double yn(n, x)
    int n;
    double x;
```

Diese Funktionen berechnen Bessel-Funktionen der 1. bzw. 2. Art der Ordnung Ø, 1 oder n für reelle Argumente x.

**bsearch**

Binäres Suchen in einer sortierten Tabelle

```
#include  <search.h>

char *bsearch(key, base, nel, width, compar)
    char *key;
    char *base;
    unsigned nel, width;
    int (*compar)();
```

Die Funktion **bsearch** sucht in einer aufsteigend sortierten Tabelle, die bei base beginnt und nel Elemente der Länge width enthält, nach dem durch key definierten Element. Die Vergleichsfunktion, auf die compar zeigt, muss der Anwender bereitstellen, wobei als Argumente zwei Zeiger auf die zu vergleichenden Tabellenelemente übergeben werden. Als Funktionswert muss ein negativer, positiver bzw. der Wert Null zurückgegeben werden, der anzeigt, dass das erste Argument als kleiner, als grösser bzw. als gleich gegenüber dem zweiten betrachtet wird.

**clock**

Ermitteln der benutzten CPU-Zeit

```
long clock()
```

Die Funktion **clock** liefert die seit dem ersten Aufruf von clock verbrauchte CPU-Zeit, gemessen in Mikrosekunden. Die ermittelte Zeit ist die Summe aus Nutzer- und Systemzeit des Prozesses und der beendeten Kindprozesse, für die die Systemrufe wait oder system gegeben wurden.

**conv - toupper, tolower, _toupper, _tolower, toascii**

Übersetzen von Zeichen

```
#include  <ctype.h>

int toupper(c)
    int c;

int tolower(c)
    int c;

int _toupper(c)
    int c;

int _tolower(c)
    int c;

int toascii(c)
    int c;
```

Die Argumente der Funktionen **toupper** und **tolower** können Integerwerte aus dem Bereich von -1 bis 255 annehmen. Wenn das Argument von toupper ein Kleinbuchstabe ist, so ist das Ergebnis der zugehörige Grossbuchstabe. Für tolower gilt das in umgekehrter Richtung. Für alle anderen Werte des Definitionsbereiches wird der Wert unverändert zurückgegeben.

Die Makros **_toupper** bzw. **_tolower** realisieren das gleiche wie toupper bzw. tolower, jedoch mit eingeschränktem Wertebereich. Der Makro _toupper (_tolower) verlangt als Argument einen Kleinbuchstaben (Grossbuchstaben). Andere Werte führen zu undefinierten Ergebnissen. Die Laufzeit der Makros ist geringer als die der Funktionen.

Die Funktion **toascii** löscht alle Bits des Arguments c, die nicht zum ASCII-Zeichensatz gehören, und gibt diesen Wert zurück.

**crypt, setkey, encrypt**

Generieren der Verschlüsselung nach dem NBS-Data-Encryption-Standard (DES)

```
char *crypt(key, salt)
    char *key, *salt;

void setkey(key)
    char *key;

void encrypt(block, edflag)
    char *block;
    int edflag;
```

**ctermid**

Generieren des Pfadnamens für das steuernde Terminal

```
#include  <stdio.h>

char *ctermid(s)
    char *s;
```

**ctime, localtime, gmtime, asctime, tzset, timezone, daylight, tzname**

Konvertieren von Datum und Zeit in eine Zeichenkette

```
#include  <time.h>
#include  <sys/types.h>

char *ctime(clock)
    long *clock;

struct tm *localtime(clock)
    long *clock;

struct tm *gmtime(clock)
    long *clock;

char *asctime(tm)
    struct tm *tm;

extern long timezone;
extern int daylight;
extern char *tzname[2];

void tzset()
```

Die Funktion **ctime** konvertiert einen **long**-Wert (auf den clock zeigt) in eine Zeichenkette der Form "Sun Sep 28 07:12:45 1986\n". Die Zeitangabe clock ist ein Wert in Sekunden seit dem 1.1.1970 00:00:00. Die Funktion liefert die Adresse dieser Zeichenkette.

Die Funktion **localtime** führt Korrekturen entsprechend der Zeitzone und ggf. für die Sommerzeit durch.

Die Funktion **gmtime** konvertiert direkt in Greenwich-Mean-Time (GMT), die auch als Systemzeit benutzt wird. Beide Funktionen liefern einen Zeiger auf die Struktur tm, die in time.h deklariert ist.

Die Funktion **asctime** konvertiert eine Struktur tm in eine Zeichenkette der oben angegebenen Form und liefert einen Zeiger auf diese Zeichenkette.

Die Deklarationen der Funktionen, externen Variablen und der Struktur tm sind im File time.h enthalten. Die Funktionsergebnisse zeigen auf Bereiche, die bei einem erneuten Aufruf überschrieben werden.

**ctype - isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii**

Klassifizierung von Zeichen

```
#include  <ctype.h>

int isalpha(c)
    int c;

int isupper(c)
    int c;

int islower(c)
    int c;

int isdigit(c)
    int c;

int isxdigit(c)
    int c;

int isalnum(c)
    int c;

int isspace(c)
    int c;

int ispunct(c)
    int c;

int isprint(c)
    int c;

int isgraph(c)
    int c;

int iscntrl(c)
    int c;

int isascii(c)
    int c;
```

Diese Makros dienen zur Zeichenklassifikation. Das Ergebnis ist ein Wert ungleich Null für TRUE und gleich Null für FALSE. Der Makro **isascii** ist für alle Integerwerte definiert. Die anderen Makros können nur auf EOF sowie auf Werte angewendet werden, für die isascii TRUE liefert. Liegt der Argumentwert nicht im festgelegten Wertebereich, so ist das Ergebnis undefiniert.

**isalpha** c ist ein Buchstabe.
**isupper** c ist ein Grossbuchstabe.
**islower** c ist ein Kleinbuchstabe.
**isdigit** c ist eine Ziffer [Ø-9].
**isxdigit** c ist eine Hexadezimalziffer [Ø-9], [A-F] oder [a-f].
**isalnum** c ist ein alphanumerisches Zeichen (Buchstabe oder Ziffer).
**isspace** c ist eines der Zeichen ' ', '\t', '\r', '\n', '\v', '\f'.
**ispunct** c ist ein Interpunktionszeichen (weder ein Steuerzeichen noch ein alphanumerisches Zeichen).
**isprint** c ist ein druckbares Zeichen (Øx2Ø <= c <= Øx7e).
**isgraph** c ist ein druckbares Zeichen (jedoch kein Leerzeichen).
**iscntrl** c ist das DEL-Zeichen (Øx7f) oder ein Steuerzeichen (c < Øx2Ø).
**isascii** c ist ein ASCII-Zeichen (ØxØØ <= c <= Øx7f).

**cuserid**

Bereitstellen des login-Namens des Nutzers

```
#include  <stdio.h>

char *cuserid(s)
    char *s;
```

**drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48**

Generieren gleichverteilter Zufallszahlen

```
double drand48()

double erand48(xsubi)
    unsigned short xsubi[3];

long lrand48()

long nrand48(xsubi)
    unsigned short xsubi[3];

long mrand48()

long jrand48(xsubi)
    unsigned short xsubi[3];

void srand48(seedval)
    long seedval;

unsigned short *seed48(seed16v)
    unsigned short seed16v[3];

void lcong48(param)
    unsigned short param[7];
```

**end, etext, edata**

Namen von Programmabschnitten

```
extern end;
extern etext;
extern edata;
```

**end** Die Adresse von end ist die erste Hauptspeicheradresse nach dem Programmtext und den nichtinitialisierten Daten.

**etext** Die Adresse von etext ist die Adresse nach dem Programmtext.

**edata** Die Adresse von edata ist die Adresse nach den initialisierten Daten.

**ecvt, fcvt, gcvt**

Konvertieren von Realzahlen in Zeichenketten

```
char *ecvt(value, ndigit, decpt, sign)
    double value;
    int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
    double value;
    int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
    double value;
    int ndigit;
    char *buf;
```

**erf, erfc**

Fehlerfunktionen

```
#include  <math.h>

double erf(x)
    double x;

double erfc(x)
    double x;
```

**exp, log, log1Ø, pow, sqrt**

Exponential-, Logarithmus-, Potenz- und Quadratwurzelfunktion

```
#include  <math.h>

double exp(x)
    double x;

double log(x)
    double x;

double log1Ø(x)
    double x;

double pow(x, y)
    double x, y;

double sqrt(x)
    double x;
```

**exp** berechnet $e^x$.
**log** berechnet den natürlichen Logarithmus von x.
**log1Ø** berechnet den Logarithmus zur Basis 1Ø.
**pow** berechnet $x^y$.
**sqrt** berechnet die Quadratwurzel von x.

**fclose, fflush**

Schliessen von Files und Puffer ausgeben

```
#include <stdio.h>

int fclose(file_p)
    FILE *file_p;

int fflush(file_p)
    FILE *file_p;
```

Die Funktion **fclose** bewirkt das Schliessen des Files file_p. Dabei werden alle zugehörigen Puffer geleert und freigegeben. Die Beendigung der Programmausführung durch den Aufruf von exit hat zur Folge, dass alle geöffneten Files mittels fclose geschlossen werden.

Die Funktion **fflush** bewirkt, dass alle gepufferten Ausgabedaten in das File file_p geschrieben werden. Das File bleibt geöffnet.

Der Funktionswert Null zeigt den ordnungsgemässen Verlauf der Funktionen an. Im Fehlerfall wird EOF geliefert.

**ferror, feof, clearerr, fileno**

Statusabfragen für geöffnete Files

```
#include <stdio.h>

int ferror(file_p)
    FILE *file_p;

int feof(file_p)
    FILE *file_p;

void clearerr(file_p)
    FILE *file_p;

int fileno(file_p)
    FILE *file_p;
```

Der Makro **ferror** gibt einen Wert ungleich Null zurück, wenn beim Lesen oder Schreiben des Files file_p ein E/A-Fehler aufgetreten war, sonst wird Null zurückgegeben. Die Fehleranzeige bleibt bis zum Schliessen des Files bzw. bis zum Aufruf von clearerr erhalten.

Der Makro **feof** liefert einen Wert ungleich Null, wenn beim Lesen des Files file_p die EOF-Bedingung erkannt wurde, andernfalls wird Null zurückgegeben.

Der Makro **clearerr** setzt die Fehler- und EOF-Anzeige für das File file_p zurück.

Der Makro **fileno** liefert den Filedeskriptor des Files file_p.

**floor, ceil, fmod, fabs**

Ganzzahliger Anteil, Divisionsrest, absoluter Betrag

```
#include <math.h>

double floor(x)
    double x;

double ceil(x)
    double x;

double fmod(x, y)
    double x, y;

double fabs(x)
    double x;
```

**floor** berechnet die grösste ganze Zahl, die nicht grösser als x ist.
**ceil** berechnet die kleinste ganze Zahl, die nicht kleiner als x ist.
**fmod** berechnet den Gleitkommarest der Division von x durch y.
**fabs** berechnet den absoluten Betrag von x.

**fopen, freopen, fdopen**

Öffnen eines Files

```
#include <stdio.h>

FILE *fopen(file_name, type)
    char *file_name, *type;

FILE *freopen(file_name, type, file_p)
    char *file_name, *type;
    FILE *file_p;

FILE *fdopen(fildes, type)
    int fildes;
    char *type;
```

Die Funktion **fopen** öffnet das durch file_name benannte File und liefert einen Filepointer, d.h. einen Zeiger auf die Struktur FILE, der diesem File im weiteren zugeordnet bleibt. Die Zeichenkette type spezifiziert die Art des Filezugriffs in folgender Weise:

"r" Öffnen für Lesen.
"w" Öffnen für Schreiben, wobei ein bereits existierendes File gelöscht wird. Andernfalls wird das File erzeugt.
"a" Öffnen für Fortschreiben eines existierenden Files bzw. Erzeugen des Files.
"r+" Öffnen für "update", d.h. für Lesen und Schreiben.
"w+" Öffnen für "update", ein existierendes File wird gelöscht.
"a+" Öffnen für "update". Ein existierendes File wird fortgeschrieben, andernfalls wird es erzeugt.

Die Funktion **freopen** ersetzt file_p durch das File file_name. Vorher wird file_p geschlossen. Die Funktion liefert einen Filepointer. Oft wird freopen benutzt, um stdin, stdout und stderr anderen Files zuzuordnen.

Die Funktion **fdopen** ordnet dem Filedeskriptor fildes eine Struktur vom Typ FILE zu und liefert als Funktionswert die Adresse dieser Struktur, d.h. den Filepointer. Der Filedeskriptor kann durch die Systemrufe open, create oder dup erzeugt worden sein. Der analog zu fopen mögliche Parameter type muss mit dem Zugriffsmodus des geöffneten Files (read, write, update) harmonieren.

**fread, fwrite**

Binäre Eingabe bzw. Ausgabe

```
#include  <stdio.h>
#include  <sys/types.h>

int fread(ptr, size, nitems, file_p)
    char *ptr;
    size_t size;
    int nitems;
    FILE *file_p;

int fwrite(ptr, size, nitems, file_p)
    char *ptr;
    size_t size;
    int nitems;
    FILE *file_p;
```

Die Funktion **fread** liest nitems Dateneinheiten aus dem File file_p in ein Feld ptr. Unter einer Dateneinheit wird dabei eine Folge von Bytes mit der Länge size verstanden, die nicht durch \Ø abgeschlossen sein muss. Die Funktion fread beendet ihre Arbeit, wenn nitems Dateneinheiten gelesen wurden bzw. eine EOF- oder Fehlerbedingung aufgetreten ist. Der Inhalt des Files wird durch fread nicht verändert.

Die Funktion **fwrite** schreibt nitems Dateneinheiten des durch ptr adressierten Feldes in das File file_p. Bei Auftreten einer Fehlerbedingung beendet sie ihre Arbeit vorzeitig.

Als Funktionswert liefern fread bzw. fwrite die Anzahl der gelesenen bzw. geschriebenen Dateneinheiten.

**frexp, ldexp, modf**

Manipulieren der Bestandteile einer Realzahl

```
double frexp(value, eptr)
    double value;
    int *eptr;

double ldexp(value, exp)
    double value;
    int exp;

double modf(value, iptr)
    double value, *iptr;
```

Die Funktion **frexp** berechnet die Mantisse x (x<1.Ø) des Wertes value (value=x*$2^n$). Der Integerexponent n wird indirekt nach *eptr gespeichert. Die Funktion **ldexp** berechnet den Wert value*$2^{exp}$, und **modf** ermittelt den gebrochenen Teil von value. Der ganze Anteil wird nach *iptr gespeichert.

**fseek, rewind, ftell**

Positionieroperationen in einem File

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

int fseek(file_p, offset, ptrname)
    FILE *file_p;
    long offset;
    int ptrname;

void rewind(file_p)
    FILE *file_p;

long ftell(file_p)
    FILE *file_p;
```

Die Funktion **fseek** legt fest, ab welcher Byteposition im File file_p die nächste Ein- oder Ausgabe erfolgen soll. Sie ergibt sich aus dem Wert von offset ausgehend vom Anfang des Files (bei ptrname gleich SEEK_SET), ausgehend von der aktuellen Position (bei ptrname gleich SEEK_CUR) oder ausgehend vom Fileende (ptrname gleich SEEK_END). Bei nichterlaubten Positionieroperationen wird ein Funktionsergebnis ungleich Null zurückgegeben.

Die Wirkung des Funktionsaufrufs **rewind**(file_p) entspricht der Wirkung von fseek(file_p, ØL, SEEK_SET).

Die Funktionen fseek und rewind heben die Wirkung eines vorhergehenden Aufrufs von ungetc auf. Files, die für "update" (s. fopen) geöffnet wurden, können nach fseek oder rewind sowohl gelesen als auch geschrieben werden.

Die Funktion **ftell** liefert die aktuelle Byteposition im File file_p relativ zum Fileanfang (d.h. offset).

**ftw**

Durchmustern eines Filesystems

```
#include  <ftw.h>

int ftw(path, fn, depth)
    char *path;
    int (*fn)(), depth;
```

Die Funktion **ftw** durchmustert die in path beginnende Verzeichnishierarchie und ruft für jedes File die durch fn adressierte Funktion auf, wobei als Funktionsargumente ein Zeiger auf einen Filenamen, ein Zeiger auf eine stat-Struktur sowie ein **int**-Wert übergeben werden. Die möglichen **int**-Werte sind in ftw.h definiert:

| | |
|---|---|
| FTW_F | File |
| FTW_D | Verzeichnis |
| FTW_DNR | Verzeichnis, das nicht gelesen werden kann |
| FTW_NS | stat konnte nicht erfolgreich ausgeführt werden. |

Zur Effektivierung kann man die Verzeichnistiefe in depth bzw. Null angeben, dabei darf depth nicht grösser als die Anzahl der noch verfügbaren Filedeskriptoren sein.

**gamma, signgam**

Logarithmus-Gammafunktion

```
#include  <math.h>

double gamma(x)
    double x;

extern int signgam;
```

**getc, getchar, fgetc, getw**

Lesen des nächsten Zeichens oder Wortes

```
#include  <stdio.h>

int getc(file_p)
    FILE *file_p;

int getchar()

int fgetc(file_p)
    FILE *file_p;

int getw(file_p)
    FILE *file_p;
```

Der Makro **getc** stellt das nächste Zeichen des Files file_p bereit. Der Makro **getchar** ist als getc(stdin) definiert.

Die Funktion **fgetc** ist in ihrer Anwendung analog zu getc. Auf Grund des Funktionsaufrufes ist ihre Laufzeit jedoch grösser. Andererseits wird je Aufruf weniger Speicherplatz benötigt.

Die Funktion **getw** liefert das nächste Wort des Files file_p. Die Wortbreite entspricht dem Speicherplatzbedarf einer **int**-Grösse und kann zwischen verschiedenen Maschinen variieren. Für den Aufruf von getw ist keine spezielle Ausrichtung im File erforderlich.

**getcwd**

Ermitteln des Pfadnamens des aktuellen Verzeichnisses

```
char *getcwd(buf, size)
    char *buf;
    int size;
```

Die Funktion **getcwd** ermittelt den Pfadnamen des aktuellen Verzeichnisses und speichert diese Zeichenkette nach buf. Die Zahl size gibt die Grösse des dafür vorgesehenen Speicherbereiches an. Sie muss mindestens um 2 grösser sein als die tatsächlich benötigte Länge. Wird für buf der Zeigerwert NULL angegeben, so fordert die Funktion den benötigten Speicherplatz (size) mit der Funktion malloc an. Als Ergebnis wird immer der Zeiger auf den Pfadnamen zurückgegeben. Der Funktionswert NULL zeigt eine Fehlerbedingung an.

**getenv**

Ermitteln des Wertes für einen Umgebungsparameter

```
char *getenv(name)
    char *name;
```

Die Funktion **getenv** sucht in der Umgebungsliste (environment list) nach einer Zeichenkette der Form "name=value" und liefert als Ergebnis die Adresse von value, falls eine solche Zeichenkette existiert. Andernfalls wird der Zeigerwert NULL zurückgegeben.

**getlogin**

Ermitteln des login-Kennzeichens aus dem File /etc/utmp

```
char *getlogin()
```

**getgrent, getgrgid, getgrnam, setgrent, endgrent**

Lesen von Eintragungen im File /etc/group

```
#include  <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
    int gid;

struct group *getgrnam(name)
    char *name;

int setgrent()

int endgrend()
```

**getopt**

Ermitteln einer Flag-Angabe im Argumentvektor

```
int getopt(argc, argv, optstring)
    int argc;
    char *argv[], *optstring;

extern char *optarg;
extern int optind, opterr;
```

Die Funktion **getopt** kann zur Analyse einer Kommandozeile verwendet werden. Als Ergebnis wird das nächste Flag aus argv bereitgestellt, dass in optstring enthalten ist. Die Zeichenkette optstring muss alle erlaubten Flag-Buchstaben enthalten. Folgt dem Buchstaben ein Doppelpunkt, so wird nach diesem Flag ein Aufrufparameter erwartet, der mit oder ohne Trennzeichen notiert sein kann. Der externe Zeiger optarg wird auf den Beginn der entsprechenden Zeichenfolge gesetzt.

Die externe Variable optind enthält den Index des nächsten zu verarbeitenden Aufrufparameters in argv. Vor dem ersten Aufruf von getopt wird sie mit dem Wert 1 initialisiert.

Wurden alle Flags verarbeitet, so liefert getopt EOF. Das reservierte Flag -- kann verwendet werden, um das Ende der Flags anzuzeigen. Die Funktion getopt ignoriert dieses Flag und liefert EOF.

Wird ein Flag-Buchstabe gefunden, der nicht in optstring enthalten ist, so schreibt getopt eine Fehlernachricht in das Standardfile stderr und liefert das Zeichen '?'. Setzt man die externe Variable opterr auf Ø, wird die Fehlernachricht unterdrückt.

**getpass**

Lesen eines Passwortes vom File /dev/tty

```
char *getpass(prompt)
    char *prompt;
```

**getpw**

Ermitteln des zur Nutzeridentifikation gehörenden Eintrags in /etc/passwd

```
void getpw(uid, buf)
    int uid;
    char *buf;
```

**getpwent, getpwuid, getpwnam, setpwent, endpwent**

Lesen eines Eintrags im File /etc/passwd

```
#include  <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
    int uid;

struct passwd *getpwnam(name)
    char *name;

int setpwent()

int endpwent()
```

**gets, fgets**

Lesen einer Zeichenkette aus einem File

```
#include  <stdio.h>

char *gets(s)
    char *s;

char *fgets(s, n, file_p)
    char *s;
    int n;
    FILE *file_p;
```

Die Funktion **gets** liest Zeichen aus der Standardeingabe (stdin) und überträgt sie in das Feld s, bis Newline gelesen oder die EOF-Bedingung erkannt wird. Das Newline wird beseitigt und die Zeichenkette mit \Ø abgeschlossen.

Die Funktion **fgets** liest Zeichen aus dem File file_p und überträgt sie in das durch s adressierte Feld, bis entweder n-1 Zeichen bzw. Newline gelesen oder die EOF-Bedingung erkannt wurde. Die Zeichenkette wird mit \Ø abgeschlossen.

Normalerweise liefern beide Funktionen als Ergebnis den Wert des Zeigers s. Bei Erkennen der EOF-Bedingung oder E/A-Fehlern wird der Zeigerwert NULL zurückgegeben.

**getutent, getutid, getutline, pututline, setutent, endutent, utmpname**

Zugriff auf das File /etc/utmp

```
#include  <utmp.h>

struct utmp *getutent()

struct utmp *getutid(id)
    struct utmp *id;

struct utmp *getutline(line)
    struct utmp *line;

void pututline(utmp)
    struct utmp *utmp;

void setutent()

void endutent()

void utmpname(file)
    char *file;
```

**hsearch, hcreate, hdestroy**

Verwalten von Hash-Tabellen

```
#include  <search.h>

ENTRY *hsearch(item, action)
    ENTRY item;
    ACTION action;

int hcreate(nel)
    unsigned nel;

void hdestroy()
```

Die Funktion **hsearch** verwaltet eine Hash-Tabelle und liefert als Ergebnis einen Zeiger auf einen gültigen Eintrag oder NULL. Dabei stellt item eine Strukturinstanz vom Typ ENTRY dar. Sie enthält die Zeiger key und data, die den Suchbegriff und die Daten eines Eintrags charakterisieren. Für action kann ENTER bzw. FIND (s. File search.h) angegeben werden, um einen Eintrag einzufügen bzw. zu suchen.

Mit der Funktion **hcreate** wird eine Hash-Tabelle initialisiert und mit **hdestroy** wieder freigegeben.

**hypot**

Euklidische Distanz

```
#include  <math.h>

double hypot(x, y)
    double x;
    double y;
```

Die Funktion **hypot** berechnet den Wert sqrt(x*x + y*y).

**l3tol, ltol3**

Konvertierung zwischen 3-Byte-Integerwerten und **long**-Integerwerten

```
void l3tol(lp, cp, n)
    long *lp;
    char *cp;
    int n;

void ltol3(cp, lp, n)
    char *cp;
    long *lp;
    int n;
```

Die Funktion **l3tol** konvertiert eine Liste von n 3-Byte-Integerwerten, die im Zeichenfeld cp untergebracht sind, in ein Feld lp von Integerwerten des Typs **long**.

Die Funktion **ltol3** führt die Konvertierung in umgekehrter Richtung durch.

**lockf**

Sperren von Teilen eines Files zur exklusiven Nutzung

```
#include  <unistd.h>

int lockf(fildes, function, size)
    long size;
    int fildes;
    int function;
```

**logname**

Bereitstellen des login-Namens des Nutzers

```
char *logname()
```

**lsearch, lfind**

Lineares Suchen und Hinzufügen

```
#include  <search.h>

char *lsearch(key, base, nelp, width, compar)
    char *key;
    char *base;
    unsigned *nelp;
    unsigned *width;
    int (*compar)();

char *lfind(key, base, nelp, width,  compar)
    char *key;
    char *base;
    unsigned nelp;
    unsigned width;
    int (*compar)();
```

Die Funktionen **lsearch** und **lfind** realisieren die lineare Suche in einer Tabelle, die durch base adressiert wird, *nelp Elemente enthält und deren Elementelänge *width beträgt. Der Zeiger key zeigt auf ein Datenelement, nach dem in der Tabelle gesucht werden soll. Wird ein solcher Eintrag gefunden, so liefern die Funktionen als Ergebnis die Adresse des betreffenden Eintrags. Wird der Eintrag nicht gefunden, so wird die Tabelle bei lsearch um diesen Eintrag erweitert und der Zähler *nelp erhöht. Die Funktion lfind liefert in diesem Fall den Zeigerwert NULL.

Die Vergleichsfunktion *compar ist vom Nutzer bereitzustellen. Sie wird mit zwei Argumenten aufgerufen, die auf die zu vergleichenden Datenelemente zeigen. Bei Gleichheit muss die Funktion den Wert Null liefern.

**malloc, free, realloc, calloc, mallopt, mallinfo**

Dynamische Speicherverwaltung

```
#include  <malloc.h>

char *malloc(size)
    unsigned size;

void free(ptr)
    char *ptr;

char *realloc(ptr, size)
    char *ptr;
    unsigned size;

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

int mallopt(cmd, value)
    int cmd;
    int value;

struct mallinfo mallinfo()
```

Die Funktion **malloc** liefert einen Zeiger auf einen Hauptspeicherbereich (Block) von mindestens size Bytes, der allen Ausrichtungsanforderungen genügt.

Mit der Funktion **free** kann man einen durch ptr adressierten Block wieder freigeben und für weitere Anforderungen verfügbar machen.

Der Versuch, auf eine Speicheradresse zuzugreifen, die hinter einem solchen Block liegt, bzw. der Aufruf von free mit einem falschen Zeigerwert führen zu undefinierten Ergebnissen.

Die Funktion **realloc** verändert die Grösse des durch ptr adressierten Blocks auf den neuen Wert size und liefert als Ergebnis die Adresse des möglicherweise verschobenen Blocks. Der Inhalt des Speicherbereichs bleibt dabei bis zum kleineren des alten oder neuen Wertes von size erhalten.

Die Funktion **calloc** ordnet Speicherplatz für nelem Elemente der Länge elsize zu und initialisiert diesen Bereich mit Null. Der Funktionswert ist ein Zeiger auf diesen Bereich.

Die Funktionen **mallopt** bzw. **mallinfo** ermöglichen die Modifikation der Speicherzuordungsalgorithmen bzw. die Kontrolle der Speicherplatznutzung. Für die Anwendung sind die Definitionen in malloc.h sowie Kenntnisse der verwendeten Algorithmen erforderlich.

**matherr**

Fehlerbehandlung für mathematische Funktionen

```
#include  <math.h>

int matherr(x)
    struct exception *x;
```

**memory - memccpy, memchr, memcmp, memcpy, memset**

Speicheroperationen

```
#include <memory.h>

char *memccpy(s1, s2, c, n)
    char *s1;
    char *s2;
    int c;
    int n;

char *memchr(s, c, n)
    char *s;
    int c;
    int n;

int *memcmp(s1, s2, n)
    char *s1;
    char *s2;
    int n;

char *memcpy(s1, s2, n)
    char *s1;
    char *s2;
    int n;

char *memset(s, c, n)
    char *s;
    int c;
    int n;
```

Diese Funktionen realisieren Operationen mit Speicherbereichen so effektiv wie möglich, wobei \Ø nicht beachtet und Speicherbereichsgrenzen nicht überprüft werden. Die Funktion **memccpy** kopiert den Bereich s2 nach s1 bis zum ersten Auftreten des Zeichens c, maximal jedoch n Zeichen. Der Funktionswert ist die Adresse von c in s1 oder NULL.

Die Funktion **memchr** sucht im Feld s nach dem Zeichen c und liefert die Adresse dieses Zeichens oder NULL, falls c unter den ersten n Zeichen nicht gefunden wurde.

Die Funktion **memcmp** vergleicht die ersten n Elemente der Felder s1 und s2. Der Funktionswert ist kleiner, gleich oder grösser als Null, je nachdem, ob s1 im lexikographischen Sinne kleiner, gleich oder grösser als s2 ist.

Die Funktion **memcpy** kopiert n Zeichen von s2 nach s1 und liefert s1 als Funktionswert.

Die Funktion **memset** initialisiert die ersten n Zeichen von s mit dem Zeichen c. Der Funktionswert ist s.

**mktemp**

Erzeugen eines eindeutigen Filenamens

```
char *mktemp(template)
    char *template;
```

Die Funktion **mktemp** ersetzt im Filenamenmuster template die letzten 6 Zeichen XXXXXX durch einen Buchstaben und die aktuelle Prozessnummer. Im Fehlerfall gibt die Funktion den Zeigerwert NULL zurück.

**monitor**

Vorbereiten der Laufzeitüberwachung (Systemruf profile)

```
#include <mon.h>

void monitor(lowpc, highpc, buffer, bufsize, nfunc)
    int (*lowpc)(), (*highpc)();
    WORD *buffer;
    int bufsize, nfunc;
```

**perror, errno, sys_errlist, sys_nerr**

Fehlermitteilungen des Systems

```
void perror(s)
    char *s;

extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;
```

Die Funktion **perror** erzeugt eine Fehlernachricht im Standardfile stderr, die den zuletzt aufgetretenen Fehler bei einem Systemruf oder in einer Bibliotheksfunktion beschreibt. Die Fehlernachricht setzt sich aus der Zeichenkette s, einem Doppelpunkt, der Systemnachricht und dem Zeichen Newline zusammen. Die Fehlerursache ist als Integerzahl verschlüsselt in der externen Variablen errno enthalten, die bei jedem auftretenden Fehler gesetzt wird.

Für den Aufbau eigener Fehlertexte kann die Systemnachricht dem externen Feld sys_errlist entnommen werden, wobei errno als Index dient. Die externe Variable sys_nerr enthält den Index der letzten verfügbaren Nachricht.

**popen, pclose**

Initiieren einer Pipe

```
#include <stdio.h>

FILE *popen(command, type)
    char *command, *type;

int pclose(file_p)
    FILE *file_p)
```

Die Funktion **popen** erzeugt eine Pipe zwischen dem rufenden Prozess und dem durch die Kommandozeile command erzeugten Kindprozess. Dabei kann mit der Zeichenkette type das Lesen ("r") oder das Schreiben ("w") vereinbart werden. Als Funktionswert liefert popen einen Filepointer file_p. Das ermöglicht dem rufenden Prozess das Schreiben in die Standardeingabe von command bzw. das Lesen von der Standardausgabe von command.

Eine durch popen geöffnete Pipe sollte mit der Funktion **pclose** geschlossen werden. Dadurch wird auf die Beendigung des Prozesses gewartet und als Funktionswert der exit-Status zurückgegeben.

**printf, fprintf, sprintf**

Formatierte Ausgabe

```
#include <stdio.h>

int printf(format [ , arg ] ...)
    char *format;

int fprintf(file_p, format [ , arg ] ...)
    FILE *file_p;
    char *format;

int sprintf(s, format [ , arg ] ...)
    char *s;
    char *format;
```

Die Funktion **printf** schreibt in das Standardfile stdout, **fprintf** in das File file_p und **sprintf** schreibt in den durch s adressierten Hauptspeicherbereich, wobei die erzeugte Zeichenkette mit \Ø abgeschlossen wird. Der Nutzer ist verantwortlich dafür, dass genügend Speicherplatz zur Verfügung steht. Alle drei Funktionen liefern als Ergebnis die Anzahl der übertragenen Zeichen oder einen negativen Wert im Falle eines Fehlers.

Jede Funktion konvertiert, formatiert und gibt ihre Argumente arg unter Steuerung von format aus. Dabei ist format eine Zeichenkette, die zwei Typen von Objekten enthält: einfache Zeichen, die lediglich in die Ausgabe kopiert werden, und Konvertierungsvorschriften, von denen jede Null oder mehr Argumente arg beansprucht. Das Ergebnis ist undefiniert, wenn nicht genügend Argumente arg für format vorhanden sind. Überzählige Argumente werden ignoriert.

Jede Konvertierungsvorschrift wird durch das Zeichen % eingeleitet und besitzt die allgemeine Form

```
%[<flags>][<feldbreite>].[<genauigkeit>][l]<konv_zeichen>
```

Nach dem % kann folgendes stehen:

- null oder mehr Flags <flags>, die die Bedeutung der Konvertierungsvorschrift modifizieren.
- eine wahlweise Dezimalziffernfolge <feldbreite>, die die minimale Feldbreite festlegt. Hat der konvertierte Wert weniger Zeichen als die Feldbreite, wird links aufgefüllt (oder rechts bei dem Flag '-' für Linksausrichtung).
- die Genauigkeit, die bei den Konvertierungszeichen d, o, u, x oder X die minimale Ziffernanzahl bzw. bei e und f die Ziffernanzahl nach dem Dezimalpunkt, bei dem Konvertierungszeichen g die maximale Anzahl signifikanter Ziffern und bei dem Konvertierungszeichen s die maximal auszugebende Zeichenzahl angibt. Die Genauigkeit ist in der Form .<genauigkeit> anzugeben, wobei <genauigkeit> eine Dezimalziffernfolge darstellt. Eine leere Ziffernfolge wird als Null betrachtet.
- wahlweise kann der Buchstabe l vor den Konvertierungszeichen d, o, u, x oder X angegeben werden, um anzuzeigen, dass das zugehörige Argument vom Typ **long int** ist. Vor anderen Konvertierungszeichen wird ein l ignoriert.
- ein Konvertierungszeichen <konv_zeichen>, das den Typ der Konvertierung festlegt.

Eine Feldbreiten- oder Genauigkeitsangabe kann statt durch eine Dezimalziffernfolge durch einen Stern (*) angezeigt werden. In diesem Fall wird die Feldbreite oder Genauigkeit durch ein Integerargument arg geliefert, das vor dem zu konvertierenden Argument angegeben werden muss.

Für <flags> können die Zeichen -, +, # sowie ein Leerzeichen (blank) stehen. Die einzelnen Zeichen haben folgende Bedeutung:

- Das Ergebnis der Konvertierung soll nach links ausgerichtet im Ausgabebereich erscheinen.

+ Das Ergebnis einer vorzeichenbehafteten Konvertierung beginnt stets mit einem Vorzeichen (+ oder -).

blank Falls das erste Zeichen einer vorzeichenbehafteten Konvertierung kein Vorzeichen ist, soll dem Ergebnis ein Leerzeichen (blank) vorangehen. Daraus folgt, falls sowohl das blank-Flag als auch das Flag + angegeben sind, wird das blank-Flag ignoriert.

# Dieses Flag gibt an, dass der Wert in eine "alternative Form" zu konvertieren ist. Für die Konvertierungszeichen c, d, s und u hat das Flag keine Auswirkung. Bei dem Konvertierungszeichen o kann damit die Ausgabe einer führenden Null erzwungen werden. Bei x oder X wird für Ergebnisse ungleich Null die Zeichenfolge Øx bzw. ØX vorangestellt. Für e, E, f, g und G erhält das Ergebnis stets einen Dezimalpunkt, auch wenn dem Punkt keine Ziffer folgt (normalerweise wird der Dezimalpunkt nur ausgegeben, wenn eine Ziffer folgt). Bei g und G werden abschliessende Nullen nicht aus dem Ergebnis entfernt, wie es sonst der Fall ist.

Die Konvertierungszeichen <konv_zeichen> haben folgende Bedeutung:

d,o,u,x,X Das entsprechende Integerargument arg wird in die vorzeichenbehaftete Dezimal-, vorzeichenlose Oktal-, Dezimal- bzw. Hexadezimaldarstellung konvertiert. Bei dem Konvertierungszeichen x werden die Buchstaben "abcdef" und bei X die Buchstaben "ABCDEF" verwendet. Die Genauigkeit gibt die minimale Anzahl von Ziffern an, wobei ggf. mit führenden Nullen aufgefüllt wird. Der Standardwert für <genauigkeit> ist 1. Die Konvertierung des Wertes Null mit der Genauigkeit Null ergibt die leere Zeichenkette.

f Das **float**- oder **double**-Argument arg wird in die dezimale Darstellung der Form "[-]ddd.ddd" konvertiert, wobei die Anzahl der Ziffern nach dem Dezimalpunkt der Genauigkeitsspezifikation entspricht. Fehlt diese Angabe, werden 6 Ziffern ausgegeben. Ist die Genauigkeit mit Null angegeben, so wird kein Dezimalpunkt ausgegeben.

e,E Das **float**- oder **double**-Argument arg wird in die Form "[-]d.ddde±dd" konvertiert, wobei vor dem Dezimalpunkt eine Ziffer ausgegeben wird. Die Anzahl der Ziffern nach dem Dezimalpunkt hängt von der Genauigkeitsangabe ab und beträgt standardmässig 6. Ist die Genauigkeit mit Null angegeben, so wird kein Dezimalpunkt ausgegeben. Das Konvertierungszeichen E erzeugt vor dem Exponenten ein E anstelle des e. Der Exponent besteht immer aus mindestens 2 Ziffern. Falls erforderlich, werden mehr Ziffern ausgegeben.

g,G Das **float**- oder **double**-Argument arg wird wie bei den Konvertierungszeichen f oder e (bzw. E bei Angabe von G) ausgegeben, wobei die Genauigkeit die Anzahl der signifikanten Ziffern angibt. Welche der beiden Formen benutzt wird, hängt vom zu konvertierenden Wert ab. Das Format e wird nur benutzt, wenn der daraus resultierende

Exponent kleiner als -4 oder grösser als die Genauigkeit ist. Abschliessende Nullen werden entfernt und der Dezimalpunkt nur ausgegeben, wenn ihm eine Ziffer folgt.

c Das zugehörige Argument arg wird als Zeichen ausgegeben.

s Das Argument arg wird als Zeiger auf eine Zeichenkette interpretiert. Es werden die Zeichen bis zum Zeichen \Ø ausgegeben oder so viele Zeichen, wie durch die Genauigkeit festgelegt ist. Fehlt die Genauigkeitsangabe, so werden alle Zeichen bis zum ersten \Ø ausgegeben. Ist der Argumentwert NULL, so sind die Ergebnisse undefiniert.

% Das Zeichen % wird ausgegeben. Es wird kein Argument bearbeitet.

Folgt dem Zeichen % kein gültiges Konvertierungszeichen, so sind die Ergebnisse nicht definiert.

Eine nicht angegebene oder zu kleine Feldbreite bewirkt in keinem Fall eine Verkleinerung des erforderlichen Ausgabebereichs. Ist das Ergebnis der Konvertierung grösser als der definierte Ausgabebereich, so wird dieser einfach um die erforderliche Stellenanzahl erweitert. Die mit printf oder fprintf erzeugten Zeichen werden so ausgegeben, als wäre putc benutzt worden.

**putc, putchar, fputc, putw**

Ausgabe eines Zeichens oder Wortes in ein File

```
#include <stdio.h>

int putc(c, file_p)
    int c;
    FILE *file_p;

int putchar(c)
    int c;

int fputc(c, file_p)
    int c;
    FILE *file_p;

int putw(w, file_p)
    int w;
    FILE *file_p;
```

Der Makro **putc** schreibt das Zeichen c in das File file_p, der Makro **putchar**(c) ist definiert als putc(c, stdout).

Die Funktion **fputc** verhält sich wie putc, arbeitet jedoch langsamer. Andererseits kann der Funktionsname als Argument an eine Funktion übergeben werden. Ausserdem benötigt fputc gegenüber putc je Aufruf weniger Speicherplatz.

Die Funktion **putw** schreibt das Wort w in das File file_p. Die Wortbreite entspricht dem Speicherbedarf eines **int**-Wertes und kann zwischen verschiedenen Maschinen variieren. Eine spezielle Ausrichtung im File ist nicht erforderlich.

Die Ausgabe erfolgt für alle Files ausser dem Standardfile stderr gepuffert. Bei Terminalausgaben erfolgt die zeilenweise Pufferung, d.h., die Ausgabe erfolgt erst, wenn ein Newline ausgegeben wird oder eine Eingabe vom Terminal erfolgen soll. Die Ausgabe in ein File geschieht gepuffert

(Pufferlänge BUFSIZ). Mit den Funktionen freopen und setbuf kann die Standardpufferzuordnung verändert werden.

Bei erfolgreicher Ausführung liefern die Funktionen den Wert des ausgegebenen Zeichens bzw. Wortes. Im Fehlerfall wird EOF zurückgegeben. Für putw kann daher zusätzlich die Benutzung von ferror erforderlich sein.

**putenv**

Ändern oder Hinzufügen eines Wertes in die Umgebungsliste

```
int putenv(string)
    char *string;
```

Die Funktion **putenv** benötigt als Argument eine Zeichenkette der Form "name=value". Auf diese Weise kann in der Umgebungsliste eine existierende Variable name geändert bzw. ein Eintrag hinzugefügt werden. Dabei ist zu beachten, dass die Zeichenkette selbst Bestandteil der Umgebungsliste wird. Die Zeichenkette sollte also nicht verändert werden. Die Anwendung von putenv steht in Beziehung zur Funktion getenv.

**putpwent**

Schreiben eines Eintrags in das File /etc/passwd

```
#include  <pwd.h>

int putpwent(p, f)
    struct passwd *p;
    FILE *f;
```

**puts, fputs**

Ausgabe einer Zeichenkette in ein File

```
#include  <stdio.h>

int puts(s)
    char *s;

int fputs(s, file_p)
    char *s;
    FILE *file_p;
```

Die Funktion **puts** schreibt die mit \Ø abgeschlossene Zeichenkette s in das File stdout. Das Zeichen \Ø wird im Ausgabefile durch Newline ersetzt.

Die Funktion **fputs** gibt die durch s adresssierte und mit \Ø abgeschlossene Zeichenkette in das File file_p aus. Dabei wird \Ø nicht mit ausgegeben.

Beide Funktionen liefern im Falle eines Fehlers EOF.

**qsort**

Sortierfunktion Quicksort

```
void qsort(base, nel, width, compar)
    char *base;
    unsigned nel, width;
    int (*compar)();
```

Die Funktion **qsort** sortiert nach dem Quicksort-Algorithmus. Das Argument base zeigt auf das erste Element, nel gibt die Elementezahl und width die Elementlänge in Bytes an. Das Argument compar ist ein Zeiger auf eine vom Nutzer bereitzustellende Vergleichsfunktion, der zwei Zeiger auf die zu vergleichenden Elemente übergeben werden. Die Vergleichsfunktion muss einen Wert kleiner, gleich oder grösser Null zurückgeben, um anzuzeigen, dass das erste Element kleiner, gleich oder grösser als das zweite ist.

**rand, srand**

Einfacher Zufallszahlengenerator

```
int rand()

void srand(seed)
    unsigned seed;
```

Die Funktion **rand** liefert Zufallszahlen im Bereich von Ø bis $2^{15}-1$. Der Startpunkt des Algorithmus kann jederzeit mit der Funktion **srand** verändert werden. Er ist zu Beginn mit 1 initialisiert.

**regcmp, regex**

Übersetzen und Verarbeiten regulärer Ausdrücke

```
char *regcmp(string1 [ , string2, ... ], (char *)Ø)
    char *string1, *string2, ...;

char *regex(re, subject [ , retØ, ... ])
    char *re, *subject, *retØ, ...;

extern char *__loc1;
```

**scanf, fscanf, sscanf**

Formatierte Eingabe

```
#include <stdio.h>

int scanf(format [ , pointer ] ...)
    char *format;

int fscanf(file_p, format [ , pointer ] ...)
    FILE *file_p;
    char *format;

int sscanf(s, format [ , pointer ] ...)
    char *s, *format;
```

Die Funktion **scanf** liest die Standardeingabe, **fscanf** das File file_p, und **sscanf** liest das durch s adressierte Zeichenfeld. Die gelesenen Zeichen werden entsprechend format konvertiert und in die zugehörigen Argumente gespeichert. Jede Funktion erwartet als Argumente eine Steuerzeichenkette format und eine Reihe von Zeigerargumenten pointer, die angeben, wohin die konvertierte Eingabe zu speichern ist.

Die Steuerzeichenkette enthält gewöhnlich Konvertierungsvorschriften, um die Interpretation der Eingabefolge zu steuern. Sie kann folgendes enthalten:

- Trennzeichen (Leerzeichen, Tabulator, Newline oder Formfeed), die (mit Ausnahme von zwei weiter unten beschriebenen Fällen) bewirken, dass die Eingabe bis zum folgenden Nichttrennzeichen gelesen wird.
- Ein normales Zeichen (nicht %), das mit dem folgenden Zeichen in der Eingabe übereinstimmen muss.
- Konvertierungsvorschriften, die durch das Zeichen % eingeleitet werden und die allgemeine Form

  %[*][<feldbreite>][l|h]<konv_zeichen>

  besitzen.

Eine Konvertierungsvorschrift legt die Konvertierung des nächsten Eingabefeldes fest. Das Ergebnis wird in die durch das zugehörige Argument adressierte Variable gespeichert, ausser wenn die Zuweisung durch die Angabe von * unterdrückt wurde. Damit ist es möglich, ein Eingabefeld zu beschreiben, das übergangen werden soll. Unter einem Eingabefeld wird eine Zeichenfolge ohne Trennzeichen verstanden. Es endet beim nächsten nicht zur Konvertierungsvorschrift passenden Zeichen oder wenn die angegebene Feldbreite erschöpft ist. Bei allen Konvertierungen ausser [ und c werden führende Trennzeichen übergangen.

Ein Konvertierungszeichen legt die Interpretation des Eingabefeldes fest. Bei unterdrückter Zuweisung ist kein Zeigerargument anzugeben. Folgende Konvertierungszeichen sind möglich:

% Ein einzelnes %-Zeichen wird an dieser Stelle in der Eingabe erwartet. Eine Zuweisung erfolgt nicht.

n Es wird die Gesamtzahl der Zeichen bereitgestellt, die seit Beginn der Auswertung verarbeitet wurden.

d Eine Integerzahl in Dezimaldarstellung wird erwartet. Das zugehörige Argument sollte ein **int**-Zeiger sein.

u Eine vorzeichenlose Integerzahl in Dezimaldarstellung wird erwartet. Das Argument sollte ein **unsigned**-Zeiger sein.

o Eine Integerzahl in Oktaldarstellung wird erwartet. Das Argument sollte ein **int**-Zeiger sein.

x Eine Integerzahl in Hexadezimaldarstellung wird erwartet. Das Argument sollte ein **int**-Zeiger sein.

i Eine Integerzahl in einer den Sprachregeln von C entsprechenden Darstellung wird erwartet. Das Argument sollte ein **int**-Zeiger sein.

e,f,g Eine Realzahl wird erwartet. Das Argument sollte ein **float**-Zeiger sein. Das Eingabefeld kann aus einer wahlweise vorzeichenbehafteten Folge von Ziffern bestehen, die einen Dezimalpunkt enthalten kann. Danach kann noch ein Exponent stehen, der aus dem Buchstaben e oder E, einem wahlweisen Zeichen +, - oder Leerzeichen sowie einer Inte-

gerzahl besteht.

s Eine Zeichenkette wird erwartet. Das Argument sollte ein Zeiger auf ein Feld von Zeichen sein. Das Feld muss gross genug sein, die Zeichenfolge und ein abschliessendes \Ø, das automatisch hinzugefügt wird, aufzunehmen. Das Eingabefeld wird durch ein Trennzeichen begrenzt.

c Ein Zeichen wird erwartet. Das Argument sollte ein Zeiger auf Zeichen sein. Das sonst übliche Übergehen von Trennzeichen wird in diesem Fall unterdrückt. Will man das nächste Zeichen lesen, das kein Trennzeichen ist, so muss man die Konvertierungsvorschrift %1s benutzen. Ist eine Feldbreite angegeben, so sollte das Argument ein Zeiger auf ein Zeichenfeld sein. Die angegebene Zeichenanzahl wird gelesen.

[ Erwartet wird eine Zeichenkette, wobei das sonst übliche Übergehen von Trennzeichen unterdrückt wird. Der linken Klammer [ folgt eine Menge von Zeichen, die scanset bezeichnet wird, und eine rechte Klammer ]. Das Eingabefeld erstreckt sich über die grösstmögliche Folge von Zeichen aus der definierten Menge scanset. Das Negationszeichen ^ dient als Komplementoperator, wenn es als erstes Zeichen in scanset vorkommt. Damit wird scanset als die Menge aller Zeichen definiert, die nicht in der restlichen Zeichenfolge enthalten sind. Es gibt Möglichkeiten zur vereinfachten Definition von scanset. Ein Zeichenbereich der Form <von>-<bis> kann verwendet werden, so dass anstelle von [0123456789] auch [0-9] notiert werden kann. Dabei muss <von> lexikographisch kleiner sein als <bis>. Ist das nicht der Fall, so steht das Zeichen - für sich selbst. Soll das Zeichen - zur Menge scanset gehören, kann es auch als erstes oder letztes Zeichen der Zeichenfolge angegeben werden. Soll das Zeichen ] selbst zur Zeichenmenge scanset gehören, so muss es als erstes Zeichen (eventuell nach dem Negationszeichen ^) notiert werden. Das zugehörige Argument muss ein Zeiger auf ein Zeichenfeld sein, das gross genug ist, alle eingelesenen Zeichen und das hinzugefügte Zeichen \Ø aufzunehmen. Eine solche Konvertierung wird als erfolgreich betrachtet, wenn mindestens ein Zeichen entsprechend der Vorschrift gelesen werden konnte.

Folgt dem Zeichen % kein gültiges Konvertierungszeichen, ist das Ergebnis des Funktionsaufrufs undefiniert.

Den Konvertierungszeichen d, u, o und x kann ein l bzw. h vorangestellt werden, um anzuzeigen, dass die zugehörigen Argumente **long**- bzw. **short**-Zeiger sind. Analog kann den Konvertierungszeichen e, f und g ein l vorangestellt werden, um mitzuteilen, dass das Argument ein **double**-Zeiger ist. Die Angabe l oder h vor anderen Konvertierungszeichen wird ignoriert.

Die Funktionen beenden die Konvertierung bei Auftreten der EOF-Bedingung, am Ende der Steuerzeichenkette format oder bei Auftreten eines Konflikts zwischen einem gelesenen Zeichen und der Konvertierungsvorschrift. Im letzteren Fall verbleibt dieses Zeichen im Eingabefeld.

Als Funktionswert wird die Anzahl der erfolgreich realisierten Konvertierungen mit Wertzuweisung zurückgegeben. Ein Funktionswert Null zeigt an, dass ein Fehler bereits bei der ersten Konvertierung aufgetreten ist. Der Wert EOF zeigt an, dass die EOF-Bedingung noch vor der ersten Konvertierung erkannt wurde. Ist der Funktionswert kleiner als die Anzahl der beabsichtigten Eingabekonvertierungen, so kann sowohl ein Konvertierungsfehler als auch die EOF-Bedingung aufgetreten sein.

**setbuf, setvbuf**

Festlegen der Pufferung für die Filearbeit

```
#include <stdio.h>

void setbuf(file_p, buf)
    FILE *file_p;
    char *buf;

int setvbuf(file_p, buf, type, size)
    FILE *file_p;
    char *buf;
    int type;
    int size;
```

Die Funktion **setbuf** kann man nach dem Öffnen eines Files, jedoch vor dem ersten Lesen oder Schreiben benutzen. Damit wird festgelegt, dass ein durch buf adressiertes Zeichenfeld anstelle eines automatisch zugeordneten Puffers benutzt werden soll. Bei Angabe des Zeigerwertes NULL für buf erfolgt die ungepufferte Ein- bzw. Ausgabe.

Die in stdio.h definierte Konstante BUFSIZ gibt an, wie gross das Zeichenfeld sein muss:

```
char buf[BUFSIZ];
```

Die Funktion **setvbuf** kann ebenfalls nach dem Öffnen und vor dem Lesen bzw. Schreiben eines Files benutzt werden. Hierbei wird mit der Angabe von type die Art der Pufferung festgelegt. Für type sind folgende in stdio.h definierten Konstanten erlaubt:

_IOFBF bewirkt die vollständig gepufferte Eingabe bzw. Ausgabe.

_IOLBF bewirkt die zeilenweise gepufferte Ausgabe. Hierbei erfolgt die Ausgabe des Puffers, wenn Newline geschrieben wird, der Puffer voll ist oder eine Eingabeanforderung vorliegt.

_IONBF hat die ungepufferte Eingabe bzw. Ausgabe zur Folge.

Ist buf ungleich NULL, so wird das durch buf adressierte Zeichenfeld anstelle eines sonst automatisch zugeordneten Puffers benutzt. Das Argument size gibt in diesem Fall die Grösse des zu nutzenden Puffers an. Die Konstante BUFSIZ ist hierfür ein geeigneter Wert. Für die ungepufferte Ein- und Ausgabe haben die Werte von buf und size keine Bedeutung.

Standardmässig erfolgt bei Terminalausgaben eine zeilenweise Pufferung, während bei allen anderen Ein- bzw. Ausgaben eine vollständige Pufferung durchgeführt wird.

**setjmp, longjmp**

Nichtlokale Sprünge

```
#include <setjmp.h>

int setjmp(env)
    jmp_buf env;

void longjmp(env, val)
    jmp_buf env;
    int val;
```

Die Funktion **setjmp** rettet ihre Stackumgebung in env. Der Typ von env ist im File setjmp.h deklariert. Die Funktion gibt den Wert Null zurück.

Die Funktion **longjmp** stellt die in env enthaltene Stackumgebung wieder her und kehrt an die Stelle des zugehörigen setjmp-Aufrufes mit dem Funktionswert val zurück. Besitzt val den Wert Null, so wird jedoch 1 zurückgegeben. Alle nicht im Stack abgelegten Variablen besitzen die gleichen Werte wie zum Zeitpunkt des longjmp-Aufrufes.

**sinh, cosh, tanh**

Hyperbelfunktionen

```
#include <math.h>

double sinh(x)
    double x;

double cosh(x)
    double x;

double tanh(x)
    double x;
```

**sleep**

Aussetzen der Programmausführung für ein Zeitintervall

```
unsigned sleep(seconds)
    unsigned seconds;
```

**ssignal, gsignal**

Softwaresignale

```
#include <signal.h>

int (*ssignal(sig, action)()
    int sig;
    int (*action)();

int gsignal(sig)
    int sig;
```

**string - strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok**

Zeichenkettenoperationen

```
#include <string.h>

char *strcat(s1, s2)
    char *s1, *s2;

char *strncat(s1, s2, n)
    char *s1, *s2;
    int n;

int strcmp(s1, s2)
    char *s1, *s2;

int strncmp(s1, s2, n)
    char *s1, *s2;
    int n;

char *strcpy(s1, s2)
    char *s1, *s2;

char *strncpy(s1, s2, n)
    char *s1, *s2;
    int n;

int strlen(s)
    char *s;

char *strchr(s, c)
    char *s;
    int c;

char *strrchr(s, c)
    char *s;
    int c;

char *strpbrk(s1, s2)
    char *s1, *s2;

int strspn(s1, s2)
    char *s1, *s2;

int strcspn(s1, s2)
    char *s1, *s2;

char *strtok(s1, s2)
    char *s1, *s2;
```

Die Argumente s1 und s2 zeigen auf Zeichenketten, die mit \Ø abgeschlossen sind. Die Funktionen strcat, strncat, strcpy und strncpy verändern die Zeichenkette s1, ohne zu überprüfen, ob in s1 genug Platz vorhanden ist.

Die Funktion **strcat** kopiert die Zeichenkette s2 an das Ende von s1. Die Funktion **strncat** kopiert maximal n Zeichen der Zeichenkette s2. Beide liefern einen Zeiger auf die Ergebniszeichenkette, die wiederum mit \Ø abgeschlossen ist.

Die Funktionen **strcmp** und **strncmp** vergleichen die Zeichenketten s1 und s2. Als Ergebnis liefern sie eine Interzahl, die kleiner, gleich oder grösser als Null ist, je nachdem, ob s1 lexikographisch kleiner, gleich oder grösser als s2 ist. Bei strncmp werden maximal n Zeichen verglichen.

Die Funktion **strcpy** kopiert die Zeichenkette s2 nach s1 einschliesslich \Ø am Ende von s2. Die Funktion **strncpy** kopiert genau n Zeichen. Ist s2 länger als n-1 Zeichen, so wird kein \Ø angefügt. Im anderen Fall wird mit \Ø aufgefüllt. Beide Funktionen geben die Adresse von s1 zurück.

Die Funktion **strlen** gibt die Anzahl der Zeichen von s zurück (das Zeichen \Ø nicht mitgerechnet).

Die Funktionen **strchr** bzw. **strrchr** suchen nach dem ersten bzw. letzten Auftreten des Zeichens c in der Zeichenkette s. Als Ergebnis wird die Adresse dieses Zeichens zurückgegeben oder der Zeigerwert NULL, falls das Zeichen nicht in s enthalten ist. Das abschliessende \Ø wird dabei als Bestandteil der Zeichenkette betrachtet.

Die Funktion **strpbrk** sucht nach dem ersten Auftreten eines beliebigen Zeichens aus s2 in der Zeichenkette s1. Sie liefert die Adresse dieses Zeichens oder den Zeigerwert NULL, falls keines dieser Zeichen existiert.

Die Funktionen **strspn** bzw. **strcspn** ermitteln die Länge der am Anfang von s1 stehenden Zeichenfolge, die nur aus Zeichen der Zeichenkette s2 bzw. überhaupt nicht aus Zeichen der Zeichenkette s2 besteht.

Die Funktion **strtok** untersucht die Zeichenkette s1, die als eine Folge von null oder mehr lexikalen Einheiten (Token) besteht. Die einzelnen Token müssen durch Trennzeichen getrennt sein. Diese Trennzeichen werden in s2 definiert. Der erste Aufruf liefert einen Zeiger auf das erste Token in s1. Das erste Zeichen nach dem Token wird mit \Ø überschrieben und die gefundene Position vermerkt. Will man auf sie Bezug nehmen, so muss s1 den Wert NULL besitzen. Eine Folge solcher Funktionsaufrufe liefert dann die einzelnen Token in der Zeichenkette s1. Die Trennzeichenkette s2 kann in der Aufruffolge variiert werden. Ist kein Token mehr in s1 enthalten, wird der Zeigerwert NULL zurückgegeben.

**strtod, atof**

Konvertieren einer Zeichenkette in eine Realzahl des Typs **double**

```
double strtod(str, ptr)
    char *str;
    char **ptr;

double atof(str)
    char *str;
```

Die Funktion **strtod** konvertiert die Zeichenkette str in eine Realzahl und gibt diese als Funktionswert zurück. Trennzeichen am Anfang von string werden übergangen. Weitere Bestandteile der Zeichenkette sind ein Vorzeichen (wahlweise), eine Folge von Dezimalziffern einschliesslich eines Dezimalpunktes, ein wahlweises Exponentenzeichen (e oder E) sowie ein wahlweises Vorzeichen und ein Integerwert.

Ist der Wert von ptr nicht (**char** **)Ø, so enthält *ptr nach der Abarbeitung der Funktion die Adresse des Zeichens, bei dem die Konvertierung beendet wurde. Tritt bei der Konvertierung eine Fehlerbedingung auf, so zeigt *ptr auf str und der Funktionswert ist Null.

Der Funktionsaufruf strtod(str, (**char** **)Ø) ist zu atof(str) äquivalent.

**strtol, atol, atoi**

Konvertieren einer Zeichenkette in eine Integerzahl

```
long strtol(str, ptr, base)
    char *str, **ptr;
    int base;

long atol(str)
    char *str;

int atoi(str)
    char *str;
```

Die Funktion **strtol** konvertiert eine Zeichenkette str in eine Integerzahl vom Typ **long**. Führende Trennzeichen werden übergangen. Das Argument base (Ø <= base <= 36) bestimmt die Zahlenbasis. Die Konvertierung endet beim ersten unzulässigen Zeichen. Die Zeichenkette kann ein Vorzeichen enthalten. Führende Nullen werden ignoriert, ebenso die Zeichenfolgen Øx bzw. ØX bei base gleich 16.

Ist der Wert von ptr nicht gleich (**char** **)Ø, so enthält *ptr nach der Abarbeitung der Funktion die Adresse des Zeichens, bei dem die Konvertierung beendet wurde. Tritt bei der Konvertierung eine Fehlerbedingung auf, so zeigt *ptr auf str, und der Funktionswert ist Null.

Ist für base der Wert Ø angegeben, so bestimmt die Zeichenkette str selbst die zu verwendende Zahlenbasis. Enthält str (nach dem eventuellen Vorzeichen) als erstes das Zeichen Ø, so wird als Basis 8, bei einer Zeichenfolge Øx bzw. ØX wird als Basis 16 und in allen anderen Fällen 10 angenommen.

Die Funktionsaufrufe **atol**(str) bzw. **atoi**(str) sind äquivalent zu strtol(str, (**char** **)Ø, 1Ø) bzw. (**int**) strtol(str, (**char** **)Ø, 1Ø).

**swab**

Vertauschen von Bytes

```
void swab(from, to, nbytes)
    char *from, *to;
    int nbytes;
```

Die Funktion **swab** kopiert nbytes Bytes von from nach to. Dabei werden jeweils zwei benachbarte Bytes miteinander vertauscht.

**system**

Ausführen eines Kommandos

```
#include <stdio.h>

int system(string)
    char *string;
```

Die Funktion **system** übergibt die Zeichenkette string an den Kommandointerpreter (shell). Der aufrufende Prozess wartet auf die Beendigung des Kommandos. Der Funktionswert von system ist der exit-Status des Kommandos.

**tmpfile**

Erzeugen eines temporären Files

```
#include  <stdio.h>

FILE *tmpfile()
```

Die Funktion **tmpfile** erzeugt ein temporäres File, dessen Name mit der Standardfunktion tmpnam gebildet wird. Wenn das File nicht geöffnet werden kann, dann wird eine Fehlernachricht ausgegeben und der Zeigerwert NULL zurückgegeben, ansonsten der entsprechende Filepointer. Das File ist für Lesen und Schreiben ("w+") geöffnet und wird bei Prozessende automatisch beseitigt.

**tmpnam, tempnam**

Erzeugen eines Filenamens für ein temporäres File

```
#include  <stdio.h>

char *tmpnam(s)
    char *s;

char *tempnam(dir, pfx)
    char *dir, *pfx;
```

Beide Funktionen generieren Filenamen, die für temporäre Files benutzt werden können.

Die Funktion **tmpnam** generiert einen Namen, der den im File stdio.h definierten Pfadpräfix P_tmpdir enthält. Ist s gleich NULL, so wird der Filename in einem eigenen Feld erzeugt und die Adresse dieses Feldes zurückgegeben. Ansonsten wird s als ein Feld der Mindestlänge L_tmpnam angenommen, der Filename dorthin gespeichert und s zurückgegeben.

Mit der Funktion **tempnam** kann das Verzeichnis dir festgelegt werden, in dem das File erzeugt werden soll. Ist dir gleich NULL, so wird versucht, den Pfadpräfix P_tmpdir zu verwenden. Ist auch das nicht möglich, wird /tmp angenommen. Der Filename wird in einem mit malloc angeforderten Bereich erzeugt und die Adresse dieses Bereiches zurückgegeben. Mit der Zeichenkette pfx kann man maximal die ersten 5 Zeichen des Filenamens vorgeben.

**trig - sin, cos, tan, asin, acos, atan, atan2**

Trigonometrische Funktionen

```
#include  <math.h>

double sin(x)
    double x;

double cos(x)
    double x;

double tan(x)
    double x;
```

```
double asin(x)
    double x;

double acos(x)
    double x;

double atan(x))
    double x;

double atan2(y, x)
    double y;
    double x;
```

Mit **sin, cos** und **tan** werden für im Bogenmass angegebene Argumentwerte x die entsprechenden Funktionswerte der trigonometrischen Funktionen ermittelt.

**asin** berechnet arcsin von x im Bereich -pi/2 bis pi/2.
**acos** berechnet arccos von x im Bereich Ø bis pi.
**atan** berechnet arctan von x im Bereich -pi/2 bis pi/2.
**atan2** berechnet arctan von y/x im Bereich -pi bis pi.

## tsearch, tfind, tdelete, twalk

Verwalten von Binärbäumen

```
#include  <search.h>

char *tsearch(key, rootp, compar)
    char *key, **rootp;
    int (*commpar)();

char *tfind(key, rootp, compar)
    char *key, **rootp;
    int (*compar)();

char *tdelete(key, rootp, compar)
    char *key, **rootp;
    int (*compar)();

void twalk(root, action)
    char **rootp;
    void (*action)();
```

Die Funktionen **tsearch** und **tfind** durchsuchen einen binären Baum mit der Wurzel **rootp nach dem Schlüssel *key. Der Vergleich wird mit der vom Anwender bereitgestellten Funktion *compar durchgeführt, der als Argumente zwei Zeiger auf die zu vergleichenden Elemente übergeben werden. Diese Funktion muss einen Wert kleiner, gleich oder grösser als Null liefern, je nachdem, ob das erste Argument kleiner, gleich oder grösser als das zweite Argument ist. Wird der Schlüssel *key nicht gefunden, so fügt tsearch das neue Element ein, tfind gibt in diesem Fall den Zeigerwert NULL zurück.

Die Funktion **tdelete** beseitigt das Element *key. Die Funktion **twalk** durchmustert die Baumstruktur. Der Anwenderfunktion *action wird jeweils der aktuelle Knoten und eine Information darüber übergeben, wann der Knoten getroffen wurde (preorder, postorder oder endorder) bzw. ob es sich um ein Blatt handelt (leaf). Diese Konstanten sind als Aufzählungstyp im File search.h definiert.

**ttyname, isatty**

Bereitstellen des Terminalnamens

```
char *ttyname(fildes)
    int fildes;

int isatty(fildes)
    int fildes;
```

**ttyslot**

Suchen des Index für den aktuellen Nutzereintrag im File /etc/utmp

```
int ttyslot()
```

**ungetc**

Zurückstellen eines Zeichens in den Eingabepuffer

```
#include  <stdio.h>

int ungetc(c, file_p)
    int c;
    FILE *file_p;
```

Die Funktion **ungetc** stellt das Zeichen c in den zum File file_p gehörenden Puffer zurück. Dieses Zeichen wird beim nächsten Lesezugriff bereitgestellt. Der Funktionswert von ungetc ist c bzw. EOF im Fehlerfall.

Das Zurückstellen eines Zeichens wird garantiert, vorausgesetzt, aus dem File wurde bereits gelesen. Beim Standardfile stdin kann ein Zeichen auch vor dem ersten Lesen zurückgestellt werden.

Bei c gleich EOF ist der Funktionsaufruf ohne jede Wirkung, als Funktionswert wird EOF geliefert.

Die Funktion fseek beseitigt zurückgestellte Zeichen.

**vprintf, vfprintf, vsprintf**

Formatierte Ausgabe mit einer in varargs.h definierten Argumentliste

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

int vprintf(format, ap)
    char *format;
    va_list ap;

int vfprintf(file_p, format, ap)
    FILE *file_p;
    char *format;
    va_list ap;

int vsprintf(s, format, ap)
    char *s, *format;
    va_list ap;
```

# Anhang C. Prüfprogramm lint

Zu den Programmierwerkzeugen (tools), die das Betriebssystem UNIX bereitstellt, gehört ein spezielles Prüfprogramm **lint**, das C-Quelltexte auf potentielle Portabilitätsprobleme untersucht. Ausserdem führt lint eine gegenüber dem Compiler strengere Prüfung auf Verträglichkeit der Datentypen sowie eine genauere Syntaxprüfung durch, entdeckt einige spezielle Probleme, wie die Benutzung nicht initialisierter **auto**- und **register**-Variablen, nicht erreichbare Programmteile usw. Man kann lint anwenden, um die zu einem Programm gehörenden Quelltextfiles auf Konsistenz der externen Datenverbindungen zu untersuchen.

Als Ergebnis liefert lint ggf. Fehlernachrichten bzw. Warnungen. Im Gegensatz zum Compiler wird keine Kodegenerierung durchgeführt.

Es folgt eine Aufzählung der wesentlichen (potentiellen) Fehlerquellen, die lint entdeckt:

- definierte, aber niemals verwendete Variablen und Funktionen sowie nicht benutzte Funktionsargumente
- Benutzung nicht initialisierter (nicht belegter) **auto**- und **register**-Variablen
- nicht erreichbare Programmteile (z.B. nicht markierte Anweisungen unmittelbar nach **goto**, **break**, **continue** und **return** sowie Schleifenkörper, die niemals am Ende verlassen werden bzw. bei denen der Eintritt niemals am Beginn erfolgt)
- zurückgegebene Funktionswerte, die in der rufenden Funktion nicht benutzt werden
- undefinierte Funktionswerte, die in der rufenden Funktion benutzt werden
- unverträgliche Operandentypen bei Wertzuweisungen, Vergleichsoperatoren, dem Entscheidungsoperator, den Operatoren zur Auswahl von Struktur- und Unionkomponenten
- Typkonflikte zwischen Funktionsparametern und den entsprechenden Funktionsargumenten sowie zwischen der Funktionsdefinition und dem gelieferten Funktionswert
- nichtportable Verwendung von **char**-Variablen
- Zuweisungen von **long**-Werten an Variablen des Typs **int**
- Ausdrücke mit Nebeneffekten, deren Wert von der Auswertungsreihenfolge abhängt.

Bestimmte Warnungen und Fehlernachrichten können durch spezielle Kommentare im Quelltext unterdrückt werden:

| | |
|---|---|
| /*ARGUSED*/ | nicht benutzte Argumente der folgenden Funktion |
| /*VARARGS*/ | variable Anzahl von Argumenten beim Funktionsaufruf |
| /*VARARGSn*/ | nur die ersten n Argumente werden geprüft |
| /*NOTREACHED*/ | eine nicht erreichbare Stelle im Programm. |

Das Prüfprogramm lint kann folgendermassen aufgerufen werden, wobei nur diejenigen Flags angegeben sind, die den Grad der Prüfung beeinflussen:

lint [-abhnuvx] <file$_1$> [<file$_2$> ...]

Die einzelnen Flags besitzen folgende Bedeutung:

a Unterdrücken von Nachrichten bei Zuweisung von **long**-Werten an Variable der Typen **int**, **short**, **char**

b Unterdrücken von Nachrichten über nicht erreichbare **break**-Anweisungen

h keine heuristischen Untersuchungen (Reduzierung von Redundanz, Verbesserung des Stils, Fehlererkennung)

n keine Prüfung der Kompatibilität zur Standardbibliothek

u Unterdrücken der Nachrichten bei Funktionen oder externen Variablen, die verwendet werden, aber nicht definiert sind bzw. die definiert sind, aber nicht verwendet werden

v Unterdrücken von Nachrichten bei Argumenten, die nicht verwendet werden

x keine Nachrichten bei Variablen, für die externe Deklarationen existieren, die aber nicht verwendet werden.

# Anhang D. ASCII- und EBCDIC-Zeichensatz

**ASCII**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 nul | 10 dle | 20 sp | 30 0 | 40 @ | 50 P | 60 ` | 70 p |
| 01 soh | 11 dc1 | 21 ! | 31 1 | 41 A | 51 Q | 61 a | 71 q |
| 02 stx | 12 dc2 | 22 " | 32 2 | 42 B | 52 R | 62 b | 72 r |
| 03 etx | 13 dc3 | 23 # | 33 3 | 43 C | 53 S | 63 c | 73 s |
| 04 eot | 14 dc4 | 24 ¤ | 34 4 | 44 D | 54 T | 64 d | 74 t |
| 05 enq | 15 nak | 25 % | 35 5 | 45 E | 55 U | 65 e | 75 u |
| 06 ack | 16 syn | 26 & | 36 6 | 46 F | 56 V | 66 f | 76 v |
| 07 bel | 17 etb | 27 ' | 37 7 | 47 G | 57 W | 67 g | 77 w |
| 08 bs | 18 can | 28 ( | 38 8 | 48 H | 58 X | 68 h | 78 x |
| 09 ht | 19 em | 29 ) | 39 9 | 49 I | 59 Y | 69 i | 79 y |
| 0a lf | 1a sub | 2a * | 3a : | 4a J | 5a Z | 6a j | 7a z |
| 0b vt | 1b esc | 2b + | 3b ; | 4b K | 5b [ | 6b k | 7b { |
| 0c ff | 1c fs | 2c , | 3c < | 4c L | 5c \ | 6c l | 7c \| |
| 0d cr | 1d gs | 2d - | 3d = | 4d M | 5d ] | 6d m | 7d } |
| 0e so | 1e rs | 2e . | 3e > | 4e N | 5e ^ | 6e n | 7e ~ |
| 0f si | 1f us | 2f / | 3f ? | 4f O | 5f _ | 6f o | 7f del |

**EBCDIC**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 nul | 10 dle | 20 ds | 30 | 40 sp | 50 & | 60 - | 70 |
| 01 soh | 11 dc1 | 21 sos | 31 | 41 | 51 | 61 / | 71 |
| 02 stx | 12 dc2 | 22 fds | 32 syn | 42 | 52 | 62 | 72 |
| 03 etx | 13 dc3 | 23 | 33 | 43 | 53 | 63 | 73 |
| 04 pf | 14 res | 24 byp | 34 pn | 44 | 54 | 64 | 74 |
| 05 ht | 15 nl | 25 lf | 35 rst | 45 | 55 | 65 | 75 |
| 06 lc | 16 bs | 26 etb | 36 uc | 46 | 56 | 66 | 76 |
| 07 del | 17 il | 27 esc | 37 eot | 47 | 57 | 67 | 77 |
| 08 | 18 can | 28 | 38 | 48 | 58 | 68 | 78 |
| 09 | 19 em | 29 | 39 | 49 | 59 | 69 | 79 ` |
| 0a smm | 1a cc | 2a sm | 3a | 4a [ | 5a ] | 6a \| | 7a : |
| 0b vt | 1b cu1 | 2b cu2 | 3b cu3 | 4b . | 5b ¤ | 6b , | 7b # |
| 0c ff | 1c fs | 2c | 3c stop | 4c < | 5c * | 6c % | 7c @ |
| 0d cr | 1d gs | 2d enq | 3d nak | 4d ( | 5d ) | 6d _ | 7d ' |
| 0e so | 1e rs | 2e ack | 3e | 4e + | 5e ; | 6e > | 7e = |
| 0f si | 1f us | 2f bel | 3f sub | 4f ! | 5f ^ | 6f ? | 7f " |
| | | | | | | | |
| 80 | 90 | a0 | b0 | c0 { | d0 } | e0 \ | f0 0 |
| 81 a | 91 j | a1 ~ | b1 | c1 A | d1 J | e1 | f1 1 |
| 82 b | 92 k | a2 s | b2 | c2 B | d2 K | e2 S | f2 2 |
| 83 c | 93 l | a3 t | b3 | c3 C | d3 L | e3 T | f3 3 |
| 84 d | 94 m | a4 u | b4 | c4 D | d4 M | e4 U | f4 4 |
| 85 e | 95 n | a5 v | b5 | c5 E | d5 N | e5 V | f5 5 |
| 86 f | 96 o | a6 w | b6 | c6 F | d6 O | e6 W | f6 6 |
| 87 g | 97 p | a7 x | b7 | c7 G | d7 P | e7 X | f7 7 |
| 88 h | 98 q | a8 y | b8 | c8 H | d8 Q | e8 Y | f8 8 |
| 89 i | 99 r | a9 z | b9 | c9 I | d9 R | e9 Z | f9 9 |
| 8a | 9a | aa | ba | ca | da | ea | fa |
| 8b | 9b | ab | bb | cb | db | eb | fb |
| 8c | 9c | ac | bc | cc | dc | ec | fc |
| 8d | 9d | ad | bd | cd | dd | ed | fd |
| 8e | 9e | ae | be | ce | de | ee | fe |
| 8f | 9f | af | bf | cf | df | ef | ff |

# Anhang E. Standardisierung von C

Seit dem Zeitpunkt des Entwurfes wurde die Programmiersprache C ständig weiterentwickelt. Über einen längeren Zeitraum galt die von Ritchie und Kernighan in /34/ festgelegte Sprachdefinition als De-facto-Sprachstandard. Viele der gegenwärtig im Einsatz befindlichen Compiler realisieren den Sprachumfang, wie er im C Reference Manual der Version UNIX V, Release 2.0 /2/ und im Teil 3 des X/OPEN Portability Guide /50/ definiert und in diesem Buch vorgestellt wird.

Seit mehreren Jahren währen die Bemühungen des ANSI-Komitees X3J11, einen Standard für die Programmiersprache C zu erarbeiten. Obwohl der endgültige ANSI-C-Standard noch nicht vorliegt, zeichnen sich in den vorgelegten Entwürfen einige Erweiterungen und Änderungen gegenüber dem hier zur Grundlage genommenen Sprachumfang ab.

Der Anhang gibt einen Überblick über die wesentlichen Unterschiede von C entsprechend /2/ bzw. /50/ (X/OPEN-C) gegenüber /34/ (Ritchie-C) einerseits und ANSI-C entsprechend /51/ gegenüber X/OPEN-C andererseits.

**Unterschiede von X/OPEN-C gegenüber Ritchie-C**

1. Neue Datentypen

Als zusätzliche Datentypen wurden der Aufzählungstyp (Schlüsselwort **enum**) und der Typ für die leere Wertemenge (Schlüsselwort **void**) aufgenommen. Als Typspezifikation ist auch **unsigned char** möglich.

2. Zuweisung von Strukturen und Unions

Erlaubt ist die Wertzuweisung von Strukturen bzw. Unions sowie die Übergabe von Strukturen bzw. Unions als Funktionsargument und als Funktionsergebnis.

3. Erweiterte Namensräume für Struktur- und Union-Komponenten

Ursprünglich war festgelegt, dass die Bezeichner für Komponenten von Strukturen und Unions alle voneinander verschieden sein müssen, wenn sie nicht zur gleichen Anfangsfolge gehören. Diese starke Einschränkung wurde dadurch aufgeweicht, dass der Compiler nur prüft, ob ein in unterschiedlichen Strukturen vorkommender Komponentenbezeichner den gleichen Typ und Offset besitzt.

X/OPEN-C erweitert die Namensräume für Komponentenbezeichner folgendermassen: Bei der Vereinbarung von Strukturen oder Unions können gleiche Bezeichner für Komponenten unterschiedlicher Struktur- oder Union-Typen verwendet werden.

Diese Änderung ist nicht rückwärtskompatibel.

4. Strengere Typprüfung bei Zugriff auf Strukturkomponenten

Laut /34/ ist es erlaubt, dass in Strukturbezugnahmen der Form

<struktur>.<komponente>

für <struktur> ein beliebiger Lvalue steht. Analog kann bei Strukturbezugnahmen der Form

<struktur_zeiger>-><komponente>

für <struktur_zeiger> ein beliebiger Zeiger- oder Integerwert stehen. Der Compiler geht dabei von der Annahme aus, dass sich <struktur> bzw. <struktur_zeiger> auf eine Instanz eines Strukturtyps beziehen, in der <komponente> als Komponente enthalten ist.

Dagegen fordert X/OPEN-C, dass <struktur> der Bezeichner einer Strukturinstanz bzw. <struktur_zeiger> ein Zeiger auf eine Strukturinstanz und <komponente> der Name einer Komponente der jeweiligen Struktur sind.

5. Benutzung von Union-Komponenten

X/OPEN schränkt die Benutzung von Union-Komponenten folgendermassen ein: Eine Union-Komponente sollte nur dann benutzt werden, wenn der aktuelle Wert dieser Union aus einer Wertzuweisung an die gleiche Komponente resultiert. Um die Arbeit mit Unions zu erleichtern, wird jedoch folgendes garantiert: Enthält eine Union mehrere Strukturen, die alle die gleiche Anfangsfolge von Strukturkomponenten enthalten, und enthält die Union zum betreffenden Zeitpunkt eine dieser Strukturen, so ist es erlaubt, eine beliebige Komponente irgendeiner der Strukturen zu untersuchen, wenn sie zur gemeinsamen Anfangsfolge gehört.

6. Veraltete Syntaxformen

Nicht mehr erlaubt ist es, das Zeichen = bei einer Initialisierung der Form

<typ> <bezeichner> = <init_wert>;

wegzulassen, z.B.

```
int x 2;
```

Ebenso ist die veraltete Form der zusammengesetzten Wertzuweisung

<lvalue> =<op> <ausdruck>;

nicht mehr zulässig.

7. Zeiger- und Integerwerte

Nicht mehr erlaubt ist die Vermischung von Zeiger- und Integerobjekten in Wertzuweisungen und bei Tests auf Gleichheit bzw. Ungleichheit. Ausnahmen bestehen in der Zuweisung der Integerkonstanten Ø an einen Zeiger und im Vergleich eines Zeigerwertes mit der Konstanten Ø.

8. Vereinbarung externer Datenobjekte

Laut /34/ muss ein Datenobjekt mit dem Speicherklassenattribut **extern** in allen zum Programm gehörenden Quellfiles genau einmal definiert werden:

<typ> <bezeichner> [ = <init_wert> ];

In jedem der Files können eine oder mehrere Deklarationen für diesen Bezeichner enthalten sein, wobei der Typ mit dem in der Definition übereinstimmen muss:

**extern** <typ> <bezeichner>;

X/OPEN-C übernimmt diese Regeln für sog. "eingeschränkte Umgebungen". In allen anderen Compilerumgebungen ist für **extern**-Datenobjekte festgelegt, dass sie wenigstens einmal in allen zum Programm gehörenden Quellfiles definiert werden müssen, wobei eine explizite Initialisierung nur einmal vorgenommen werden darf.

9. Konstanten

Nicht mehr erlaubt sind die Ziffern 8 bzw. 9 innerhalb von Oktalkonstanten der Form '\ddd'. (Sie wurden früher als Oktalwerte 10 bzw. 11 interpretiert.)

Die Notation '\v' für den Vertikal-Tabulator ist in X/OPEN-C zulässig.

Neu festgelegt ist die Reaktion des Compilers, wenn dem Backslash \ andere Zeichen als n, t, b, f, v, ', \, " bzw. Oktalziffern folgen. In Ritchie-C wird der Backslash einfach ignoriert, während laut X/OPEN-C das Verhalten des Compilers undefiniert ist.

Die beiden letztgenannten Änderungen sind nicht rückwärtskompatibel.

10. Typdefinition

Bisher offen war die Möglichkeit, mittels **typedef** eingeführte Bezeichner mit den Typbezeichnern **long**, **short** oder **unsigned** zu verknüpfen, z.B.

```
typedef int abstand;
...
long abstand x;
```

Das ist nicht mehr zulässig.

11. Präprozessor

In Compilersteuerzeilen der Form

**#if** <konst_ausdruck>

kann <konst_ausdruck> zusätzlich einen Ausdruck der Form

**defined** <bezeichner>

oder

**defined** (<bezeichner>)

enthalten. Diese Möglichkeit ist neu in X/OPEN-C.

**Unterschiede von ANSI-C gegenüber X/OPEN-C**

1. Typattribut **const**

Datenobjekte, die mit dem Typattribut **const** vereinbart werden, können während der Programmausführung nicht modifiziert werden. Mit der Vereinbarung ist eine explizite Initialisierung erforderlich, der Wert ist damit festgelegt. Die allgemeine Form lautet:

```
const <typ> <bezeichner> = <init_wert>;
```

Zum Beispiel haben die Vereinbarungen

```
const float pi = 3.14;
const float *pcf = &pi;
```

folgende Bedeutung: pi ist ein Datenobjekt mit dem konstanten Wert 3.14 und kann nur gelesen werden (read-only-Variable) und pcf ist ein Zeiger auf einen konstanten Wert.

Zeiger auf konstante Werte können modifiziert werden:

```
const float e = 2.72;
const float pi = 3.14;
float x;
const float *pcf = &pi;
...
pcf = &e;
...
pcf = &x;
```

Es ist jedoch nicht möglich, über diesen Zeiger den Wert des Datenobjektes x zu verändern:

```
*pcf = ...;                /* nicht zulässig! */
```

Es ist auch nicht möglich, die Adresse eines mit dem Attribut **const** vereinbarten Objektes einem Zeiger zuzuweisen, der auf Objekte ohne das **const**-Attribut zeigt:

```
const float pi = 3.14;
float *pf;
...
pf = &pi;                  /* nicht zulässig! */
```

Zwischen einem "Zeiger auf einen konstanten Wert" und einem "konstanten Zeiger auf einen Wert" besteht ein Unterschied:

```
float x = 3.14;
float y = 3.14;
...
const float *pcf = &x;
float *const cpf = &y;
```

Der Wert von x kann über pcf nicht verändert werden, aber der Wert von pcf kann sich ändern (s. o.).

Der Wert von cpf kann nicht modifiziert werden, der Zeiger bezieht sich immer auf die gleiche Speicheradresse. Andererseits ist es möglich,

über cpf den Wert von y zu verändern:

```
*cpf = ...;
```

2. Initialisierung

Mit der Vereinbarung ist die explizite Initialisierung beliebiger Datenobjekte möglich. Insbesondere können auch Felder und Strukturen mit dem Speicherklassenattribut **auto** initialisiert werden, wobei beliebige Ausdrücke zulässig sind, die Konstanten, bereits vereinbarte Bezeichner, Funktionsaufrufe und Wertzuweisungen enthalten. Bei einer Union erfolgt die Initialisierung, indem der Initialisierungswert der ersten in der Union aufgeführten Komponente zugewiesen wird. Bei Objekten mit dem Attribut **const** ist die explizite Wertzuweisung bei der Vereinbarung notwendig.

3. **float** und **double**

In arithmetischen Ausdrücken wird keine automatische Typkonvertierung von **float** in **double** vorgenommen. Bei Funktionsargumenten findet die automatische Typkonvertierung von **float** in **double** statt, wenn nicht mit der Funktionsdeklaration als Parametertyp **float** angegeben wird (s. u.).

4. Funktionsdeklaration

Bei der Deklaration einer Funktion können die Typen der Parameter angegeben werden. Das bewirkt beim Aufruf der Funktion die Prüfung der Typen der Aufrufargumente mit den entsprechenden Parametertypen. Sind Argument- und Parametertyp zueinander verträglich, so erfolgt die Konvertierung des Argumenttyps in den entsprechenden Parametertyp. Sind die Typen nicht verträglich, so wird ein Fehler angezeigt.

Folgende Beispiele verdeutlichen die Möglichkeiten der Typprüfung von Funktionsargumenten.

```
extern char *f1(int, char *);
```

Diese Funktionsdeklaration gibt an, dass jeder Aufruf der Funktion f1 mit zwei Argumenten erfolgt, wobei das erste Argument vom Typ **int** und das zweite Argument vom Typ **char** * sein muss.

```
extern int f2(int, char *, float, );
```

Beim Aufruf von f2 sind wenigstens drei Argumente erforderlich. Das abschliessende Komma bedeutet, dass eventuell weitere Argumente möglich sind, deren Typ aber nicht geprüft werden soll. Nebenbei bemerkt, ist in diesem Fall eines der weiteren Argumente vom Typ **float**, so erfolgt (wie bisher) die automatische Typangleichung an den Typ **double**.

```
extern int f3(void);
```

Die Funktion f3 erwartet keine Argumente.

```
extern void f4();
```

Es erfolgt beim Aufruf von f4 keine Prüfung der Argumenttypen. (Damit ist die Aufwärtskompatibilität gegeben.)

5. Zeichenkonstanten

Als zusätzliche Notation sind '\a' für das akustische Alarmsignal und '\xddd' für hexadezimale Zeichenkonstanten (ddd: 1 bis 3 hexadezimale Ziffern) zulässig.

6. **switch**-Wert

Der Wert von <ausdruck> in

```
switch(<ausdruck>) {
    ...
}
```

ist nicht mehr nur auf den Typ **int** beschränkt, sondern es sind beliebige integrale Typen erlaubt.

7. Vereinbarung externer Datenobjekte

Aus Gründen der Portabilität wurde in den ANSI-C-Standardentwurf wieder die strengere Forderung aufgenommen, dass ein Datenobjekt mit dem Speicherklassenattribut **extern** in allen zum Programm gehörenden Quellfiles genau einmal definiert werden muss.

8. Unärer arithmetischer Operator +

Analog zum unären arithmetischen Operator - (negatives Vorzeichen) wurde der unäre arithmetische Operator + (positives Vorzeichen) eingeführt. Der Operand muss einen arithmetischen Typ besitzen.

9. Präprozessor

Vor dem Zeichen #, das eine Präprozessorsteueranweisung einleitet, koennen in dieser Zeile beliebig viele Leerzeichen und Tabulatorzeichen stehen.

Bei einer Steueranweisung der Form

**#define** <bezeichner> <zeichenfolge>

bzw.

**#define** <makro_bezeichner>(<par$_1$>,..., <par$_n$>) <zeichenfolge>

erfolgt keine Ersetzung von <bezeichner>, <makro_bezeichner> und <par$_i$>, wenn sie innerhalb <zeichenfolge> in Zeichenkonstanten oder Zeichenkettenkonstanten vorkommen.

Ebenso wird keine Ersetzung des Bezeichners <bezeichner> oder <makro_bezeichner> vorgenommen, wenn er im zugehörigen Ersetzungstext <zeichenfolge> auftritt, d.h. es erfolgt keine rekursive Auswertung.

Ein zusätzliches Mittel zur Steuerung der bedingten Compilerung ist mit einer Steueranweisung der Form

```
...
#elif <konst_ausdruck>
...
```

gegeben. Zum Beispiel wird bei einem Eingabetext

```
...
#if  defined(I80286)
     printf("Prozessortyp: Intel 80286\n");
#elif  defined(M68000)
     printf("Prozessortyp: Motorola 68000\n");
#elif  defined(Z8000)
     printf("Prozessortyp: Zilog 8000\n");
#endif
...
```

nur eine der printf-Anweisungen an den Compiler weitergereicht, abhängig davon, welcher Bezeichner vorher mittels **#define** definiert wurde. Sind z.B. I80286 und M68000 nicht definiert, dafür aber Z8000, so übergibt der Präprozessor die dritte printf-Anweisung an den Compiler. (Natürlich werden alle Zeilen übergangen, wenn keiner der Bezeichner definiert ist.)

## Literaturverzeichnis


/1/ Anderson, B.: Type Syntax in the Language C. ACM SIGPLAN Notices, Vol. 15, No. 3, 1980, pp. 21-27

/2/ AT&T UNIX System V Programming Guide, Release 2.0, Chapter 2: C Language Definition

/3/ Baggenstos, T.; Marty, R.; Mergler, B.; Schnorf, P.: UNIX als Basis für Softwareentwicklung. Berlin - Heidelberg: Springer-Verlag 1986

/4/ Binding, C.: Cheap Concurrency in C. ACM SIGPLAN Notices, Vol. 20, No. 9, 1985, pp. 21-26

/5/ Boyd, S.: Modular C. ACM SIGPLAN Notices, Vol. 18, No. 4, 1983, pp. 45-54

/6/ Boyd, S.: Free and Bound Generics: Two Technics for Abstract Data Types in Modular C. ACM SIGPLAN Notices, Vol. 19, No. 3, 1984, pp. 12-20

/7/ Budd, T.A.: An Implementation of Generators in C. Computer Languages, Vol 7, No. 2, 1982, pp. 69-87

/8/ Christian, K.: The UNIX Operating System. New York: John Wiley & Sons 1983

/9/ Classen, L.; Oefler, U.: UNIX und C - Ein Anwenderhandbuch. Berlin: VEB Verlag Technik 1987 und Heidelberg: Dr. Alfred Hüthig Verlag 1987

/10/ Clauss, M.; Fischer, G.: Sprachbeschreibung C unter den Betriebssystemen MUTOS-8000 und MUTOS-1630. VEB Robotron Buchungsmaschinenwerk Karl-Marx-Stadt, 1985

/11/ Ehrig, M.; Fischer, G.: Zur Weiterentwicklung des ESER-C-Compilers. edv-Aspekte, Heft 3, 1985, S. 37-39

/12/ Einert, E.: Die Programmiersprache C. rechentechnik/datenverarbeitung, Heft 9, 1984, S. 9-13

/13/ Feldman, S.I.: Make - A Program for Maintaining Computer Programs. UNIX Programmer's Manual (Seventh Edition) 1979

/14/ French, D.D.: Computer System Design: C Language. Electronic Design, May 26, 1983, pp.165-170

/15/ French, D.D.: C Language Procedures. Electronic Design, June 23, 1983, pp. 143-148

/16/ Feuer, A.R.: The C Puzzle Book. Englewood Cliffs, N. J.: Prentice-Hall, Inc. 1982

/17/ Feuer, A.R.; Gehani, N.: A Comparison of the Programming Languages C and PASCAL. ACM Computing Surveys, Vol. 14, No. 1, 1982, pp. 73-92

/18/ Fitzhorn, P.A.; Johnson, G.R.: C: Toward a Concise Syntactic Description. ACM SIGPLAN Notices, Vol. 16, No. 12, 1981, pp. 14-21

/19/ Gehani, N.: C: An Advanced Introduction. Rockville, MD: Computer Science Press 1984

/20/ Hoare, C.A.R.: Quicksort. Computer Journal, Vol. 5, No. 1, pp. 10-15

/21/ Johnson, S.C.: Lint, a Program Checker. UNIX Programmer's Manual (Seventh Edition) 1979

/22/ Johnson, S.C.; Kernighan, B.W.: The C Language and Models for Systems Programming. Byte, Vol. 8, No. 8, 1983, pp. 48-60

/23/ Johnson, S.C.; Ritchie, D.M.: Portability of C Programs and the UNIX System. Bell Systems Technical Journal, Part 2, Vol. 57, No. 6, 1978, pp. 2021-2048

/24/ Joyce, J.: A C Language Primer. Part 1: Constructs and Conventions. Byte, Vol. 8, No. 8, pp. 64-78

/25/ Joyce, J.: A C Language Primer. Part 2: Tool Building in C. Byte, Vol. 8, No. 9, pp. 289-302

/26/ Katzenelson, J.: Higher Level Programming and Data Abstractions - A Case Study Using Enhanced C. Software - Practice and Experience, Vol. 13, No. 7, 1983, pp. 577-595

/27/ Katzenelson, J.: Introduction to Enhanced C. Software - Practice and Experience, Vol. 13, No. 7, 1983, pp. 551-576.

/28/ Kern, C.O.: Five Compilers for CP/M-80. Byte, Vol. 8, No. 8, 1983, pp. 110-130

/29/ Kern, C.O.: Another Look at CP/M-80 C Compilers. Byte, Vol. 9, No. 6, 1984, pp. 303-320

/30/ Kernighan, B.W.; McIlroy, M.D.(Eds.): UNIX Programmers Manual (Seventh Edition). Murray Hill, N. Y.: Bell Laboratories 1979

/31/ Kernigan, B.W.; Plauger, P.J.: Software Tools in PASCAL. Reading, Mass.: Addison-Wesley 1981

/32/ Kernighan, B.W.; Pike, R.: The UNIX Programming Environment. Englewood Cliffs, N. J.: Prentice-Hall, Inc. 1984

/33/ Kernighan, B.W.; Ritchie, D.M.: UNIX Programming - Second Edition. UNIX Programmer's Manual (Seventh Edition) 1979

/34/ Kernighan, B.W.; Ritchie, D.M.: The C Programming Language. Englewood Cliffs, N. J.: Prentice-Hall, Inc. 1978

/35/ Kilian, M.F.: A Conditional Region Pre-Processor for C Based on the Owicki and Gries Scheme. ACM SIGPLAN Notices, Vol. 20, No. 4, 1985, pp.42-56

/36/ Lee, P.A.: Exception Handling in C Programs. Software - Practice and Experience, Vol. 13, No. 5, 1983, pp.389-407

/37/ Linhart, J: Managing Software Development with C. Byte, Vol.8, No.8, pp. 172-182

/38/ Miller, P.; Watson, A.J.: The Power and the Portability: Evaluating the C Programming Language. Electronics, April 19, 1984, pp. 152-154

/39/ Müller, C.M.: C - Die UNIX-Implementierungssprache. In: Kreifelts, Schnupp (Hrsg.): UNIX Konzepte und Anwendungen. Berichte des German Chapter of the ACM, Band 12, S. 49-65, Stuttgart: B.G. Teubner 1983

/40/ Phraner, R. A.: Nine C Compilers for the IBM PC. Byte,Vol. 8, No. 8, pp. 134-168

/41/ Richards, M.: BCPL: A Tool for Compiler Writing and Systems Programming. Proceedings AFIPS SCJJ, Vol. 34, 1969, pp. 557-566

/42/ Ritchie, D.M.; Johnson, S.C.; Lesk, M.E.; Kernighan, B.W.: The C Programming Language. Bell Systems Technical Journal, Part 2, Vol. 57 No. 6, 1978, pp. 1991-2019

/43/ Ritchie, D.M.; Thompson, K.: The UNIX Time Sharing System. Communications of the ACM, Vol. 17, No. 6, 1974, pp. 363-375

/44/ Ritchie, D.M.: A Tour Through the UNIX C Compiler. UNIX Programmer's Manual (Seventh Edition) 1979

/45/ Rochkind, M.J.: The Source Code Control System. IEEE Transactions on Software Engineering SE-1, No. 4, 1975, pp. 364-370

/46/ Rosler, L.: The Evolution of C - Past and Future. AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, 1984, pp. 1685-1700

/47/ Stroustrup, B.: Adding Classes to the C Language: An Exercise in Language Evolution. Software - Practice and Experience, Vol. 13, No. 7, 1983, pp. 139-161

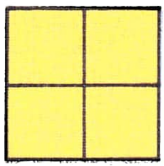
/48/ Stroustrup, B.: Data Abstractions in C. AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, 1984, pp. 1701-1732

/49/ Tsujino, Y.; Ando, M.; Araki, T.; Tokura, N.: Concurrent C : A Programming Language for Distributed Multiprocessor Systems. Software - Practice and Experience, Vol. 14, No. 11, 1984, pp. 1061-1078

/50/ X/OPEN Group Members: X/OPEN Portability Guide. Amsterdam: Elsevier Science Publishers 1985

/51/ ANSI X3J11 Language Subcommittee: Draft Proposed Standard - Programming Language C, July 1984

# Sachwörterverzeichnis



Die Autoren beschreiben den vollständigen Umfang der Programmiersprache C und bieten dem Leser ein Lern- und Arbeitsbuch für den Einstieg in diese aktuelle und wichtige Programmiersprache.
Nach einer Einführung in die grundlegenden Sprachelemente erfolgt die Behandlung der für C charakteristischen Aspekte anhand vieler Beispielfunktionen und Programme. Außerdem werden spezielle Fragen der Anwendung von C in UNIX-kompatiblen Programmierumgebungen diskutiert und die Funktionen der Standardbibliothek vorgestellt.

ISBN 3-341-00864-0